

Ultimate Python Programming

Learn Python with 650+ programs, 900+ practice questions, and 5 projects



Deepali Srivastava



Ultimate Python Programming

Learn Python with 650+ programs, 900+ practice questions, and 5 projects



Deepali Srivastava



Ultimate Python Programming

*Learn Python with 650+ programs,
900+ practice
questions, and 5 projects*

Deepali Srivastava



www.bpponline.com

First Edition 2024

Copyright © BPB Publications, India

ISBN: 978-93-55516-558

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Dedicated to

Sri Anjaneya Swamy

About the Author

Deepali Srivastava has a Master's degree in Mathematics and is an author and educator in the field of computer science and programming. Her books "C in Depth" and "Data Structures Through C in Depth" are widely used as reference materials by students, programmers and professionals looking to enhance their understanding of programming languages and data structures. These books are known for their clarity, depth of coverage, and practical approach to learning. In addition to her writing, Deepali Srivastava has been involved in creating online video courses on Data structures and Algorithms, Linux and Python programming. Her books and courses have helped 350,000+ students learn computer science concepts. Her work has been appreciated by students and has been a valuable resource for those looking to build their programming skills.

Acknowledgement

I would like to thank God for blessing me with the opportunity and inspiration to write this book, and for giving me the strength to do it.

I am grateful to my husband Suresh Kumar Srivastava for always believing in my capabilities and consistently inspiring me to give my best. He introduced me to book writing and helped me unleash my potential. His thoughtful suggestions and feedback helped me improve the content and presentation of this book.

I would like to thank my parents, my brother and my sister-in-law for their unwavering love and support. Blessings of my parents and late parents-in-law are a major source of my inner strength.

I am indebted to my teachers in my journey of education, especially my teachers and friends in MJP Rohilkhand University Bareilly, where I got introduced to the world of programming.

I extend my appreciation to the readers of my books and students of my online courses for their interest in my work, and for their appreciation and suggestions. Any sort of feedback is valuable to me and helps me in improving my work and creating better content.

I am grateful to the BPB publication team for their guidance and support provided during every step of the publishing journey. Special appreciation goes to the editing team, layout team, and all other contributors involved in bringing this book to life.

Preface

Python is a widely used general-purpose programming language. Its popularity can be attributed to its simplicity and a rich set of powerful features. The clean and intuitive syntax makes it an excellent choice for novices, allowing them to grasp the fundamentals of programming quickly, and the advanced features make it appealing to experienced programmers too. It can run on various platforms, including Windows, macOS, and Linux. Since it is an open-source software, it is freely available to all.

The widespread usage of Python is evident in the technology world, with major companies and organizations such as Google, Amazon, Instagram, Facebook, and NASA using it in different ways. Whether you are involved in machine learning, data science, artificial intelligence, scientific computing, automation or you need to create robust web applications and games, Python provides the necessary tools and resources. The extensive collection of libraries available in Python can be effectively utilized across diverse domains. Therefore, adding Python to your skill set can greatly enhance your capabilities and open up numerous opportunities in various fields.

This book provides a thorough and comprehensive introduction to Python, focusing on the core programming concepts and problem-solving skills required for building a solid foundation in programming. Throughout the book, there are numerous programming examples and end-of-chapter exercises to give you a hands-on experience. The exercises include multiple-choice questions and programming problems; multiple-choice questions will assess both your memory and comprehension of the topic, while the programming exercises will provide you with a chance to apply the acquired concepts. The book includes coding conventions and best practices for writing efficient, readable, and maintainable code. The code in the book is written and tested using Python version 3.11, which is the most recent version at the time of publishing the book.

Python is easy to learn. You can start writing Python programs within a few days. However, if you wish to leverage all the powerful features of Python, a more in-depth exploration is required. The content in this book can assist you in achieving that. This book includes 21 chapters that gradually introduce new topics so learners can proceed at a sustainable pace. If you are a beginner, start from the first chapter and go through all the chapters in order, and work out the examples and exercises along the way. If you have a working knowledge of Python, you can quickly browse through the initial chapters and then randomly jump to topics that are new to you or that you want to master. However, I would still recommend reading the chapters in sequence to get the most out of the book. If you are transitioning from some other language, you might be tempted to skip the initial information, but I would suggest you go through all the basic details to avoid any confusion later. Here is a brief summary of the chapters presented in the book.

[Chapter 1](#) is an introduction to Python and shows the installation process. [Chapter 2](#) covers the fundamental elements of Python, such as data types, variables, input, output, and many other basic concepts you need to get started in Python. [Chapter 3](#) provides a detailed explanation of strings that represent textual data in Python. [Chapters 4](#) and [5](#) cover the container types: lists, tuples, dictionaries, and sets. [Chapter 6](#) provides an insight into conditional execution. In [chapter 7](#), we will see how to perform repetitive tasks using loops, and [chapter 8](#) discusses some common looping techniques in Python. [Chapter 9](#) introduces the concept of comprehensions which help us write shorter and readable code.

[Chapter 10](#) contains detailed coverage of functions. We will see how to create our own functions and will discuss parameters, arguments, arguments passing, function objects, and many other details about functions. [Chapter 11](#) shows how to create and use modules and packages. [Chapter 12](#) is about namespaces and scoping rules. [Chapter 13](#) shows how to write programs that can create files, write data into files, and read the data stored in files. [Chapters 14, 15, and 16](#) provide you with a strong understanding of the object-oriented concepts. We will discuss classes, objects, methods, inheritance, polymorphism, and magic methods. [Chapters 17 and 18](#) are devoted to advanced topics like iterators, generators, and decorators. [Chapter 19](#) is about functional programming and lambda functions. [Chapter 20](#) shows how to handle run-time errors in Python, and [Chapter 21](#)

discusses context managers that are used to automate common resource management patterns.

At the end of each chapter, you will find exercises, and their solutions are provided at the end of the book. I would suggest that you try to solve these exercises by yourself before looking at the solution. Solving exercises and writing code will help you to internalize the concepts presented in the book.

Some typographical conventions are followed throughout the book for a good reading experience. The code snippets and programs in the book appear in `this font` to differentiate them from the regular text. Program elements, such as variable names, types, etc., within the regular text, are in `this font`. Any output produced by the code on the screen as a result of running a program or anything that the user has to input through the screen appears in `this font`.

My aim was to write an absolute hands-on book that is simple enough to follow and yet gives detailed knowledge. Reading this book will be a breeze, yet it will give you a comprehensive knowledge of Python and instill the confidence to excel in any written test, interview, or professional work. Programming is fun only when you get your hands dirty with code. Reading a book is not enough for learning programming. I highly recommend that you try the coding examples and exercises presented in the book. The efforts you put in to strengthen your fundamentals of core programming concepts will take you a long way in your software development journey.

By the end of this book, you will develop a strong foundation in core Python skills and will get the ability to explore the vast range of functionalities offered by the standard library and third-party libraries. As you progress, you will continue to be amazed by the capabilities of Python and the remarkable libraries available. With your newfound skills you can venture into diverse fields like data science or machine learning. Moreover, if this is the first programming language you are learning, equipped with the foundation of programming concepts and problem-solving skills, you can easily learn any other programming language.

After using this book as a tutorial to learn the language, you can always refer to it as a handy resource whenever you need to recall or review any concept and apply it to your work.

Writing this book was a very enjoyable, insightful, and amazingly satisfying journey for me and I am sure my readers will have a similar experience while reading the book. I hope you enjoy reading the book and start loving Python.

Happy programming!

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the *Coloured Images* of the book:

<https://rebrand.ly/z815rfg>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Ultimate-Python-Programming>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to Python

- 1.1 What makes Python so popular
- 1.2 Python implementation
- 1.3 Installing Python
- 1.4 Python Interactive Mode
- 1.5 Executing a Python Script
- 1.6 IDLE
- 1.7 Getting Help

2. Getting Started

- 2.1 Identifiers
- 2.2 Python Types
- 2.3 Objects
- 2.4 Variables and assignment statement
- 2.5 Multiple and Pairwise Assignments
- 2.6 Deleting a name
- 2.7 Naming convention for constants
- 2.8 Operators
 - 2.8.1 *Arithmetic operators*
 - 2.8.2 *Relational operators*
 - 2.8.3 *Logical operators*
 - 2.8.4 *Identity operators*
 - 2.8.5 *Membership operators*

- 2.9 Augmented assignment statements
- 2.10 Expressions
- 2.11 Order of operations: Operator Precedence and Associativity
- 2.12 Type Conversion
- 2.13 Statements
- 2.14 Printing Output
- 2.15 Getting user input
- 2.16 Complete programs
- 2.17 Comments
- 2.18 Indentation in Python
- 2.19 Container types
- 2.20 Mutable and Immutable Types
- 2.21 Functions and methods
- 2.22 Importing
- 2.23 Revisiting interactive mode
- 2.24 Errors
- 2.25 PEP8
- Exercise

- 3.1 Indexing
- 3.2 Strings are immutable
- 3.3 String Slicing
- 3.4 String Concatenation and Repetition
- 3.5 Checking membership
- 3.6 Adding whitespace to strings
- 3.7 Creating multiline strings

- 3.8 String methods
- 3.9 Case-changing methods
- 3.10 Character classification methods
- 3.11 Aligning text within strings
- 3.12 Removing unwanted leading and trailing characters
- 3.13 Searching and replacing substrings
- 3.14 Chaining method calls
- 3.15 String comparison
- 3.16 String conversions
- 3.17 Escape Sequences
- 3.18 Raw string literals
- 3.19 String formatting
- 3.20 String formatting using the format() method of string class
- 3.21 Representation of text - character encodings
- Exercise

4. Lists and Tuples

- 4.1 Accessing individual elements of a list by indexing
- 4.2 Getting parts of a list by slicing
- 4.3 Changing an item in a list by index assignment
- 4.4 Changing a Portion of the list by slice assignment
- 4.5 Adding an item at the end of the list by using append()
- 4.6 Adding an item anywhere in the list by using insert()
- 4.7 Adding multiple items at the end by using extend() or +=
- 4.8 Removing a single element or a slice by using the del statement
- 4.9 Removing an element by index and getting it by using pop()
- 4.10 Removing an element by value using remove()
- 4.11 Removing all the elements by using clear()

- 4.12 Sorting a List
- 4.13 Reversing a List
- 4.14 Finding an item in the list
- 4.15 Comparing Lists
- 4.16 Built-in functions used on lists
- 4.17 Concatenation and Replication
- 4.18 Using a list with functions from the random module
- 4.19 Creating a list
- 4.20 Using range to create a list of integers
- 4.21 Using the repetition operator to create a list of repeated values
- 4.22 Creating a list by splitting a string
- 4.23 Converting a list of strings to a single string using join()
- 4.24 List of Lists (Nested lists)
- 4.25 Copying a list
- 4.26 Shallow copy and deep copy
- 4.27 Repetition operator with nested lists
- 4.28 Tuples
- 4.29 Tuple packing and unpacking
- Exercise

5. Dictionaries and Sets

- 5.1 Dictionaries
- 5.2 Adding new key-value pairs
- 5.3 Modifying Values
- 5.4 Getting a value from a key by using the get() method
- 5.5 Getting a value from a key by using the setdefault() method
- 5.6 Getting all keys, all values, and all key-value pairs
- 5.7 Checking for the existence of a key or a value in a dictionary

- 5.8 Comparing dictionaries
- 5.9 Deleting key-value pairs from a dictionary
- 5.10 Creating a Dictionary at run time
- 5.11 Creating a dictionary from existing data by using dict()
- 5.12 Creating a dictionary by using the fromkeys() method
- 5.13 Combining dictionaries
- 5.14 Nesting of dictionaries
- 5.15 Aliasing and Shallow vs. Deep Copy
- 5.16 Introduction to sets
- 5.17 Creating a set
- 5.18 Adding and Removing elements
- 5.19 Comparing sets
- 5.20 Union, intersection, and difference of sets
- 5.21 Frozenset
- Exercise

6. Conditional Execution

- 6.1 if statement
- 6.2 else clause in if statement
- 6.3 Nested if statements
- 6.4 Multiway selection by using elif clause
- 6.5 Truthiness
- 6.6 Short circuit behavior of operators and and or
- 6.7 Values returned by and and or operators
- 6.8 if else operator
- Exercise

7. Loops

- 7.1 while loop

- 7.1.1 *Indentation matters*
 - 7.1.2 *Removing all occurrences of a value from the list using the while loop*
 - 7.1.3 *while loop for input error checking*
 - 7.1.4 *Storing user input in a list or dictionary*
- 7.2 for loop
 - 7.2.1 *Iterating over a string with for loop*
 - 7.2.2 *Unpacking in for loop header*
 - 7.2.3 *Iterating over dictionaries and sets*
 - 7.2.4 *Iterating through a series of integers*
- 7.3 Nesting of Loops
 - 7.3.1 *Using nested loops to generate combinations*
 - 7.3.2 *Iterating over nested data structures*
- 7.4 Premature termination of loops using the break statement
- 7.5 continue statement
- 7.6 else block in Loops
- 7.7 pass statement
- 7.8 for loop vs. while loop
- Exercise

8. Looping Techniques

- 8.1 Iterating in sorted and reversed order
- 8.2 Iterating over unique values
- 8.3 Index-Based for loops
- 8.4 Making in-place changes in a list while iterating
- 8.5 Skipping some items while iterating
- 8.6 Using range and len combination to shuffle a sequence
- 8.7 enumerate function
- 8.8 Iterating over multiple sequences using zip
- 8.9 Modifying a collection while iterating in a for loop

8.10 Infinite loop with break

8.11 Avoiding complex logical conditions using break

Exercise

9. Comprehensions

9.1 List Comprehensions

9.2 if clause in list comprehension

9.3 Ternary operator in list comprehension

9.4 Modifying a list while iterating

9.5 Getting keys from values in a dictionary using list comprehension

9.6 Using list comprehensions to avoid aliasing while creating lists of lists

9.7 Multiple for clauses and Nested list Comprehensions

9.8 Extracting a column in a matrix

9.9 Dictionary Comprehensions

9.10 Inverting the dictionary

9.11 Set Comprehensions

9.12 When not to use comprehensions

Exercise

10. Functions

10.1 Function Definition

10.2 Function call

10.3 Flow of control

10.4 Parameters and Arguments

10.5 No type checking of arguments

10.6 Local Variables

10.7 return statement

10.8 Returning Multiple Values

10.9 Semantics of argument passing

10.9.1 Why study argument passing

10.9.2 Pass by assignment

10.9.3 Assignment inside function rebounds the parameter name

10.9.4 Immutables vs Mutables as arguments

10.9.5 How to get the changed value of an immutable type

10.9.6 How to prevent change in mutable types

10.9.7 Digression for programmers from other languages

10.9.8 Advantages of Python's information passing

10.10 Default Arguments

10.11 Default arguments that could change over time

10.12 Positional and Keyword Arguments

10.13 Unpacking Arguments

10.14 Variable number of positional arguments

10.15 Variable number of keyword arguments

10.16 Keyword-only arguments

10.17 Positional-Only Arguments

10.18 Multiple Unpackings in a Python Function Call

10.19 Arguments and Parameters summary

10.20 Function Objects

10.21 Attributes of a function

10.22 Doctrings

10.23 Function Annotations

10.24 Recursive Functions

Exercise

11. Modules and Packages

11.1 Modules

11.2 Types of modules

- 11.3 Exploring modules
- 11.4 Creating and naming a new module
- 11.5 Importing a module
- 11.6 Importing all names from a module
- 11.7 Restricting names that can be imported
- 11.8 Importing individual names from a module
- 11.9 Using an alias while importing
- 11.10 Documenting a module
- 11.11 Module search Path
- 11.12 Module object
- 11.13 Byte compiled version of a module
- 11.14 Reloading a module
- 11.15 Scripts and modules
- 11.16 Packages
- 11.17 Importing a package and its contents
- 11.18 Subpackages
- 11.19 Relative imports
- Exercise

12. Namespaces and Scope

- 12.1 Namespaces
- 12.2 Inspecting namespaces
- 12.3 Scope
- 12.4 Name Resolution
- 12.5 global statement
- 12.6 nonlocal statement
- Exercise

13. Files

- 13.1 Opening a File
- 13.2 File opening modes
- 13.3 Buffering
- 13.4 Binary and Text Files
- 13.5 Closing a file
- 13.6 with statement
- 13.7 Random Access
- 13.8 Using seek in text mode
- 13.9 Calling seek in append mode
- 13.10 Reading and writing to the same file
- 13.11 Reading a File using read()
- 13.12 Line oriented reading
- 13.13 Writing to a file
- 13.14 Redirecting output of print to a file
- 13.15 Example Programs
- 13.16 File Related Modules
- 13.17 Command Line Arguments
- 13.18 Storing and Retrieving Python objects using pickle
- Exercise
- Project : Hangman Game

14. Object Oriented Programming

- 14.1 Programming Paradigms
- 14.2 Introduction to object-oriented programming
- 14.3 Defining Classes and Creating Instance Objects
- 14.4 Adding methods to the class
- 14.5 Adding instance variables
- 14.6 Calling a method inside another method

- 14.7 Common pitfalls
- 14.8 Initializer
- 14.9 Data Hiding
- 14.10 Class Variables
- 14.11 Class and object namespaces
- 14.12 Changing a class variable through an instance
- 14.13 Class Methods
- 14.14 Creating alternative initializers using class Methods
- 14.15 Static Methods
- 14.16 Creating Managed Attributes using properties
 - 14.16.1 Creating read only attributes using properties*
 - 14.16.2 Creating Computed attributes using properties*
 - 14.16.3 Deleter method of property*
- 14.17 Designing a class
- Exercise
- Project : Quiz creation
- Project : Snakes and Ladders Game
- Project : Log in system

15. Magic Methods

- 15.1 Overloading Binary Arithmetic operators
- 15.2 Reverse methods
- 15.3 In-place methods
- 15.4 Magic Methods for comparison
- 15.5 Comparing objects of different classes
- 15.6 String representation of an instance object
- 15.7 Construction and destruction of objects
- 15.8 Making instance objects callable
- 15.9 Overloading type conversion functions

15.10 List of magic methods

Exercise

Project : Date Class

16. Inheritance and Polymorphism

16.1 Inheriting a class

16.2 Adding new methods and data members to the derived class

16.3 Overriding a base Method

16.4 Invoking the base class methods

16.5 Multilevel Inheritance

16.6 object class

16.7 Multiple Inheritance

16.8 Method Resolution Order (MRO)

16.9 super and MRO

16.10 Polymorphism

16.11 Abstract Base classes

16.12 Composition

Exercise

17. Iterators and Generators

17.1 Iterables

17.2 Iterators

17.3 for loop Iteration Context – How for loop works

17.4 Iteration Tools

17.5 Iterator vs Iterable

17.6 Creating your own Iterator

17.7 Making your class Iterable

17.8 Some More Iterators

17.9 Lazy evaluation

- 17.10 itertools Module
- 17.11 Generators
- 17.12 Generator function vs Normal function
- 17.13 Generator expressions
- Exercise

18. Decorators

- 18.1 Prerequisites for understanding decorators
- 18.2 Introduction to decorators
- 18.3 Writing your first decorator
- 18.4 Applying your decorator to multiple functions
- 18.5 Automatic decoration syntax
- 18.6 Decorator Example: Timer
- 18.7 Decorator Example: Logger
- 18.8 Decorator Example: Counting function calls
- 18.9 Applications of decorators
- 18.10 Decorating functions that take arguments
- 18.11 Returning values from decorated functions
- 18.12 Decorator Example: Checking return values
- 18.13 Decorator Example: Checking argument values
- 18.14 Applying Multiple Decorators
- 18.15 Preserving metadata of a function after decoration
- 18.16 General template for writing a decorator
- 18.17 Decorators with parameters
- 18.18 General template for writing a decorator factory
- 18.19 Decorator factory example
- 18.20 Applying decorators to imported functions
- 18.21 Decorating classes

18.22 Class Decorators

18.23 Class Decorators with parameters

Exercise

19. Lambda Expressions and Functional Programming

19.1 Lambda expression

19.2 Comparing def statement and lambda expression

19.3 Examples of lambda expressions

19.4 Using Lambda expressions

19.5 Using lambda expressions for returning function objects

19.6 Lambda expressions as closures

19.7 Creating jump tables using lambda functions

19.8 Using lambda expressions in sorted built-in function

19.9 Functional programming

19.10 map

19.11 map with multiple iterables

19.12 filter

19.13 Reducing an iterable

19.14 Built-in reducing functions

19.15 operator module

Exercise

20. Exception Handling

20.1 Types of Errors

20.2 Strategies to handle exceptions in your code

20.3 Error Handling by Python (Default exception handling)

20.4 Built-in Exceptions: Python Exceptions Class Hierarchy

20.5 Customized Exception Handling by using try...except

- 20.6 Catching multiple exceptions using multiple except handlers and single except handler
- 20.7 How to handle an exception
- 20.8 Guaranteed execution of finally block
- 20.9 else Block
- 20.10 Why do we need an else block
- 20.11 How to get exception details
- 20.12 Nested try statements
- 20.13 Raising Exception
- 20.14 Re-raising Exception
- 20.15 Chaining Exceptions
- 20.16 Creating your own exceptions in Python (Custom exceptions)
- 20.17 Assertions
- Exercise

21. Context Managers

- 21.1 with statement
- 21.2 Implementing our own context manager
- 21.3 Exception raised inside with block
- 21.4 Why we need with statement and context managers
- 21.5 Runtime context
- 21.6 Example: Sending output of a portion of code to a file
- 21.7 Example : Finding time taken by a piece of code
- 21.8 Using context managers in the standard library
- 21.9 Nested with statements and multiple context Managers
- 21.10 Implementing a context manager by using a decorator on a generator
- Exercise

Solutions

Index

Introduction to Python

1

Python is a widely used high-level and general-purpose programming language originally developed by Guido Van Rossum in the early 1990s in the Netherlands. It is maintained by a community of core developers who are actively engaged in its growth and advancement. Although the official logo of Python shows two intertwined snakes, it is not named after any snake. Van Rossum named this language after a 1970s comedy show 'Monty Python's Flying Circus'.

Python has three major versions; the initial version, Python 1.0, was released in January 1994. The second major version, Python 2.0, was released in 2000, and the third major version, Python 3.0, was released in 2008. Python 3 is not backward compatible with Python 2; this means that the code written in Python 2 may not work as expected in Python 3 without making some modifications. In this book, we will use Python 3. The latest release of Python is available on its official website www.python.org. Python is an open-source software, which means that it is free to use and distribute.

1.1 What makes Python so popular

Python is a general-purpose language used in a wide variety of domains. It is used extensively in different fields such as web development, data mining, artificial intelligence, image processing, robotics, network programming, developing user interfaces, database programming, scientific and mathematical computing, game programming, and even education. Most of the top companies and organizations, such as Google, Facebook, Amazon, and NASA, use Python in different ways. Let us see some of the key factors that contribute to Python's popularity.

Python is very easy to learn. It doesn't take much time to become productive with Python. This is why it is often the introductory programming language taught in many universities. Compared to languages such as C++ or Java, Python code tends to be more concise, requiring fewer lines of code to achieve the same functionality. Due to the simple syntax of Python, programmers can focus more on finding the solution to a problem instead of getting caught up in complex language features. Python uses indentation for grouping together statements, resulting in a visually clean layout that enhances code readability.

Python offers a convenient command line interface known as the 'Python interactive shell' or 'Python REPL' (Read-Eval-Print Loop). With the Python interpreter, you have the option to work interactively, allowing you to test and debug small sections of code in real-time. The interactive mode serves as a useful tool for experimenting and exploring Python's features.

One of the main advantages of Python is that it takes care of memory management automatically. Python's built-in memory management system allocates memory when needed and frees it up when it is no longer in use. Programmers do not have to worry about managing memory manually, as they would have to do in other languages like C or C++.

Python includes a vast standard library of modules; this is why the phrase 'Batteries included' is often used for Python. These modules contain code that you can use in your own programs. In addition to the extensive standard library, many third-party libraries are also available for use. Thus, you have access to lots of prewritten reusable code in the form of standard library modules and third-party modules, which can do most of the work for you and save you from reinventing the wheel. This code can be incorporated into your code to develop complex solutions with minimal effort. Whether you are working on web programming, creating graphics, analyzing data, performing mathematical calculations, engaging in scientific computing, or developing games, you will find reusable code modules that can help you achieve your goals.

Python supports multiple programming paradigms, including procedural, functional, and object-oriented programming. Thus, programmers have the flexibility to choose the coding structure that best suits their needs. The object-oriented features of Python are much easier to implement and are

more intuitive when compared to similar features found in other programming languages.

Python is a cross-platform and portable programming language, which means that programs written in Python can be developed and executed on various hardware platforms and operating systems. The same code can be executed on multiple platforms without making any significant changes. The cross-platform development minimizes the efforts required to adapt the programs to different systems and thus facilitates code reuse and sharing on different platforms.

Python has the capability to interact with software components written in other languages. Python code can call libraries written in C and C++, and it can also integrate with components developed in Java and .NET. This allows Python programmers to tap into the strengths and functionalities of other languages and libraries written in them. Python is also embeddable which means that Python code can be placed within the code of another language like C or C++.

Another reason for Python's popularity is its large base of active and supportive developer community. Community members are actively engaged in improving and enhancing the capabilities of Python as well as in developing various libraries and tools. There are numerous resources and extensive support available due to the vibrant community members.

Python has emerged as the preferred programming language for developers because of its ease of use and powerful features. It is suitable both for beginners and experts alike, and due to its versatility, it can be used in a variety of applications.

In the next section we will learn about Python implementations and will see what happens internally when a Python program is executed. While it is not necessary to have this knowledge in order to write and run programs, having a fundamental understanding of what occurs behind the scenes during program execution is beneficial for a comprehensive understanding of the language.

1.2 Python implementation

The terms C, C++, Basic, Java, or Python refer to programming languages, which are essentially sets of rules and specifications. In order to use these languages, they need to be implemented by creating software that allows us to write programs in that language and run them on a computer. The implementation of a language is the program that actually runs the code that you write in that language. An implementation translates the source code to native machine code instructions (binary 0s and 1s) so that the computer's processor can execute it.

There are primarily two approaches to implementing a programming language: compilation and interpretation. In compilation, a compiler translates the complete program code in one go to another language such as machine code or bytecode. If the translated code is machine code that is understood by the processor, then it is directly executed, and if it is bytecode, then it has to be again input to another interpreter or compiler. In interpretation, an interpreter translates the code to machine code one line at a time; a line of code is read, translated, and executed, then the next line is read, translated, and executed, and so on. The code is translated line by line at run time, so the interpreted implementations tend to be slower than the compiled ones, which translate the whole code at once.

An implementation of a language can be a compiler, interpreter, or a combination of both. A programming language can have multiple implementations, and these implementations can be written in different languages and can use different approaches to compile or interpret code. The notion of interpretation and compilation is associated with language implementation rather than the language itself; describing a language as compiled or interpreted is not technically correct. The language implementations that are written for a language are described as compiled or interpreted and not the language. Compilation or interpretation is not a part of the language specification; it is an implementation decision. The implementations of C and C++ mostly use the compilation approach, while Java, Python, and C# implementations generally use a combination of compilation and interpretation techniques. C and C++ compilers translate source code to machine code, which is executed directly by the processor.

Python has multiple implementations. The original and standard implementation of Python is CPython written in C language. It is the most widely used and up-to-date implementation of Python. When you download

Python software from the official site python.org, this is the implementation that you get. The other implementations are Jython written in Java, and IronPython written for the .NET platform. PyPy is the implementation that is written in RPython, which is a subset of Python.

The software that is used for running Python programs is referred to as Python interpreter. Let us understand how CPython interpreter combines the compilation and interpretation techniques to execute a Python program.

We write our Python code in a source file (.py file), but the computer cannot understand and execute this code; it can execute only machine code, which consists of instructions written in binary form (0s and 1s). The source code has to be converted to machine code so that the processor can execute it. The source code is not directly converted to machine code. It is first compiled into an intermediate form known as the bytecode. This bytecode is a low-level code that is Python-specific and platform-independent, but it is not understandable to the processor.

There is another software called Python Virtual Machine (PVM), that is responsible for executing this bytecode on a specific platform. The bytecode passes through the Python Virtual machine; it interprets this bytecode, which means that it converts the bytecode instructions to machine code instructions one by one and sends these machine code instructions to the processor for execution, and we get the output. So, the job of PVM is to convert the bytecode instructions to machine code instructions that the processor can understand and execute.

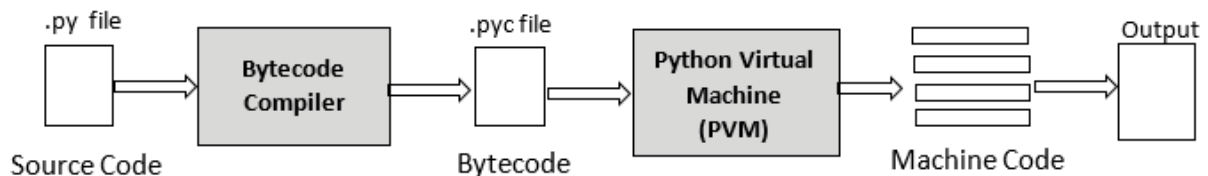


Figure 1.1: The execution of a Python program

This is what happens when we execute a Python program. The intermediate compilation step is hidden from the programmer; we can just type and run our program immediately. The programmer does not have to explicitly compile the code, so there is no separate compile time in Python; there is

only runtime. The compilation to bytecode is done to improve the efficiency as the bytecode can be interpreted faster than the original source code.

In this whole process, the bytecode compiler is a software that converts source code to bytecode, and PVM is a software that converts bytecode to machine code for the target platform. Python Virtual machine contains some platform-specific components that may be implemented differently for each platform. This allows the virtual machine to convert the bytecode into native machine code according to the platform. It abstracts away the underlying hardware and operating system details and thus provides a consistent runtime environment for Python programs across different platforms. Both the bytecode compiler and the virtual machine are part of the Python interpreter software and are included in your Python installation.

The intermediate bytecode is generally cached for faster execution. It is stored in .pyc or .pyo files inside a folder named `__pycache__` and the programmer can just ignore these files. When the program is run multiple times without modifying the source code, the compiled bytecode from the cached file is loaded and executed instead of re-compiling from source code to bytecode every time. This bytecode is stored only for imported files, not for the top-level scripts; we will see the difference between the two later in the book.

The Jython implementation translates Python code into Java bytecode, enabling its execution on a Java virtual machine. An advantage of Jython is its ability to directly access Java libraries. Similarly, IronPython is designed for the .NET framework and facilitates integration with .NET components.

Some implementations of virtual machines (bytecode interpreters) use just-in-time (JIT) compilation approach to speed up the interpretation process. The PyPy implementation of Python has better speed as it includes a just-in-time compiler for faster execution of the bytecode. Just-in-time compiler will compile the frequently executed blocks of bytecode to machine code and cache the result. Next time, when the virtual machine has to execute the same block of bytecode, the precompiled(cached) machine code is utilized and executed, resulting in faster execution. So, the JIT compiler uses the compilation approach to improve the efficiency of bytecode execution.

1.3 Installing Python

To download Python, visit the official website of Python. On the homepage, select the Downloads option to go to the download page, or you can directly go to www.python.org/downloads/. The website will automatically detect your operating system and provide a suitable installer that corresponds to your system's requirements, whether it be 32-bit or 64-bit. Click on the Download button to download the installer (.exe) file for the latest version of Python. At the time of writing this book, the latest version is 3.11.3. If you wish to download any previous version of Python, you can scroll down the page and click on the download button located next to the version number you desire.

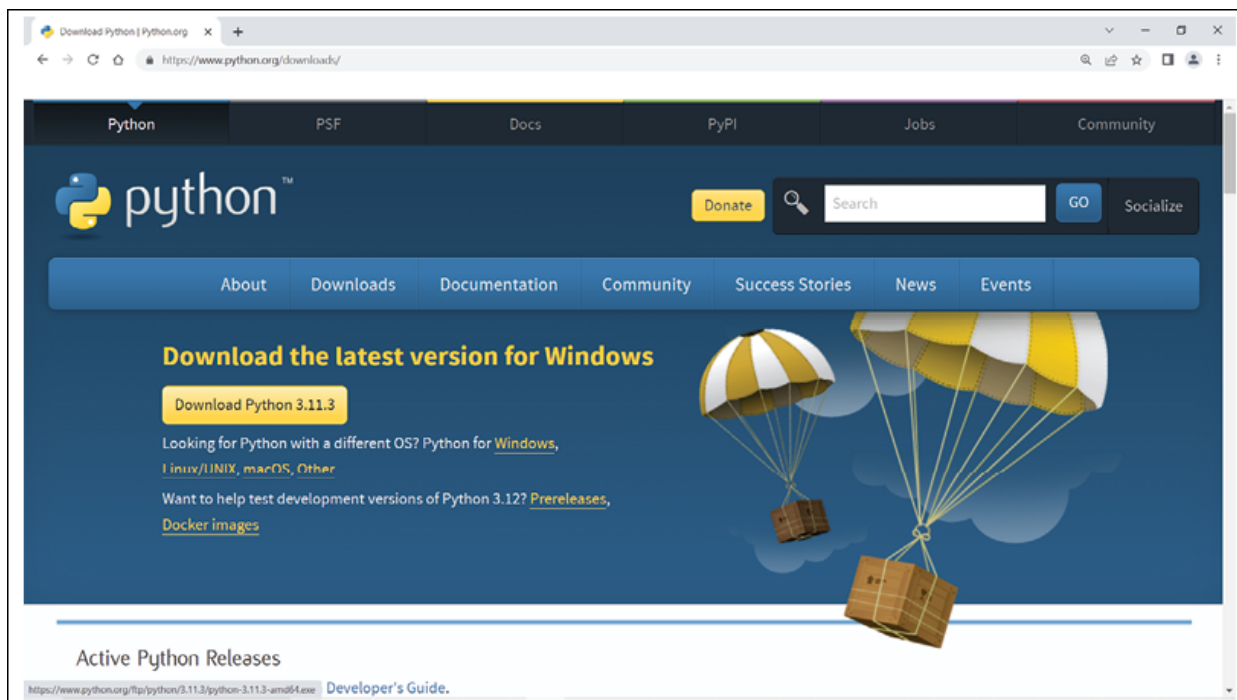


Figure 1.2: Official website of Python

Once the download is complete, double-click on the installer to execute it and begin the installation process. On the first screen of the installer, you will be presented with two choices: "**Install Now**" and "**Customize Installation**." Clicking on "**Install Now**" will install Python with the default features, while clicking on "**Customize Installation**" will allow you to change the installation location or install other optional and advanced features. The defaults should work well for now, so we will go with Install Now. Before clicking on Install Now, make sure to select the **Add python.exe to PATH** checkbox, as this will add Python to your system's

PATH environment variable and will enable you to run Python from the command prompt.

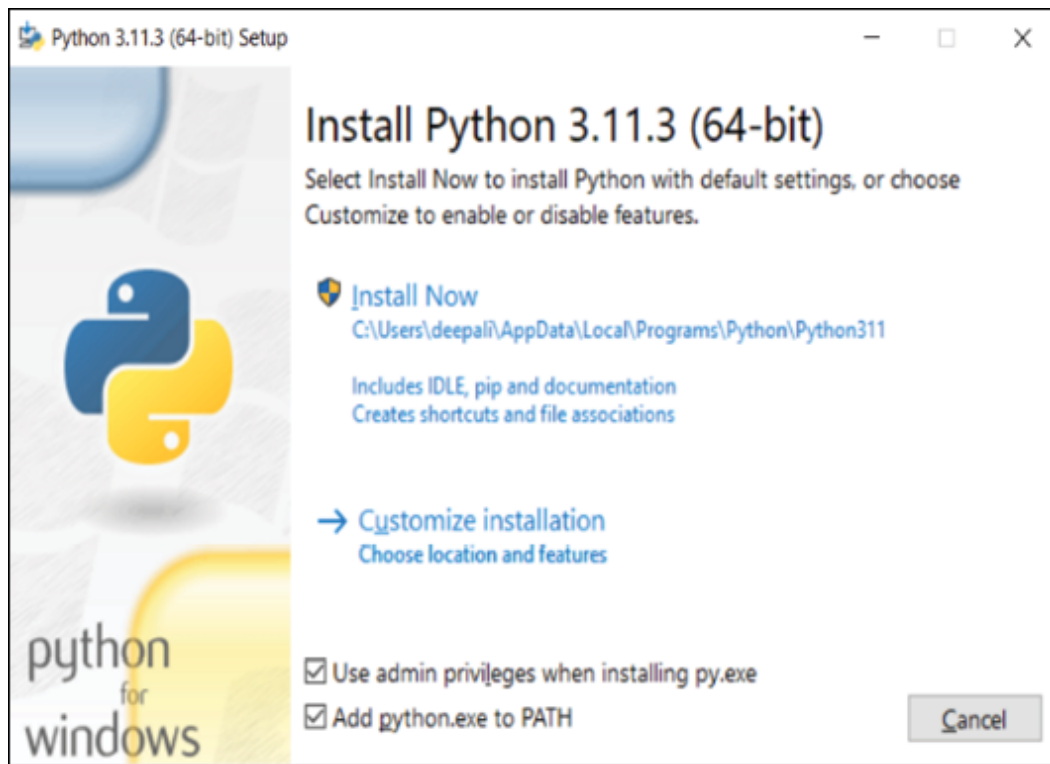


Figure 1.3: Installing Python

Click **Yes** if it asks for permission to make changes to your device. The installation begins, and all the required Python files, along with the standard library, will be installed on your system.

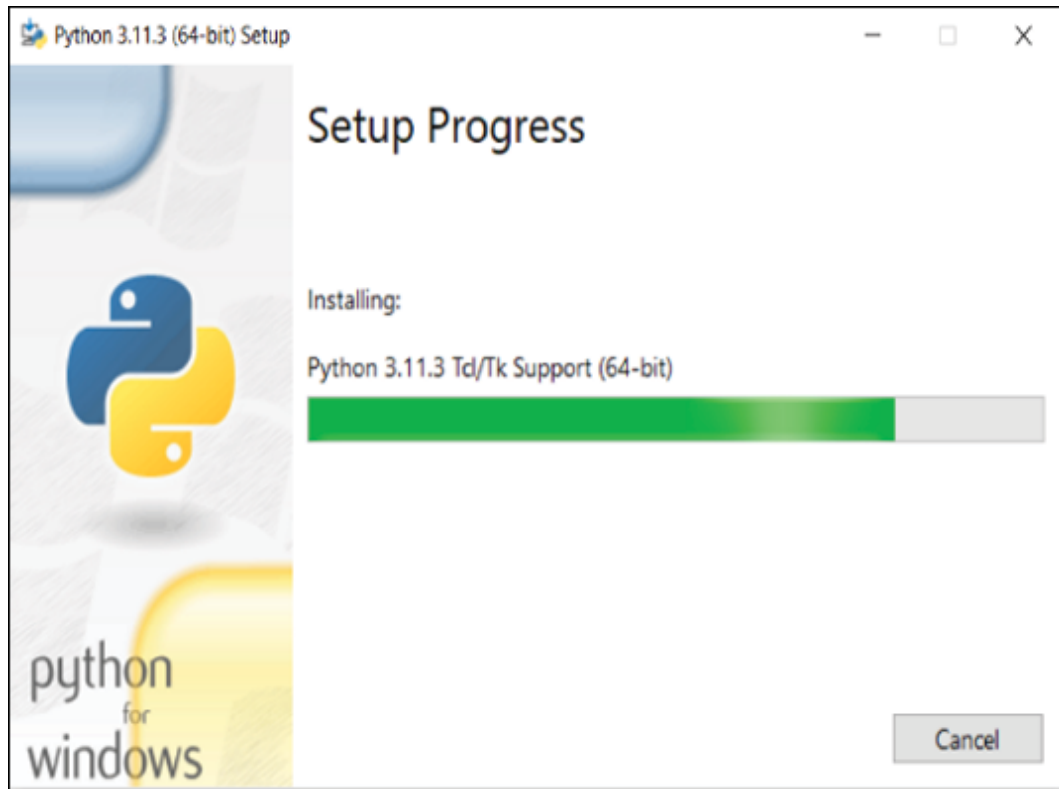


Figure 1.4: Installation in progress

After the installation is complete, the following pop-up box will appear. This shows that Python is installed on your system. Click on **Close** to complete the installation and exit the installer. The appearance of the images shown in the screenshots may vary depending on the version of Python that you choose to install.

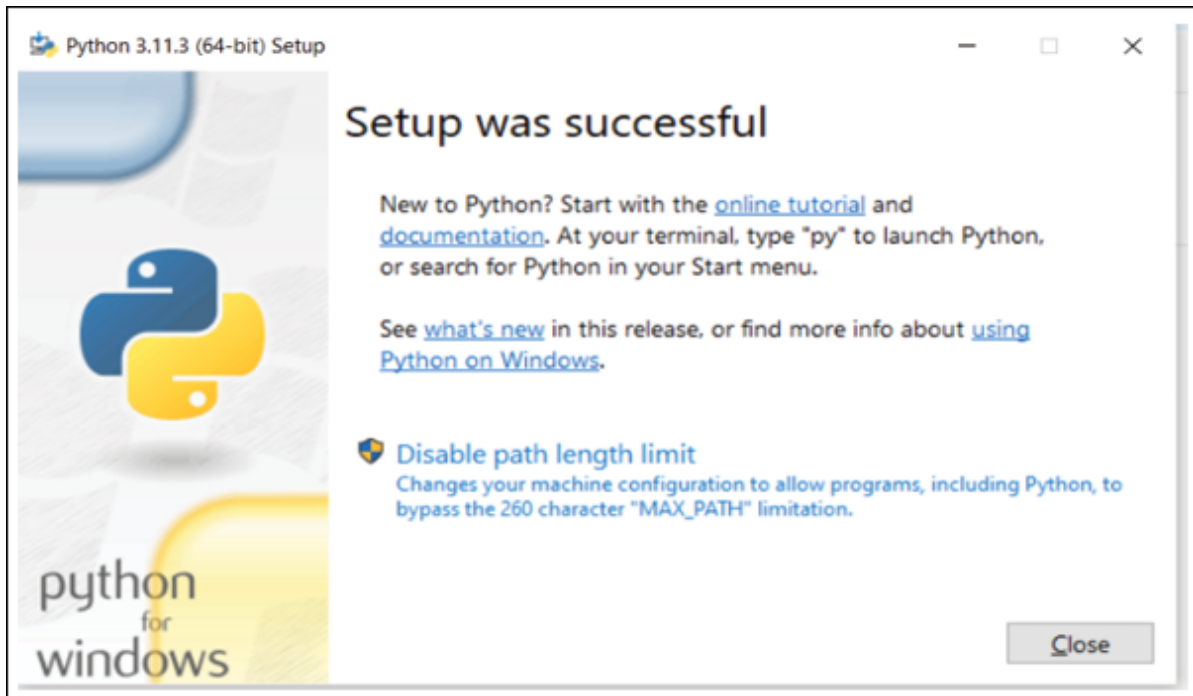


Figure 1.5: Installation successful

To verify the installation, write **cmd** in the Start search menu to open the command prompt window and type the command `python --version`. If Python has been successfully installed on your system, it will show the version of the Python installed. Now write `python` (all in lowercase) in the command window. You will see a line with some text describing the Python version, and after that, you will see a prompt with three greater-than signs (`>>>`). This is the Python shell prompt. Write `8 + 2` and press Enter; you will get the output as 10 on the next line. The prompt appears again; this time, write `print('Hello world')`, and the text **Hello world** will appear on the next line. This verifies that Python is up and working on your system. On this interactive Python shell, you can execute single statements of Python. To quit this Python shell and come back to your command prompt, type `quit()` or `exit()` or press Ctrl-Z.

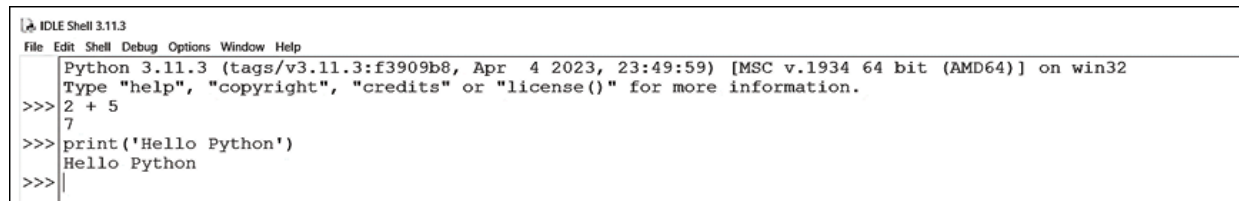
```
C:\Users\deepali>python --version
Python 3.11.3

C:\Users\deepali>python
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 8 + 2
10
>>> print('Hello World')
Hello World
>>> exit()

C:\Users\deepali>
```

Figure 1.6: Verifying installation on the command line

You can also verify your installation by opening the Integrated Development and Learning Environment application (IDLE), which is installed by default with Python. To open IDLE, type `idle` or `python` in the Start search menu and click on the IDLE app. If the installation is successful, IDLE will show an interactive Python shell window in which you can type Python commands at the shell prompt (`>>>`) and execute them.



```
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 2 + 5
7
>>> print('Hello Python')
Hello Python
>>> |
```

Figure 1.7: Verifying installation on IDLE

Installation on Mac is done in a similar way. Most macOS systems come with Python, but usually, it is the 2.x version. To check if Python is installed, type `python --version` on your terminal. To check if Python 3 is installed, type `python3 --version` on your terminal. If Python 3 is not installed, you can install it from the official website, and if it is installed, you can update it to get the latest version.

Visit the official Python website and download the installer package (.pkg file) that it offers for your system. After downloading, double-click on the installer to run the installation process. Proceed with the installation by following the on-screen instructions and accepting the defaults. You may need to enter your administrator password to authorize the installation.

After the installation process is complete, Python's installation folder will automatically open up. Inside this folder, you will find IDLE application, which, as we have seen, is the development environment that comes with Python. Double-click on this application to open it. If the installation is successful, IDLE will display the interactive Python shell. You can type `print('Hello world')` at the shell prompt to verify that it is functioning correctly. To confirm the installation on the terminal, open the Terminal application and type `python3 --version`, and press Enter. This should show the version of Python that you have installed. Type `python3` to open the Python shell, which shows the `>>>` prompt where you can start typing Python statements. You can close this Python shell by entering Ctrl D or typing `exit()`.

Installation on Linux can be done through the Package Manager specific to your distribution. Linux systems come with Python installed on them. To check whether Python is installed correctly or to check before installation whether Python is already there on the system, execute the following command irrespective of your operating system:

```
$ python3 --version
```

or

```
$ python --version
```

On Mac and Linux, the `python --version` command will mostly show the Python 2 version and the `python3 --version` command will show the Python 3 version.

After you have installed Python on your machine, you can either use an Integrated Development Environment (IDE) to write and execute your Python scripts, or you can write your script in a text editor and execute them on the command line. An IDE combines a text editor and software tools to provide a program development environment. You can create, edit, run, and debug your programs using a single interface, and this makes program development easier.

We will be working on IDLE, which is the built-in IDE for Python. IDLE is included with the Python standard distribution for Windows and macOS, so there is no need to install it separately. To get started with Python, IDLE is a good IDE. It has an interactive interpreter and features like smart indentation, auto-completion, and syntax highlighting, and also includes a

basic debugger. There are many other popular text-based editors and IDEs available that can work with Python. If you want, you can choose any of them to write your programs.

Eclipse is the IDE for development in Java. If you are familiar with Eclipse, you can install the PyDev plugin to develop Python programs. If you are comfortable working with Vim, you can use it for Python development by adding some plugins. PyCharm is the Python IDE for professional developers by JetBrains and comes in both free and paid editions. Sublime Text is a code editor that supports Python and many other languages. If you don't want to install Python on your computer, there are online platforms available that provide a web-based Python interpreter.

1.4 Python Interactive Mode

Whether you work on the command line or use IDLE, there are two ways in which you can use the Python interpreter - script mode and interactive mode.

In the script mode, we write our program statements in a file and then execute the contents of that file to get the output of the whole program. In the interactive mode, we type single Python statements on the prompt, and we get to see the output immediately. This interactive experimentation is particularly useful for beginners who have just started to learn the language. Even when we have learned the language well, we can use this mode to write short snippets of code and see how they work before putting them into a big program. As we have seen in the previous section, we can enter the interactive mode (Python shell) either through the command line or through IDLE.

The shell prompt or interactive prompt (`>>>`) denotes that you are in the Python interactive mode, so you can type just any valid Python statement or expression, hit Enter, and the result will be displayed immediately.

```
>>> 4 + 6
10
>>> print('Hello')
Hello
```

The interactive mode has active memory that remembers the previously executed statements on the prompt. However, this memory is active only for the current session. If you exit the interpreter and open it again, the code you typed in will not be available or remembered. So, if you want to retain and reuse your code, you should place it in a file and save it.

Although the interactive mode is not used for developing programs, it can serve as an excellent learning tool and can also be used to test code snippets. This book will often use this mode to explain different language features. You can also use this mode to play around with different Python constructs and functionalities and explore more about them.

Here is something different and interesting that you can try on the interactive prompt. Type `import this`, and you will get a short poem written by Tim Peters. This poem summarizes the style and philosophy of Python in the form of some guiding principles.

```
>>> import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
```

```
Explicit is better than implicit.
```

```
Simple is better than complex.
```

```
Complex is better than complicated.
```

```
Flat is better than nested.
```

```
Sparse is better than dense.
```

```
Readability counts.
```

```
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
```

```
Errors should never pass silently.
```

```
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
```

```
There should be one-- and preferably only one -- obvious way to do it.
```

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

1.5 Executing a Python Script

Interactive mode is good for experimenting and exploring, but when we have to write complete working programs that can be reused, we need to save our work in a file. In script mode, we create a file of Python statements and instruct the interpreter to execute the whole file, which is often called a script. The interpreter will execute all statements in the file sequentially, maintaining the order in which they appear.

There are two different ways of creating and executing scripts - we can write the script in a text editor and execute it on the command line, or we can create and execute the script in a development environment like IDLE. First, let us see how to execute a script from the command line.

Create a new text file using a text editor like Notepad++. Write the following two lines of code in the file and save it with the name `hello.py`; all Python files are conventionally saved with the `.py` extension.

```
print('Hello!')  
print(5 + 3)
```

You have written your first Python program. Now, let us see how to run it from the command line. On the command prompt, type **python** followed by the filename with full path and press Enter. Our program will be executed, and we can see the output.

```
C:\Users\deepali>python E:\Python_programs\hello.py
Hello!
8
```

Figure 1.8: Executing a Python program on the command line

If you do not want to type the whole path, you can first change your current directory to the directory in which you have your file by using the **cd** command. On recent versions of Windows and Python 3.3 onwards, you can write **py** instead of **python** or even write the name of the file to execute it.

Next, let us see how to run a Python script using IDLE, the built-in IDE of Python. When you open IDLE on your system, the Shell window appears. In the File Menu, click on New File, and a new window will open with Untitled written on its title bar. Save the file with the name **hi.py**. By default, your file will be saved in the Python installation folder where the Python code is stored. It is better to make a working folder for your programs in some other location and save your files in that folder. After saving the file, write the following code in the file:

```
print('Hi!')
print(5 - 2)
```

To run this program, either press **F5** or click on Run Module in the Run menu. The output of the program appears in the Python Shell window. Similarly, we can execute any existing Python program in IDLE; for example, we can open and execute our file **hello.py** that we had created using a text editor.

So now you know how to create a Python program and execute it. You can either use IDLE to write and run your programs, or you can write your program in a text editor and then run it on the command line. For beginners, using IDLE is recommended. If you are using a text editor, Notepad++ would be a better choice than Notepad. You should not use a word processor like MS Word, which uses formatted text. The text editor should store text in its pure form.

1.6 IDLE

To write programs effectively, you need to have a good understanding of the programming environment. Therefore, it is worth spending some time looking at the features of IDLE, the IDE that you will be using to write your programs. If you choose to use a different programming environment, make sure that you familiarize yourself with it before **starting to write programs**.

IDLE is the abbreviated form for "Integrated Development and Learning Environment"; Van Rossum probably named it after Eric Idle, who is a member of 'Monty Python's Flying Circus'. It is a very simple integrated development environment with features like syntax highlighting, automatic code indentation, auto completion, call tips, and a basic debugger. It is coded in Python using the TKinter GUI toolkit, and it is not platform-specific. It works mostly in the same way on Windows, Unix, and macOS. IDLE provides you with a simple graphical user interface (GUI) for performing your programming tasks, so it is easier to use than the command line.

As we have seen, there are two window types in IDLE - Shell Window and Code Editor Window. The Shell window provides a **Read-Eval-Print Loop (REPL)** environment for executing single statements. This window is interactive; it gives output for your command immediately. When you launch IDLE, the shell window opens up. Within this shell window, if you select "New File" or "Open" from the File menu, the code editor window will appear. In the code editor window, you can write and save a new program, or you can open an existing program. The editor of IDLE is multi-window, so you can have multiple code editor windows open at a time. In any of the open windows, the Windows menu presents a list of currently active windows, enabling you to switch between them.

When you run the program written in the code editor window, the Shell window automatically becomes active, and any output or error messages for the program will be displayed in this window. This means that if you have to do some editing in the editor window, then you have to activate it either by clicking on it or by switching to it through the Windows menu. If you want, you can arrange both the code editor window and the shell window side by side on your screen and then click on the one that you want to work in.

While the menu options in both windows are mostly similar, each one has some distinct options. We will briefly discuss some menu options; more detailed information can be found in the IDLE doc option of the Help menu of any of the two windows.

The **File** menu has its regular features like creating, opening, saving a file, closing the current window and exiting IDLE.

The **Edit** menu also has its typical options like Undo, Redo, Cut, Copy, Paste, Select All, Goto line, Find and Replace. You can use Ctrl+space to see a list of possible completions while typing a word. The Expand Word option can be used to expand a prefix to match a full word used in the same window. You can also use Tab for expanding words or for seeing a list of possible completions. This feature can be used to avoid typing long names, for instance, if you have defined the name `total_marks`, you can simply type "to" and press the tab key to quickly access and reuse the name in your current window. The Show Call Tip option is used while calling functions, and Show Surrounding Parentheses highlights the surrounding parenthesis.

In the **Shell** menu, Restart Shell will restart the shell and clean the environment. All the names that you have defined will be gone. The View Last Restart will scroll this shell window to the last Shell restart. To access command line history, you can use the options Previous History or Next History or press Alt+p and Alt+n, respectively. This way, you can scroll through previously entered commands.

The **Debug** menu is for debugging the program, which involves detecting and removing errors for your program.

In the **Options** menu, we have the configure IDLE option, which can be used to change the settings of IDLE. We can change the preferences for font, indentation, key shortcuts, startup windows and size, and text color theme. IDLE uses color coding for highlighting different types of text, for example, red for comments and orange for keywords. You can change the colors in the Settings box and save your selections as a theme. IDLE comes in with a built-in set of shortcut keys and you can define your own shortcut keys also. You can change the window that opens up when you launch Python. By default, the shell window is open when IDLE is launched; you can change this default to make the editor window open. You can create new help sources for IDLE. For example, you can provide a link to an online link or any other file on your computer. The help source that you provide will appear in the help menu. For most of the tasks, the default settings are pretty good, and most of the time, there is no need to change them.

The **Options** menu in the editor window has the Show Code Context option, which is useful in programs that have long functions or classes. If the name of the function or class has scrolled above the top of the screen, you can enable this option to see which function or class you are currently in.

The **Help** menu gives you access to the IDLE and Python documentation available on the official website. You can use this documentation even when you are not connected to the internet.

The code editor window includes a **Format** menu, which can be used to format a selection in different ways. The Indent Region and Dedent Region will shift the selected lines right or left. The default indent width is 4 spaces, and it can be changed. However, changing it is not recommended since this is the standard. The options Comment Out Region and Uncomment Region will comment or uncomment the selected text; we will learn about comments in the next chapter. The Tabify Region will turn leading stretches of spaces into tabs, and the Untabify Region will turn all tabs into spaces. Toggle Tabs is there to switch between indenting with spaces and tabs.

The code editor window also includes the **Run** menu, which can be used to run your code or check your code for syntax errors. You can also use it to open or activate the Python Shell window.

There are **context menus** available, which you can open by right-clicking in the window. In the Shell window, you have Cut, Copy, Paste, Go to file/line. In the Editor window, you have Cut, Copy, Paste, Set Breakpoint, and Clear Breakpoint. The last two options are used while debugging.

1.7 Getting Help

To get help on any Python feature, you can utilize the shell prompt by typing `help` followed by an opening parenthesis, a single quote, the keyword or topic you require help with, another single quote, and finally, a closing parenthesis. Here are some examples:

```
>>> help('print')
>>> help('keywords')
>>> help('for')
```

If you write `help('topics')`, the available topics will appear, and you can get help on any of those topics.

The other way to get help is to get into the help mode. To start the help mode, type `help` followed by empty parentheses and press Enter.

```
>>> help()
```

```
help>
```

In the help mode, we can see the help prompt in the window. Now, here, we can directly type the item on which we are seeking help.

```
help>print
```

```
help>keywords
```

To see what topics are available, you can type `topics` and press Enter. The help mode is suitable when you want to browse the help topics. To quit this help mode and return to the interactive interpreter, type `quit`, or you can just press Enter without typing anything.

Before you start writing programs, it is important to have a strong base in the fundamentals of Python. This chapter will introduce you to the basic concepts and building blocks that are used to construct a Python program. While many of the concepts presented in this chapter will be explored in more depth later in the book, it is important to familiarize yourself with certain terms right from the beginning. This chapter will provide a gentle introduction to these terms, offering you a solid foundation to build upon as we move to more comprehensive discussions in the following chapters.

In this chapter, you will learn how to name things in Python, what type your data can be, what operators you can use, how to input and output data from your program, how to structure your program, and many more things. Even if you have programmed in any other language before, the subject matter presented in the chapter will prove to be useful because you will find that Python operates differently in many aspects. If you dive into coding without having a solid foundation, you will always find yourself looking back at the basics. While you may manage to make your programs work to some extent, you will lack a comprehensive understanding of how they work and the underlying processes going on. A strong grasp of the fundamentals will serve as a solid framework for further exploration and growth in your Python programming journey.

2.1 Identifiers

As you start writing programs, you will create different program elements like variables, functions, classes, modules, instance objects, etc. To identify these elements in a program, you will have to give them some names. These

names are called identifiers, as they are used to identify program elements. There are some rules and conventions for naming identifiers. You have to follow the rules to prevent any errors and make your program work. Following the conventions increases the readability of your code and makes it easier to understand and maintain. Let us first see the rules for naming identifiers.

- The first character should be a letter or an underscore.
- The rest of the characters can be any combination of letters (A to Z, a to z), digits (0 to 9), and underscores. Special characters like @, %, \$, #, & are not allowed.
- There is no limit on the length of an identifier.
- Identifiers are case-sensitive. For example, marks, Marks and MARKS are considered different identifiers.

Here are some examples of valid and invalid identifiers:

Valid	p	part3	min_length
	Student		

Invalid	cost\$	min-length	3rd_part	cost price

cost\$ is invalid as it contains the illegal character dollar sign, min-length is invalid as it contains the illegal character dash, 3rd_part is invalid as it starts with a number, and cost price is invalid as it contains a space.

There are some special words that programmers cannot use for naming their program elements even though they satisfy all these rules. Here is the list of those words:

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while

```
async    except    lambda    with
await    finally  nonlocal          yield
break    for       not
```

You cannot use any of these words for naming your program elements. For example, you cannot have a variable named `import` or a function named `raise`. These names are reserved by the language for specific purposes; they are called *keywords* of the language. These keywords have predefined meanings in the language so you cannot use these names for naming your program elements. You can see the list of keywords in the interactive shell by using `help`.

```
>>> help('keywords')
```

These were the rules that need to be followed while naming identifiers. Now, let us see some conventions.

It is good to choose meaningful and descriptive names for identifiers. The name should indicate the purpose; for example, a variable name should describe the contents of the variable, a function name should indicate what the function does, and so on. This approach makes your code self-documenting and, therefore, easy to understand. For example, `shortest_path` and `spath` are both valid identifiers, but the former makes more sense. Similarly, `min_height` is better than `mheight`. However, there are some exceptions where single-letter or abbreviated names are fine. For example, names like `i`, `j`, `k` are generally used for loop indices. When names have to be used in big and complex expressions, longer names would make the code harder to read, so in these cases also we can think of shorter names.

We have seen that spaces are not allowed in identifiers, so when we need names with multiple words, we can use underscore as the word separator (eg. `marks_maths`, `calculate_tax`). For most of the names, all lowercase letters are used, but for class names, we generally use the CapWords convention, in which the initial letters of all the words are capitalized. As we proceed through the chapters and get introduced to different program elements, we will see the naming conventions for them.

There are some built-in names, like `all`, `any`, `print`, `sum`, `max`, etc., that you should not use as identifiers, although Python will not complain if you

use them. Using these names as your identifiers will overwrite the built-in names and may cause subtle problems in your program. To view the built-in names, you can type the following on the prompt:

```
>>> dir(__builtins__)
```

When you write your program, you will notice that the editor will highlight different terms in your program with different colors. For example, IDLE will highlight the keywords and built-in names in orange and purple colors, respectively. This feature of text editors is called syntax highlighting. They can recognize the category of a term and highlight it accordingly. So, in IDLE, if a word is highlighted in orange, it means that it is a keyword, and you cannot use it for your identifier name; if you try to do so, you will get an error. If a word is highlighted in purple, it is a built-in name, and it is better not to use it for your identifiers.

Another convention is to avoid names that start with single or double leading underscores. However, a single underscore on its own can be used as an identifier, and it has special meaning in the interactive mode.

2.2 Python Types

The programs that we write mainly store and process data, and data can be in different forms; it can be numeric, text, or a list. Data can be categorized, and each category is called data type or simply type. A data type represents a domain of values and a set of possible operations that can be performed on those values. For example, for integer data type, the domain of values contains all integers, and the set of possible operations are addition, subtraction, multiplication, etc. The data types that are predefined in Python are called built-in data types. Python has a variety of data types that you can use to represent your data. Here are some of them:

```
int    float    complex    str    bool    list    tuple    set  
dict
```

You can also define your own types by combining these types. We will see how to do that later when we learn how to define classes. In this chapter, we will look at some of Python's basic built-in data types. Before looking at Python types, let us see what the term **literal** means. A literal is a notation

for a constant value of some built-in type. A literal can be a number or some text that appears in a program; it is just a value. For example, these are some literals:

```
12      35.2      'hello'      True
```

12 is a literal of type `int`, 35.2 is a literal of type `float`, 'hello' is a literal of string type `str`, and `True` is a literal of boolean type `bool`. Now, let us see the types in detail. The first one that we will see is `int` type.

These are some examples of `int` literals:

```
34      123      1233      6532216
```

Integer values can be arbitrarily long. There is no limit on the size of integers in Python. In practice, they are limited by the size of your computer's memory. If we enter an `int` literal on the interactive prompt, it prints the literal back. We can also perform simple arithmetic operations on integers at the interactive prompt.

```
>>> 25
```

```
25
```

```
>>> 3 + 42
```

```
45
```

```
>>> 6 ** 200
```

```
426825223812027400796974891518773732342988745354489
429495479078935112929549619739
019072139340757097296812815466676129830954465240517
595242384015591919845376
```

In the last example, we are performing an exponentiation operation, which gives us the value of 6^{200} . We can have such big integers in Python; they can be of unlimited size. While performing arithmetic calculations on integers, we do not have to think about overflow.

By default, the integer literal values are expressed in decimal base (number system), but they can also be expressed in hexadecimal, octal, or binary base.

In Hexadecimal form (Base 16) Prefix 0x (or 0X)

In Octal form (Base 8) Prefix 0o (or 0O)

In Binary form (Base 2)

Prefix 0b (or 0B)

If you want to express an integer value in hexadecimal, then you have to prefix the value with zero and letter x; for octal, you have to prefix the value with zero and letter o; and for binary, you have to prefix the value with zero and letter b. Here are some examples of `int` literals in different bases:

```
0x1abc          0X1ABC          0o1776
0b11000011
```

The first 2 integers are expressed in hexadecimal, the third integer is in octal base, and the fourth one is in binary base. If you enter these numbers on the interactive prompt, it will print them in decimal form, as shown below:

```
>>> 0x1abc
6844
>>> 0X1ABC
6844
>>> 0o1776
1022
>>> 0b11000011
195
```

Floating-point values are numbers with a decimal point and an optional exponent represented by lowercase or uppercase E. Here are some examples of `float` type literals:

```
2.34      5.8      3e5      7.2e42      6.5E-24
```

When the letter e or E is used, the floating-point value is said to be in scientific notation. This letter separates the number from the exponent. You can easily represent very large or very small values using this notation. For example, the value `6.54e25` denotes 6.54×10^{25} , which is a very big number, and the value `5.32e-13` denotes 5.32×10^{-13} , which is a very small number.

In general mathematics, we use commas in large numbers for clarity; for example, we would write one million as 1,000,000. In Python, commas are not allowed inside numbers, but you can use underscores to separate the digits of numeric literal values so that you can write 1 million as

`1_000_000`. This feature was added in 3.6 to enhance the readability of numeric values. Here are some examples:

```
45_345_678      0x_234_CAB      0o_231_354
23_456.678_566
```

Python supports a Boolean type `bool`, which takes a value of either `True` or `False`. These are the only literal values for `bool` type. The first letter is capitalized in both `True` and `False`, and both of these are keywords. `bool` type is generally used in comparisons; we will see it in detail when we learn about operators.

The complex number type is mostly used in scientific applications. Complex numbers have a real part and an imaginary part. The imaginary part is denoted with a suffix of lowercase or uppercase `j`. Here are some literals of `complex` type:

```
3 + 5j      2 + 4j      3 + 6j
```

The string type `str` is the most commonly used type. We will explore strings in detail in the next chapter. We have already used strings inside the `print` function. A string is just a group of characters placed inside a pair of quotes. In Python, you can enclose a string literal within a pair of single quotes (`' . . . '`) or a pair of double quotes (`" . . . "`). You can even use triple quotes, but the most commonly used delimiters are single quotes. Here are some literals of type `str`:

```
'Bareilly'      '430164'      '$129'      'Enter your
name: '
```

Python has a special type `NoneType`, which can be used to represent *no value* or *nothing*. It has a single literal value `None`. There is only one `None` object, and all references to `None` refer to that same object. So, whenever you want to make any null object in your program, you can use `None`.

You can use the built-in `type` function to check the type of any value, as shown below:

```
>>> type(23)
<class 'int'>
>>> type(True)
```

```
<class 'bool'>
>>> type(2.3)
<class 'float'>
>>> type('hi')
<class 'str'>
>>> type("Hello")
<class 'str'>
>>> type(None)
<class 'NoneType'>
```

The collection types like lists, tuples, dictionaries, sets, and frozensets are covered in separate chapters in the book. Types are also known as classes in Python. Later, we will see how to define our own types by using the `class` keyword.

2.3 Objects

Everything in Python is implemented as an object. Any data value you write, like any number or a string, is an object. Program elements like functions, classes, and modules are also implemented as objects. An object is just a chunk of memory used to store some data. So, objects are Python's abstraction for data.

Whenever we write any literal value in our program, Python identifies its type because of its notation and creates an object of the appropriate type. If it sees a sequence of digits, it will create an `int` object; if it sees text inside quotes, it will create a `str` object. For example, if we write the literal `56` in our program, Python recognizes it as an integer literal and creates an object of type `int`. Similarly, for the string literal `'Hello'`, it creates an object of `str` type.



Figure 2.1: Objects of type int and str

Python uses objects to hold data values. Every object has a type, a value, and an identity. For the first object, 15031263572 is the identity or the id, 56 is the value, and the type is `int`. For the second object, 18043139781 is the id, 'Hello' is the value, and `str` is the type. The value of an object is the data that it contains, and the type of an object determines what kind of operations can be performed on the value. For example, we can slice `str` values but not `int` values, and we can divide two `int` values but not `str` values.

The identity(id) of an object is an integer that is guaranteed to be unique among simultaneously existing objects. Each object in our program will have a different id, which will never change during its lifetime. An object is stored in memory, and typically, the identity of an object is the memory address of that object, i.e., the location in the memory where the object is stored. We can use the built-in `id()` function to get the identity of an object in our program.

2.4 Variables and assignment statement

We have seen that Python uses objects to store values. If we want to work with a value later on in our program, we can associate a name with the object that contains the value. So, objects contain values, and to access these objects and manipulate them in our program, we can create names and bind them to objects. In the following example, the name `x` is bound to `int` object with value 56, and the name `p` is bound to the `str` object with value 'Hello'.

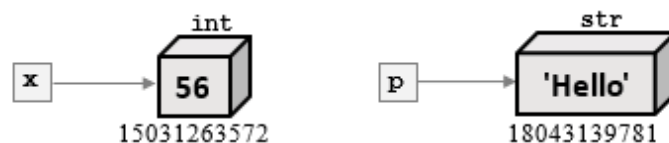


Figure 2.2: References to objects

Whenever we write the name `x` in our program, the value 56 will be used, and whenever we write `p`, the string 'Hello' will be used. The names `x` and `p` are variables. In Python, variables are just names that refer (or point)

to objects; the actual data is contained in the objects. So, objects are chunks of memory that store the actual values, and variables are names that link to objects by storing the memory address or location of the object. We can think of variables as object references - they are just names attached to objects.

Now, let us see how to create a variable in our program and bind it to an object. For that, we need to write an assignment statement:

```
>>> x = 56
```

When Python executes this statement, it creates an object of type `int` with value 56. It also creates a variable named `x` and will make that variable refer to this object, or we can say that it binds the name `x` to this object. After the execution of this statement, whenever `x` appears in an expression, it will be substituted with the value of the object that is bound to the name `x`. The value 56 is contained within the object, and we can refer to the object by using the name `x`. At the prompt, typing just the name of the variable will display its value. In the program file, we have to use the `print` function to print the value of the variable on the output screen:

```
>>> x
56
>>> print(x)
56
```

Since `x` refers to an `int` object, we can perform all operations on `x` that make sense for `int` type:

```
>>> x + 5
61
```

If we send a variable to the `type()` or `id()` function, we will get the type or id of the object that the variable is currently referring to:

```
>>> type(x)
<class 'int'>
>>> id(x)
15031263572
```

The following statement will create an object of type `str` with the value `'hello'`, and it will create and bind the variable named `p` to this object:

```
>>> p = 'Hello'
```

Now, whenever we write `p`, it will give us the value of the object bound to the name `p`:

```
>>> p
'Hello'
>>> print(p)
Hello
```

Variables `x` and `p` will be available in the interactive session until we exit it.

The following assignment statement will create a new variable named `z`:

```
>>> z = x
```

The name `z` will also refer to the same object to which `x` is referring:

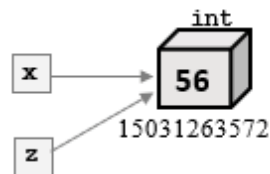


Figure 2.3: Variables `x` and `z` refer to the same object

Now, both `x` and `z` refer to the same object, so now, we can access this object by any of these two names. We have created an alias for `x`.

The following assignment statement creates one more variable named `y`, and it is bound to the object to which the name `z` is bound.

```
>>> y = z
```

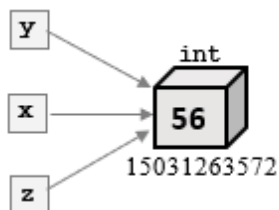


Figure 2.4: Variables *x*, *y*, and *z* refer to the same object

Now, all three variables, *x*, *y*, and *z* are bound to the same object, and any of them can be used to access the underlying object. This is known as object sharing or aliasing.

```
>>> x
56
>>> y
56
>>> z
56
```

When we apply the `id` function to a variable name, we get the identity of the object that the variable is referring to. The following output proves that all three variables, *x*, *y*, and *z*, refer to the same object.

```
>>> id(x)
15031263572
>>> id(y)
15031263572
>>> id(z)
15031263572
```

Any variable can be made to refer to another value; that is why it is called a variable. Let us see what happens when we try to change *x* by writing the following assignment statement:

```
>>> x = 25
```

A new integer object with value 25 will be created, and the variable *x* will refer to this newly created object:

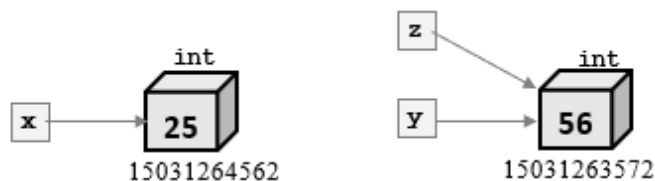


Figure 2.5: Variable *x* refers to a new object

The first time, when `x` appeared on the left-hand side of the assignment statement, the variable name `x` was created and was bound to an object. The second time, when `x` appeared on the left-hand side of the assignment statement, the name `x` already existed, so this time, it was rebound to an object. Now the value of `x` is 25, and its id has also changed.

```
>>> x
25
>>> id(x)
15031264562
```

Can you guess what happens when we write the following assignment statement?

```
>>> z = z + 3
```

First, the expression on the right-hand side is evaluated, the value of `z` is 56, 56+3 is 59, and the value on the right-hand side is 59, so a new `int` object with value 59 is created, and `z` is now bound to this new object:

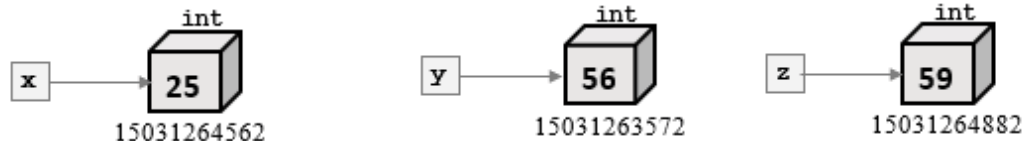


Figure 2.6: Variable `z` refers to a new object

The statement `z = z + 3` does not in any way change the object that `z` was referring to originally; instead, it rebounds `z` to another object. The variable `y` is still referring to the object with value 56. Now, we write the following statement:

```
>>> y = 3.6
```

A new `float` object is created, and `y` is now rebound to this object:



Figure 2.7: Variable `y` refers to a new object; object 56 is orphaned

Variable names have no types associated with them. They are just names, and they can refer to any type of object. The variable `y` was initially referring to an `int`; now it is referring to a `float`. You can make it refer to any other type of object later on. This is why Python is called a dynamically typed language. A variable in Python can be associated with any type of object, and it can later be rebound to any other type of object. To see the type of the object that a variable is currently referring to, we can use the `type` function:

```
>>> type(y)
<class 'float'>
```

We can see that the object `56` has been orphaned; there is no variable name referring to it. Python will notice this, and its garbage collector will automatically remove it from the memory. The memory that was occupied by this object can be used for some other purpose.

In our examples, we have taken single-lettered variables. In real applications, variables with more meaningful names will be used. While naming variables, we need to follow the same general rules that we had seen in naming identifiers. The convention for naming variables is to use lowercase letters with underscores separating words.

Here are some examples of variable names:

```
area      marks_in_english    total_marks
simple_interest
```

If you have programmed in C, C++, or Java, you might be wondering how we can use a variable without declaring it in advance. These languages are statically typed; they require you to declare a variable along with its type information before it can be used in the program, and once you declare a variable, you can never change its type. For example, if you declare a variable of type `float`, it will be a `float` for the whole duration of the program, and you can store only `float` values in it. So, in these languages, variables have predetermined types, and a variable can be used to hold values of only that type. Python is a dynamically typed language, so there is no need to declare the type of a variable. Variables do not have fixed types in Python; they are generic in nature. Instead, objects have types. A variable is just a name, and it can refer to any type of object.

In statically typed languages, a variable is considered as a storage box that can store a value of a specific type. The variable names represent fixed places in memory, and we need to declare the type because the amount of space reserved depends upon the type. In Python, a variable is visualized as a kind of label or tag that can be attached to an object of any type. There is no need to predeclare variables in Python as they automatically come into existence when they are initially assigned (assigned first time), and there is no need to specify a type because variables do not have any type associated with them. The initial assignment introduces the name of the variable in the program and binds it to an object, and all other future assignments to the variable rebind it.

2.5 Multiple and Pairwise Assignments

We have seen that a variable can be created or rebound using a simple assignment statement. More than one variable can be created or rebound in a single assignment statement by using multiple and pairwise assignments.

We can assign multiple variables simultaneously with a common value. For example, in the following assignment, variables `a`, `b`, and `c` are assigned in a single line, and all of them refer to the same object.

```
>>> a = b = c = 10
```

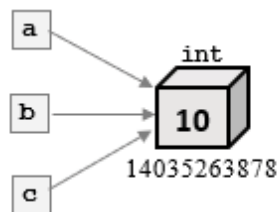


Figure 2.8: Multiple assignment makes variables refer to the same object

If any of these variable names do not exist before this assignment statement, they will be created. Variables that already exist will be reassigned.

Pairwise assignment can be done by using commas:

```
>>> x, y, z = 1, 2.5, 3
```

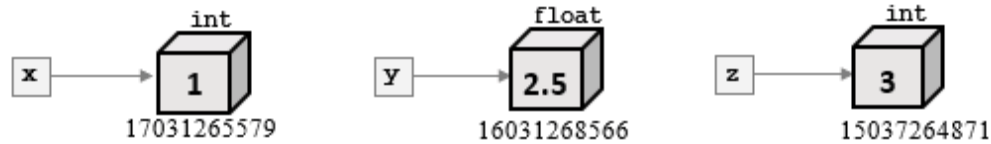


Figure 2.9: Pairwise assignment

The values 1, 2.5, and 3 are assigned to variables `x`, `y`, and `z` respectively. As usual, if this assignment is the first for any of these variables, then it will be created, and if the variable name already exists, then it will be reassigned. For pairwise assignment, the number of variables on the left side should be equal to the number of values on the right side.

2.6 Deleting a name

The `del` statement can be used to delete a variable name. It consists of the `del` keyword followed by the name that has to be deleted.

```
del name
```

Suppose we have three variables, `x`, `y`, and `z`, referring to the same object.

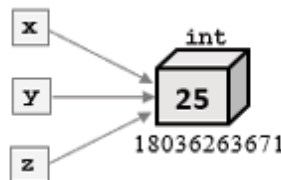


Figure 2.10: Variables `x`, `y`, and `z` refer to the same object

To delete the variable name `x`, we can write the following statement:

```
>>> del x
```

This statement will unbind the name `x` from the object and will also delete the name `x`. It will not delete the object referred to by `x`, which means that it will not free the memory occupied by the object. An object will be automatically garbage collected by Python only if there is no other name referring to it; you can never explicitly destroy an object.

While the program is running, objects are automatically created and reclaimed automatically when they become unreachable. This automatic

reclamation of the space occupied by an unreachable object is called garbage collection. It is done to free up space so that it can be used for other objects that may be created later on in the program. This memory management is automatically done by Python; programmers do not have to bother about freeing up space that is no longer in use. There is no need to manage memory manually by writing allocation and deallocation code that is required in other languages like C and C++. In these languages, the programmer is responsible for allocating and deallocating memory. This is error-prone and can cause memory leaks if not done properly. Automatic garbage collection in Python reduces efforts and minimizes the chances of memory management problems.

The process of garbage collection depends on the Python implementation; typically, a reference counting mechanism is used. In each object, a reference counter is stored, which keeps track of the number of references referring to the object. When this number drops to 0, the object is automatically reclaimed, which means that the memory allocated for the object is freed. Programmers do not need to worry about how the garbage collector works; the whole process is hidden and automatic.

Continuing our example, suppose we delete the name `y` and reassign the name `z`:

```
>>> del y
>>> z = 10
```

Now, there is no variable name referring to object 25, so it will be garbage collected.

We can use the `del` statement to delete more than one name by using commas.

```
>>> del a, b, c
```

The `del` statement is used very rarely; variable names have a lifetime, and they are deleted automatically when their lifetime is over. We will discuss this later in the book.

2.7 Naming convention for constants

In some languages, we can define names which cannot be reassigned. Once they are given a value, they cannot be changed throughout the execution of the program. They are called constants. In Python, there is no concept of constants; there is no way to define names that cannot be reset to a different value. All names in Python can be reassigned at any time. However, there is a widely used naming convention to indicate that you do not want a name to be reassigned. The convention is to use all capital letters with underscores separating words. Here are some examples:

```
PI = 3.14159
MAXIMUM_SIZE = 100
RATE_OF_INTEREST = 5
```

Use all lowercase letters in the names of variables whose values might change, and use all uppercase letters for names that should never be reassigned values. But remember that this is just a convention and not a restriction; these names with all uppercase letters can be reassigned, and the interpreter will not complain. By using all caps, you are not instructing the interpreter that it is a constant; you are telling the programmer that it should be treated as a constant and should not be changed.

So, why do we need such names that do not change? We could just use the literal value `3.14159` instead of the name `PI` or `100` instead of `MAXIMUM_SIZE`. The reason is that they can help in documenting the program. When you need to use these literals in many places, it is better to give them a name. The number `100` does not give any real information, while the name `MAXIMUM_SIZE` is clear and understandable. Using these descriptive labels is better than literal numbers scattered throughout your program.

Another reason is that they are good for code maintenance. Suppose that after some time, you decide to change the maximum size from `100` to `150`; then, you will need to change it at only one place where you have defined this name. If you use the number `100` instead of this name, then you will have to find every single place where this number `100` is used as the maximum size and change it to `150`, which is time-consuming and definitely error-prone.

2.8 Operators

An operator is a symbol or a word that specifies an operation to be performed. Here are some examples of operators in Python:

+ ** is // >> == and <=

An operator works on operands and yields a value. An operand is a data item on which an operator acts; it can be a literal value or a variable. Here are some examples of operands:

24 5.8 marks x

Python includes a large number of operators that fall under several different categories, depending on the type of task that they perform.

Arithmetic Operators	+	-	*	/	//	%	**
Relational Operators	==	!=	<	>	<=	>=	
Logical Operators	and	or	not				
Bitwise Operators	&		^	~	<<	>>	
Membership Operators	in	not in					
Identity Operators	is	is not					

Figure 2.11: Operators

If an operator operates only on one operand, it is a unary operator, and if it operates on two operands, it is a binary operator. For example, the negation operator (-) is unary, while the addition (+) and less than (<) operators are binary. Most of the operators are binary.

2.8.1 Arithmetic operators

Arithmetic operators perform arithmetic operations like addition or subtraction, and relational operators perform comparisons. Most of the operators do what you would expect them to do, but some of them require explanation. So, we will briefly discuss these categories one by one. First, let us discuss arithmetic operators:

Operator	Name	What it does	Example
-	Negation	Negates its operand	-5
+	Addition	Adds its operands	5 + 3 ⇒ 8
-	Subtraction	Subtracts right operand from left operand	5 - 3 ⇒ 2
*	Multiplication	Multiplies its operands	5 * 3 ⇒ 15
/	True Division	Divides left operand by right operand, returns a float	5 / 3 ⇒ 1.66 -5 / 3 ⇒ -1.66
//	Floor division	Divides left operand by right operand, and returns an integer which is the floor value of the result	5 // 3 ⇒ 1 -5 // 3 ⇒ -2
%	Modulo	Returns the remainder when left operand is divided by right operand	5 % 3 ⇒ 2
**	Exponentiation	Returns left operand raised to the power of right operand	5 ** 3 ⇒ 125

Figure 2.12: Arithmetic Operators

The - sign is used for both the negation operator and the subtraction operator. To specify a number as negative, we put the negation operator in front of it. The addition operator + adds its operands, and the * operator multiplies its operands. There are two division operators: true division (/) and floor division (//) operator. Both these operators divide the left operand by the right operand; the true division operator returns the result as a float value, while the floor division returns an integer, which is the floor value of the result. The floor value is calculated by rounding off to minus infinity (rounding down); for example, the value of 15//2 is 7, and the value of -15//2 is -8.

The modulo operator (%) returns the remainder when the left operand is divided by the right operand. The result has the same sign as its second operand. This operator can be used to check whether a number is divisible by another number. For example, if $x \% y$ is zero, it means that x is divisible by y . It can also be used to extract digits from the right of a number; for example, if x is an integer, $x \% 10$ will give the rightmost digit, $x \% 100$ will give the last two digits from the right, and $x \% 1000$ will give last 3 digits.

The operator with two asterisks (**) is the exponentiation operator; float values can be used both in the base and the exponent.

For addition, subtraction, multiplication, modulo, floor division, and exponentiation operators, if both operands are `int`, the result will be an

`int`. If one of the operands is a `float`, then the result will be a `float`. For the true division operator (`/`), the result will always be a `float`. The following table will help you understand the difference between the true division operator and the floor division operator:

True Division operator /		
Type of operands	Type of result	Example
int and int	float	$9/5 \Rightarrow 1.8$
int and float	float	$9.0/5 \Rightarrow 1.8$
float and float	float	$9.0/5.0 \Rightarrow 1.8$

Floor division operator //		
Type of operands	Type of result	Example
int and int	int	$9//5 \Rightarrow 1$
int and float	float	$9.0//5 \Rightarrow 1.0$
float and float	float	$9.0//5.0 \Rightarrow 1.0$

Figure 2.13: Division operators

We can see the result of an operation by typing it in the interactive terminal. Adding space around operators makes the operations more readable in the code:

```
>>> 1.2 + 4
5.2
>>> 3 ** 2
9
>>> 16 ** 0.5
4.0
>>> 17 / 5
3.4
>>> 17 // 5
3
>>> 17 % 5
2
```

When a variable is used with an operator, its value is used, and then the operation is performed.


```

>>> x = 4
>>> y = 5
>>> x + 5
9
>>> x ** y
1024
>>> x // y
0

```

2.8.2 Relational operators

Relational operators, also called comparison operators, compare their operands and return either True or False.

Operator	Name	What it does	Examples
==	equal to	Returns True if its operands are equal	5 == 5 ⇒ True 4 == 5 ⇒ False
!=	not equal to	Returns True if its operands are not equal	4 != 5 ⇒ True 4 == 5 ⇒ False
<	less than	Returns True if left operand is less than right operand	4 < 5 ⇒ True 5 < 4 ⇒ False
>	greater than	Returns True if left operand is greater than right operand	5 > 4 ⇒ True 4 > 5 ⇒ False
<=	less than or equal to	Returns True if left operand is less than or equal to right operand	4 <= 5 ⇒ True 5 <= 4 ⇒ False
>=	greater than or equal to	Returns True if left operand is greater than or equal to right operand	5 >= 4 ⇒ True 4 >= 5 ⇒ False

Figure 2.14: Relational operators

The operator == returns True if its operands are equal. This operator has two equal signs; when only one equal sign is used, an assignment is performed, as we have seen in previous sections. A common beginner's mistake is to use = instead of == for comparison since, in school maths, we use = for equality. In Python, whenever you need to perform a comparison, use two equal signs, and when you want an assignment, use one equal sign.

The operator `!=` returns `True` if its operands are not equal. We also have less than, greater than, less than or equal to, and greater than or equal to operators. All the relational operators have the expected meaning for numeric types `int` and `float`, and for strings, they are defined lexicographically and case-sensitively.

```
>>> x = 3
>>> y = 4
>>> x < y
True
>>> x == y
False
>>> x != y
True
>>> x >= y
False
```

2.8.3 Logical operators

There are three logical operators that can be used to combine Boolean values. These operators can be applied to operands that have values of `True` or `False` or to operands that can be converted to these values.

The result of `and` operator will be `True` only when both its operands are `True`; otherwise, it will be `False`. The result of `or` operator will be `False` only when both its operands are `False`; otherwise, it will be `True`. The `not` operator will negate the value of its operand. If the operand is `True`, the result will be `False`, and if the operand is `False`, the result will be `True`.

Operator	Name	What it does	Truth table
and	Logical and	Determines whether both its operands are True	True and True $\Rightarrow$ True True and False $\Rightarrow$ False False and True $\Rightarrow$ False False and False $\Rightarrow$ False
or	Logical or	Determines whether one of its operands is True	True or True $\Rightarrow$ True True or False $\Rightarrow$ True False or True $\Rightarrow$ True False or False $\Rightarrow$ False
not	Negation	Negates the value of its operand	not True $\Rightarrow$ False not False $\Rightarrow$ True

Figure 2.15: Logical operators

Since relational operators return Boolean values, we can use relational operations (like `a < b`) as operands of logical operators. This way, we can make multiple comparisons by combining different conditions.

```
>>> x = 3
>>> y = 4
>>> x > 0 and x < 6
True
>>> x == 3 and y < 6
True
>>> x > 10 and y < 6
False
>>> x > 10 or y < 6
True
```

Python allows chaining of comparison operators. So, we can write expressions like these:

```
>>> 1 < x < 8
True
```

The expression `1 < x < 8` will be True if `1 < x` is True, and `x < 8` is also True. The expression implies logical AND; it is equivalent to writing `1 < x and x < 8`. The chained form is more readable as it evaluates the subexpression only once.

2.8.4 Identity operators

Sometimes, we may want to know whether two variables refer to the same object. Instead of applying the `id` function on both variables and then comparing the results, we can use the two identity operators represented by the words `is` and `is not`.

Operator	What it does
<code>is</code>	Returns True if identity of left and right operands is same
<code>is not</code>	Returns True if identity of left and right operands is not same

Figure 2.16: Identity operators

Suppose we have three variables: `x`, `y`, and `z`. The variables `x` and `y` refer to the same object, while the variable `z` refers to a different object.

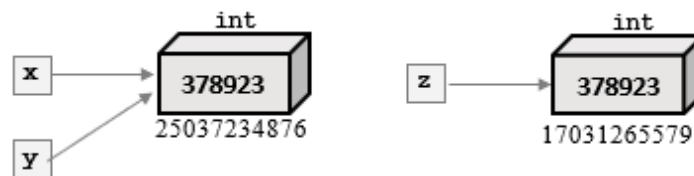


Figure 2.17: `x` and `y` refer to the same object, and `z` refers to a different object with the same value

`x is y` will return True because `x` and `y` both have the same identity and refer to the same object. `x is z` will return False because `x` and `z` have different identities and refer to different objects, although their values are the same. It is important to understand the difference between equality and identity. The relational operators `==` and `!=` test for equality, and the operators `is` and `is not` test for identity. The equality operator will return True for both `x==y` and `x==z` as it only tests for equality of values. Here are some examples on the prompt:

```
>>> a = 123456789
>>> b = 123456789
>>> c = a
>>> a is b
```

```
False
>>> a is c
True
>>> id(a)
2293201428272
>>> id(b)
2293201428240
>>> id(c)
2293201428272
>>> a is not b
True
>>> a == b
True
```

The `is` operator is commonly used to compare a variable with `None`, which is the null object of Python.

```
>>> a = None
>>> a is None
True
```

Here are a few more examples:

```
>>> c = 2
>>> d = 2
>>> c is d
True
>>> e = 1.5
>>> f = 1.5
>>> e is f
False
>>> g = 'cat'
>>> h = 'cat'
>>> g is h
```

True

We get different results here because, for small strings and small integers, Python performs optimization and maintains a cache; it does not create a new object. For big integer literals and floats, it will make separate objects.

2.8.5 Membership operators

There are two membership operators named `in` and `not in`.

Operator	What it does
<code>in</code>	Returns True if the left operand is present in the right operand
<code>not in</code>	Returns True if the left operand is not present inside the right operand

Figure 2.18: Membership operators

These operators look for the left operand in the collection represented by the right operand and return True or False accordingly. We will see their use when we learn about collection types in Python. There is also a ternary operator in Python, which we will discuss later.

2.8.6 Bitwise operators

Bitwise operators operate on individual bits in the binary representations of their integer operands.

Operator	Name	What it does
<code>&</code>	Bitwise and	Resulting bit is 1 only when the bits in both operands are 1, otherwise it is 0
<code> </code>	Bitwise or	Resulting bit is 0 only when the bits in both operands are 0, otherwise it is 1
<code>^</code>	Exclusive or	Resulting bit is 1, if bits of both operands have different value, otherwise it is 0
<code>~</code>	One's complement	Unary operator that flips the bits of its operands
<code><<</code>	Left shift	Bits in left operand are shifted left by the number denoted by right operand
<code>>></code>	Right shift	Bits in left operand are shifted right by the number denoted by right operand

Figure 2.19: Bitwise operators

We will not discuss these operators in detail; at this point, it is just sufficient to know that these low-level operations are supported in Python.

2.9 Augmented assignment statements

It is common to perform some mathematical binary operation on a variable and then assign the result back to the variable. Here are some examples:

```
count = count + 1
salary = salary - 1000
marks_in_maths = marks_in_maths + grace_marks
price_pencil = price_pencil // 2
```

In the first statement, 1 is added to the variable `count`, and then the new value is assigned back to the variable `count`. Similarly, in all the other statements, we are performing some operations on the variable and assigning the result back to the variable. Python supports augmented assignment statements, which provide a shortcut for these types of expressions.

```
count += 1
salary -= 1000
marks_in_maths += grace_marks
price_pencil //= 2
```

Augmented assignment statement	Equivalent to
<code>x += 5</code>	<code>x = x + 5</code>
<code>x -= 5</code>	<code>x = x - 5</code>
<code>x *= 5</code>	<code>x = x * 5</code>
<code>x /= 5</code>	<code>x = x / 5</code>
<code>x //= 5</code>	<code>x = x // 5</code>
<code>x %= 5</code>	<code>x = x % 5</code>
<code>x **= 5</code>	<code>x = x ** 5</code>

Figure 2.20: Augmented assignment statements

Augmented assignment syntax is available for all binary arithmetic operations.

2.10 Expressions

An expression is a combination of variables, literals, and operators, and it always evaluates to a single value, which is again represented by an object. Here are some examples of expressions:

45 + 6	20.56 - 3 * 6	marks +
50	2 + 4 * 3	
(y+1) * (x-3)	a <= b	35
marks		

A single literal or a variable by itself is also considered an expression that evaluates to itself; for example, the integer literal 35 is an expression, and the variable `marks` is also an expression. Parentheses can be used in expressions for enclosing some operations. We have already seen that if we type an expression on the interactive prompt, the result of the expression is displayed. In the program, simply writing the expression will not do anything. We have to use the value of the expression in some way.

Evaluation of an expression generally results in the creation of a new object so that it can be used on the right side of an assignment statement.

```
z = x + y * 3
```

Here, first, the expression `x + y * 3` will be evaluated, and a new object will be created for the result. This object will be assigned to the variable `z`. So, if you want to preserve the value produced by an expression, you can assign it to a variable. Otherwise, the value will just vanish.

2.11 Order of operations: Operator Precedence and Associativity

When there is only one operator in an expression, it is evaluated without any ambiguity. For example, there is no confusion in evaluating the expression `45 + 6`. However, when more than one operator appears in an expression, then you need to determine which operator will be evaluated first. For example, consider the expression `2 + 4 * 3`. There are two ways in which this expression can be evaluated. If an addition is done first, then the value of this expression will be `6 * 3`, which is 18, and if multiplication is done first, then the value will be `2 + 12`, which is 14. According to

mathematics rules, multiplication would be done first, and 14 would be the correct value.

In Python also, there are some rules that are followed while evaluating expressions with multiple operators. Let us see those rules. The order of evaluation depends on the precedence of an operator. The following table shows the operator precedence for some common operators in Python, from lowest to highest precedence.

	Operator	Description	Associativity
Highest Precedence	**	Exponentiation	Right to Left
	+x, -x, ~x	Positive, negative, bitwise NOT	Left to Right
	*, /, //, %	Multiplication, division, floor division, remainder	Left to Right
	+, -	Addition and subtraction	Left to Right
	<<, >>	Left shift and right shift	Left to Right
	&	Bitwise AND	Left to Right
	^	Bitwise XOR	Left to Right
		Bitwise OR	Left to Right
	in, not in, is, is not, <, <=, >, >=, !=, ==	Membership tests, Identity tests and Comparisons	Left to Right
	not	Boolean NOT	Left to Right
Lowest Precedence	and	Boolean AND	Left to Right
	or	Boolean OR	Left to Right

Figure 2.21: Operator Precedence and Associativity

Operators in the same box have the same precedence. For example, the operators *, /, //, % have the same precedence. To get the complete table on your interactive prompt, you can type the following:

```
>>> help('PRECEDENCE')
```

Whenever an expression contains more than one operator, the operator with a higher precedence is evaluated first. For example, in the expression $2 + 4 * 3$, multiplication will be performed before addition because multiplication has higher precedence than addition. In the expression $x + y < 10$, firstly, the addition will be performed and then comparison because the addition operator(+) has a precedence higher than that of the less than(<) operator.

In the expression $36 / 2 * 3$, division and multiplication are in the same group, so they have the same precedence. If division is performed first, then the value will be 54, and if multiplication is performed first, then the value will be 6. In the expression $19 - 12 - 4 - 2$, we have three subtraction operators, which obviously have the same precedence. If we evaluate from left to right, then the value is $19 - 12 = 7$, $7 - 4 = 3$, and then $3 - 2 = 1$. If we evaluate from right to left, we have $4 - 2 = 2$, $12 - 2 = 10$, and then $19 - 10 = 9$. So, for expressions that have operators with the same precedence, the evaluation order is still a problem. To solve these types of problems, an associativity is assigned to each group. Associativity defines the order of evaluation for operators that have the same precedence.

In the precedence table, we can see that all the operators associate from left to right except for the exponentiation operator, for which the precedence is right to left. So, in the expression $36 / 2 * 3$, the interpreter will first perform division and then multiplication. The expression $19 - 12 - 4 - 2$ will also be evaluated from left to right, and the answer will be 1.

In the expression $2 ** 3 ** 2$, we have the exponentiation operator, which associates from right to left, therefore, firstly, $3 ** 2$ will be evaluated, which is 9, and then $2 ** 9$, which is 512.

So, these were the precedence and associativity rules in Python. If you want to override these rules and change the default evaluation order, you can use parentheses. The operations that are enclosed within parentheses are performed first. For example, in the expression $2 + 4 * 3$, if you want to perform addition first, you can enclose it inside parentheses. The value of the expression $(2 + 4) * 3$ is 18 because addition is performed before multiplication.

For evaluation of the expression inside parentheses, the same precedence and associativity rules apply. For example, in the expression $39 / (5 + 2 * 4)$, inside the parentheses, multiplication will be performed before addition.

You can use nested parentheses in expressions, which means a pair of parentheses can be enclosed within another pair of parentheses. In these cases, expressions within the innermost parentheses are always evaluated first, and then next to innermost parentheses, and so on, till the outermost parentheses. After evaluating all expressions within parentheses, the

remaining expression is evaluated as usual. For example, in the expression `5 * ((10 - 2) / 4)`, `10 - 2` is evaluated first, then `8 / 4`, and then `5 * 2`.

You can use appropriate spacing to show the evaluation order explicitly. PEP8 suggests adding whitespace around operators with the lowest priority. In the following expressions, the order of operations performed is clearer due to spacing.

```
x + y**2 - a/b
a+b < c+d
```

This approach makes the code more readable. Anyone reading the code does not need to refer to the precedence table to figure out which operation will be performed first.

2.12 Type Conversion

You can combine different types of values in an expression. For example, `2 * 3.5` is a mixed-type expression. The two operands are of different types: `int` and `float`. Similarly, `1.5 < x < 8` and `9 + '5'` are also mixed type expressions. Before the evaluation of such expressions, the operands have to be converted to a common type. There can be other situations also where you will want to convert from one type to another. For example, you might have some numeric data in string form, and you want to convert it to `int` or `float` so that arithmetic calculations can be performed on that data.

The process of converting a value of one type to another type is called type conversion. There are two kinds of type conversions in Python:

- Implicit type conversion (Coercion)
- Explicit type conversion (Casting)

Implicit type conversion is done automatically by the interpreter when evaluating expressions of mixed types. For example, in the expression `2 * 3.5`, the interpreter will convert integer `2` to the floating point equivalent `2.0`, and then both `float` operands will be multiplied, and the result will be a `float`. The interpreter always promotes the smaller type to the larger type to avoid any loss of data. It then performs the operation in larger type

and returns the result in larger type. The type `int` is considered "smaller" than `float`, and `float` is considered "smaller" than `complex`. The implicit conversion is done only in related types; it is not performed between unrelated types like, for example, `int` and `str`.

If we try to add a string and an `int`, for example, `'2' + 5`, Python will not perform any conversion automatically. In this case, the programmer has to request a conversion explicitly. Explicit type conversion is performed by writing the required type name followed by the value to be converted inside parentheses. For example, `int('2')` will convert the `str` value `'2'` to `int` value `2`, and `float(28)` will convert the `int` value `28` to `float` value `28.0`. Here, `int()` and `float()` are type conversion functions. They will try to convert a value to their respective types. For example, the `int()` function will take any value and try to convert it to an integer, if possible.

```
>>> int(12.3)
```

```
12
```

```
>>> int('100')
```

```
100
```

```
>>> int(True)
```

```
1
```

```
>>> int(False)
```

```
0
```

```
>>> int('two')
```

```
ValueError: invalid literal for int() with base 10:  
'two'
```

When we convert a `float` to an `int`, the fractional part is truncated. When Boolean values `True` and `False` are converted to `int`, we get `1` and `0` because `True` is equivalent to integer `1` and `False` is equivalent to integer `0`. When we tried to convert the string value `'two'` to an `int`, we got an error because the `int()` function cannot convert a string to an integer if the string does not represent a valid integer value.

The `int()` function can convert a string to an integer if the string represents a number in hexadecimal or binary base. In this case, we have to

inform the `int()` function about the base. In the following examples, we are converting strings that contain hexadecimal and binary values to integer values, which are displayed in a decimal base.

```
>>> int('FF', 16)
255
>>> int('1010', 2)
10
```

We can use the `str()` function to convert a value to `str` type and `float()` function to convert a value to `float` type.

```
>>> str(100)
'100'
>>> str(3.6)
'3.6'
>>> float('3.45')
3.45
>>> float(3)
3.0
```

If the string that you send to the `float()` function is something that cannot be converted to a valid `float`, then Python will raise an error.

We know that the type of an object cannot be changed, so whenever there is a type conversion, whether implicit or explicit, Python creates a new object for the converted value.

2.13 Statements

A program is a sequence of statements, and a statement is an instruction that the Python interpreter can execute. Statements can be simple or compound. Statements like `a = 5`, `x *= 10`, `y = a + b` are simple statements. Compound statements (e.g. `if`, `while`, `for`) are a group of statements that are treated as a single statement. They generally consist of a header line ending in a colon and an indented block that contains other statements. We will learn about compound statements in the coming chapters.

Simple statements in Python generally end with a newline. Unlike other languages like C++ or Java, there is no need to place a semicolon (;) to end a statement. In Python, the end of the line means the end of the statement. So, Python uses newline as the statement terminator. However, there are two exceptions to this rule. If there is a backslash at the end of the line, then the statement continues on the next line. For example, the following statement continues on the next line because of the backslash character:

```
total_marks = marks_science + marks_maths \
              + marks_english + marks_socials \
              + grace_marks
```

So, if you have to write a statement that is too long to fit on a single line, you can spread it on multiple lines by using backslash (\) as the continuation character. This character at the end of the line indicates that the next line is a continuation. This way, you can join multiple adjacent lines to construct a single statement. This is called explicit line joining or explicit continuation.

Another situation when a statement does not end with a newline is when an opening delimiter like parentheses, square brackets, or curly braces has not been closed yet. In this case, Python automatically continues the statement on the next adjacent line. This is called implicit line joining or implicit continuation.

```
months = [
    'January', 'February', 'March', 'April',
    'May', 'June', 'July', 'August',
    'September', 'October', 'November',
    'December'
]
if (is_leap==TRUE and month=='MARCH'
    and weekday=='SUNDAY'):
student = {
    'name': 'John',
    'gender': 'M',
    'city': 'Paris',
```

```
    'age': 21,  
}
```

Thus, any expression that is inside parentheses (), square brackets [], or curly braces {} can be split over more than one line without using backslashes. An exception to this is when there is an unterminated string literal enclosed in single or double quotes.

```
print('Age should be less than 80  
      and greater than 18')
```

Here, the implicit line joining will not work.

You can take advantage of the implicit continuation to write more readable code. Instead of inserting backslashes to continue the statement, it is recommended to enclose your expression in parentheses to increase readability.

```
total_marks = ( marks_science + marks_maths  
                + marks_english + marks_socials  
                + grace_marks )
```

You can place multiple statements on a single line by separating the statements with a semicolon. For example, the following line of code consists of actually four statements.

```
a = 10; x = 5; y = a + x; z = a - y;
```

However, this style is not recommended. Writing a single statement on each line is preferred as it makes the code more readable and easier to understand.

2.14 Printing Output

Most computer programs interact with the user; they take some input from the user at run time and display some sort of output on the screen. In Python, we use the `input` function to get input from the console and `print` function to display the output on the console. In this section, we will discuss the `print` function, and in the next section, we will discuss the `input` function.

We have already seen how to display information on the screen by using the `print` function. We have used it to print a literal, value of a variable or any expression. Write the following statements in a `.py` file, execute it, and observe the output.

```
print('Let us start programming')
print(5 + 3*6)
name = 'Devank'
age = 10
print(name)
print(age)
```

Output-

```
Let us start programming
23
Devank
10
```

The first statement prints a string literal, the second one prints the value of an expression, and the last two print the values of variables.

We can display multiple items in a single print call by separating the items with commas. Here are some examples:

```
name = 'Devank'
age = 10
print(name, age)
print('Age =', age)
print('Five times six is', 5 * 6)
print('My name is', name, 'I will be an adult
after', 18 - age, 'years')
```

Output-

```
Devank 10
Age = 10
Five times six is 30
```


My name is Devank I will be an adult after 8 years

The first statement with the `print` function prints two variables, the second one prints a string and a variable, the third one prints a string and an expression, and the fourth one prints a combination of strings, a variable, and an expression, all separated by commas. Note that only string literals are enclosed in quotes, while other items are written without quotes.

When we use the `print` function to display multiple items, all the items are separated by a single space in the output. If we want to change this default behavior and want the items to be separated by something else, then we can specify a separator by adding a `sep` parameter at the end of the `print` call. We will learn about the term parameter when we discuss functions. At this point, you just need to know that you can write `sep=` followed by a string literal that you want to be used as the separator.

```
day = 9
month = 11
year = 1977
print(day, month, year, sep='/')
print(day, month, year, sep='-')
print(day, month, year, sep='::')
print(day, month, year, sep='')
```

Output-

```
9/11/1977
9-11-1977
9::11::1977
9111977
```

In the first `print` call, we have specified `'/'` for the `sep` parameter, so each value in the output is separated by a `'/'`. Similarly, for the second `print` call, each value in the output is separated by a dash, and in the third `print` call, it is separated by two colons. If we do not want anything to be printed between the values, we can specify an empty string for the `sep` parameter, as we have done in the fourth `print` call. In this case, nothing is printed between the values, and so all the values are just joined together in the output.

From the output of our programs, we can see that every `print` call ends with a newline. This means that after printing everything, the cursor automatically moves to the next line. Thus, the output of the next `print` call starts with a fresh line. If we want the `print` call to end with something else instead of a newline, we can specify the `end` parameter. For example, `end='?'` will end the line with a question mark.

```
print('Hello world', end='---')
print('Python is easy', end=' ')
print('Python is interesting!', end='')
print('Programming is fun')
print('Good Bye')
```

Output-

```
Hello world---Python is easy Python is
interesting!Programming is fun
Good Bye
```

In the first `print` call, we have specified `'---'` for the `end` parameter, so the output of this call ends with `'---'` instead of a newline. Similarly, the output of the second `print` ends with a space because of the `end` parameter. The third `print` has an empty string as the `end` parameter, so it prints nothing at the end, and the last two `print` calls end with the default newline. If required, we can write both the `sep` and `end` parameters in a single `print` call to specify our own custom separator and custom line ending. This gives us more control over the format of our output.

You can write a `print` call with empty parentheses to insert an empty line in the output. For example, when the following code is executed, two empty lines will be printed between the two lines of text.

```
print('Let us start programming')
print()
print()
print('Python is interesting')
```

Output-

```
Let us start programming
```

Python is interesting

There are other ways of formatting the output, which we will learn in the next chapter.

2.15 Getting user input

A program that does not take any input from the user will essentially perform the same computations and will produce the same output every time it is executed. Most of the time, we have to write programs that interact with the user and behave differently depending on the user's response. To write such interactive programs, we should know how to get input from the user and use it in our program.

The built-in function `input()` can be used to get keyboard input from the user. When the `input()` function executes, the program is paused, and the user is expected to enter some text on the screen.

```
print('Enter name of a city : ', end='')  
city = input()  
print('You entered', city)
```

When you execute this code, first, the message of the `print` call is displayed on the screen. After this, the `input` function is called; this call pauses the program, and the interpreter waits for the user to enter some text. The user types the input and ends the input by pressing the Enter key, and after this, the program execution continues. The `input` function returns the entered text as a string, which means that a string object is created. To use this string in our program, we have to assign it to a variable. In our program, we have assigned the string to a variable named `city`. After this, we used the variable `city` in a `print` function call. Here are two sample runs of the program:

Sample Run 1-

```
Enter name of a city : Bangalore  
You entered Bangalore
```

Sample Run 2-

```
Enter name of a city : Bareilly
```

```
You entered Bareilly
```

The `input()` function captures the data entered by the user in a string, and that data can be used in the program by using the variable name.

Before asking the user to input something, we need to print a clear message telling the user exactly what kind of data to enter. This message is called a prompt. We have displayed this message by using the `print` function, but the `input` function is also capable of displaying the prompt. Writing a separate `print` function for the prompt is not required; we can place the prompt inside the parentheses of the `input` function.

```
city = input('Enter name of a city : ')
print('You entered', city)
```

This `input` function call first prints the prompt and then returns the text entered by the user as a string. In our next example, we are going to enter salary, display it, and then increment by using the augmented assignment syntax, and then again display it.

```
salary = input('Enter salary : ')
print('Initial salary', salary)
salary += 200
print('Incremented salary', salary)
```

Here is a sample run of the program -

```
Enter salary : 1200
```

```
Initial salary 1200
```

```
Traceback (most recent call last):
```

```
  File "C:\Users\test.py", line 3, in <module>
```

```
    salary += 200
```

```
TypeError: can only concatenate str (not "int") to
str
```

We got a `TypeError`, because the `input` function always returns the user input in the form of a string. We typed 1200 on the screen, and it was returned as the string '1200'. You can check the type of variable salary by

using the `type` function. It will show `str`. When we added 200 to `salary`, the interpreter complained, saying that the two types are different and it cannot perform implicit conversion. We want `salary` to be of numeric type since we will have to do arithmetic calculations with it. The type `str` does not support arithmetic operations, so we will perform an explicit conversion here.

```
salary = int(input('Enter salary : '))
print('Initial salary', salary)
salary += 200
print('Incremented salary', salary)
```

We enclosed the call to the `input` function inside the `int` function, so now the value returned by the `input` function is converted to `int`. Now, when we run it, it gives the expected output.

```
Enter salary : 1200
Initial salary 1200
Incremented salary 1400
```

Now, if we check the type of `salary`, it will show `int`. The `input` function always returns a string; it is your responsibility to convert the data returned by `input` to the required type. So, when you expect a numeric input from the user, make sure to convert the input to a numeric type using the correct conversion function.

2.16 Complete programs

Now we know enough basic concepts to start writing short and simple but complete programs. We know how to get input from the user, perform some basic calculations, and how to print and format output. So let us start writing some programs.

I. Write a program that enters two numbers and displays their sum, product, and difference.

```
n1 = int(input('Enter first number : '))
n2 = int(input('Enter second number : '))
```

```
print('Sum =', n1 + n2)
print('Difference =', n1 - n2)
print('Product =', n1 * n2)
```

II. Write a program that enters height in inches and displays it in feet and inches.

```
ht_inches = int(input('Enter the height in inches : '))

ft = ht_inches // 12
inches = ht_inches % 12
print(ft, 'feet', inches, 'inches')
```

III. Write a program that inputs the length and breadth of a rectangle and displays its area, perimeter, and length of the diagonal.

```
length = float(input('Enter length of rectangle in cm : '))

breadth = float(input('Enter breadth of rectangle in cm : '))

area = length * breadth
perimeter = 2 * (length + breadth)

diagonal = (length*length + breadth*breadth) ** 0.5
print('Area of rectangle is ', area, 'sq cm')
print('Perimeter of rectangle is ', perimeter, 'cm')
print('Diagonal of rectangle is ', diagonal, 'cm')
```

IV. Write a program that prompts the user to enter the values of principal, interest rate, and time and compute simple interest and compound interest. Formulas for calculating simple interest and compound interest are:

simple interest = $(\text{principal} * \text{rate} * \text{time}) / 100$

compound interest = amount – principal, where amount = $\text{principal} (1 + \text{rate} / 100)^{\text{time}}$

```
principal = float(input('Enter the principal : '))
```

```
time = int(input('Enter the time in years : '))
rate = float(input('Enter the interest rate : '))
simple_interest = (principal * time * rate) / 100
print('Simple interest is ', simple_interest)
compound_interest = principal * (1 + rate / 100) **
time - principal
print('Compound interest is ', compound_interest)
```

V. Write a program that prompts the user to enter a student name and marks in 3 subjects. Calculate the percentage marks and display the student's name with the percentage.

```
name = input('Enter name : ')
marks_maths = int(input('Enter marks in maths : '))
marks_physics = int(input('Enter marks in physics : '))

marks_chemistry = int(input('Enter marks in
chemistry : '))
total_marks = marks_maths + marks_physics +
marks_chemistry
percentage = (total_marks/300) * 100
print(name, percentage)
```

We would suggest you type in the programs shown in the book, run them, modify them, and experiment with them in different ways. Initially, while typing and coding, you will make mistakes that the interpreter will flag as errors. Fixing these mistakes and getting your program to run is an integral part of the learning process. It will help you become familiar with the syntax and features of the language. Active engagement with code will also help you to understand and retain the concepts and have a solid grasp of the topic. This hands-on approach is the most effective way to learn programming.

2.17 Comments

In your program file, you can not only write Python code but can also include notes to explain the code. This becomes more important if your programs are lengthy and complicated and there is a team of programmers working together. When you are developing a program, you are deep into it and have an understanding of how it works. However, upon revisiting the code later, you might forget how you made things work. Understanding a complicated program by just looking at the code is difficult; reading the notes will help you understand the code faster and save you time. This is also true for other fellow programmers who need to read and understand your code.

These notes are called comments in programming languages. A comment is a piece of text that is inserted in between the code to explain the purpose of your code to other programmers or to yourself when you revisit the code. Code that is properly documented with comments makes the program more readable and understandable, and so it is easier to maintain and update. Comments are written only for human readers; they are ignored by the interpreter, so they have no effect on the execution of the program.

In Python, a comment starts with a hash sign (#) and lasts till the end of the current line. Any text after the # sign till the end of the line will not be executed. The interpreter just ignores it. A comment can be written on a new line or after a statement on the same line. [Figure 2.22](#) shows a code snippet that contains some comments.

Do not try to understand the code because many structures used in it have not been introduced yet. The code is here just to illustrate how comments are used to explain the purpose of the code. Comments should not be written for code that is doing something obvious; such comments are unnecessary and should be avoided.


```

# This program sorts a list by using Quick sort technique
def quick_sort(a):
    sort(a, 0, len(a)-1)
def sort(a, low, up):
    if low >= up: # zero or one element in sublist
        return
    p = partition(a, low, up)
    sort(a, low, p-1) # Sort left sublist
    sort(a, p+1, up)  # Sort right sublist
def partition(a, low, up):
    pivot = a[low]
    i = low+1 # moves from left to right
    j = up    # moves from right to left
    while i <= j:
        while a[i] < pivot and i < up:
            i+=1
        while a[j] > pivot:
            j-=1
        if i < j: # swap a[i] and a[j]
            temp = a[i]
            a[i] = a[j]
            a[j] = temp
            i+=1
            j-=1
        else: # found proper place for pivot
            break
    # Proper place for pivot is j
    a[low] = a[j]
    a[j] = pivot
    return j
list1 = [6, 3, 1, 5, 9, 8]
quick_sort(list1)
print(list1)

```

Figure 2.22: Comments in a Python program

If you need to write a multi-line comment (block comment), then you have to precede each line with the # sign. In IDLE, you can easily comment multiple lines by selecting those lines, going to the Format menu, and selecting Comment region.

In addition to documentation, there is another use of comments. You can use comments to disable part of your program while testing or debugging. Debugging is the process where you are trying to find out why the code is not working. You can temporarily comment some parts of the program that you think might be creating problems.

The code that is commented out will not be executed when you run your program. So, if your program is not working as expected, then you can

comment a piece of code and see if the code runs fine. Text editors generally have the facility of commenting out pieces of code, so you do not have to manually put a # sign in front of each line that you want to disable. Later, you can remove the commenting signs from your disabled code by choosing the Uncomment option in your editor.

2.18 Indentation in Python

Indentation is the whitespace (spaces or tabs) that is present before the beginning of a code line. In most of the languages, indentation is done just to increase the readability, but in Python, it is very important. Python forces programmers to structure their code through indenting. So, indentation is not a matter of style, but a part of syntax in Python. Indentation of each line matters; wrong indentation can result in either an indentation error or incorrect behavior of your program.

Python uses indentation for grouping statements to form code blocks. In the code snippet that we saw in [section 2.17](#), we can see the code blocks being defined with different indentations. Continuous statements with the same indentation belong to the same code block. Higher levels of indentations indicate nested code blocks. Unlike other languages, Python does not use braces or words like begin or end to define the boundaries of blocks of code. It uses indentation for this purpose. As we move through the chapters and learn about different compound statements such as if..else, while, for, def, etc., we will see how to use indentation for defining blocks.

The code that we have written till now is top-level code of the file; it is not indented, which means that there should be no whitespace at the beginning of the statement. So, till we get introduced to compound statements, we will write all our code without any indentation. The following program will give an indentation error if you try to execute it.

```
name = input('Enter name : ')
    age = int(input('Enter age : '))
print(name, age)
```

The error is caused due to an unexpected indentation in the second statement. The program will execute if you remove the two spaces present at

the beginning of the second statement.

2.19 Container types

In the next few chapters, we will learn in detail about the built-in data structures or collection types in Python -lists, tuples, dictionaries, and sets. These are also called containers, as they provide a way of combining and organizing data. These data structures are used to hold different types of objects. Here are some examples of literals of these types:

Lists (type `list`) `[1, 2, 3, 4]`

Tuples (type `tuple`) `(1, 2, 4, 5)`

Dictionaries (type `dict`) `{'a': 1, 'b': 2, 'c': 3}`

Sets (type `set`) `{2, 3, 4, 6, 8}`

Lists and tuples are sequence types in Python, which means that they are ordered collections of values. These types contain a left-to-right order among the items that they contain. We can tell which one is the first element, which is the second, which is the last, and so on. In these types, the contained items are accessed using their positions. Dictionaries are the mapping type as they store objects by keys. In Python 3.6 and earlier versions, dictionaries were unordered, but from version 3.7, they are ordered. Sets are neither mapping nor sequences; they are just collections of unique objects.

2.20 Mutable and Immutable Types

We know that each object has a type, an id, and a value. The type and id of an object remain the same throughout the program; they cannot be changed. Whether the contained value can be changed or not depends on the mutability of the object.

Python types can be categorized as either mutable or immutable depending on whether the value of an object of that type can be changed or not. If a type is immutable, the value inside an object of that type cannot be changed. You can never overwrite the value of an immutable object. If a type is

mutable, then the value contained inside the object of that type can be changed at run time. Here are some immutable and mutable types in Python.

Immutable - bool int float str tuple
frozenset

Mutable - list set dict

Mutable types support operations that can change the value inside the object at run time, while immutable types do not provide any operation that can change the value inside the object. The state of an immutable object is fixed at the time of creation and cannot be modified later.

You need to keep in mind that mutability has nothing to do with the variable names. Let us try to understand this. Suppose the variable name `x` refers to a mutable object, and the name `a` refers to an immutable object.

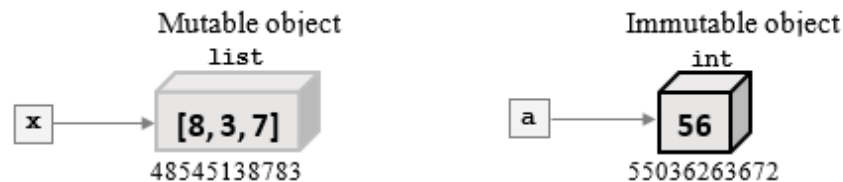


Figure 2.23: Mutable and Immutable objects

The object to which `x` is referring is a mutable object, so the value inside it can be changed. We generally say that it can be changed in-place. The object to which `a` is referring is an immutable object, so it will remain as it is throughout its lifetime. You cannot overwrite it; it cannot be changed in-place.

The variable referencing any object can always be reassigned to a different object; we can make `x`, or `a` refer to a different object. So, mutability is associated with types and objects, not with variables.

You need to clearly understand the difference between the terms rebinding a variable and mutating an object. Rebinding a variable means making a variable refer to a different object, and mutating an object means making in-place changes in that object. Only mutable objects can be mutated. In our example, the variable `a` refers to an `int` object with value 56; the operations like `a = a + 3` seem to change the value, but remember this is rebinding. `int` is an immutable type, so you cannot modify the value inside

an object once it is created. You can only create a new object with a different value. The value 56 inside the object is not changed to 59. Instead, a new object with value 59 is created, and `a` refers to that object. So, any operation on the immutable types that seems to modify the value results in the creation of a new object with the modified value.

If a variable refers to an immutable object, you can see changes in that variable only by rebinding that variable. If a variable refers to a mutable object, you can see changes in that variable by rebinding it or by making in-place changes in the object that it is referring to. In our example, we have variable `x` referring to a `list` object and variable `a` referring to an `int` object. We can see changes in `x` by rebinding `x` to a different object or by changing in-place the list object that it is currently referring to. We can see changes in `a` only when we rebind it to a different object.

Mutability matters when there are multiple references referring to an object. Suppose we have three variables `x`, `y`, and `z`, that refer to a `list` object and three variables `a`, `b`, and `c` that refer to an `int` object.



Figure 2.24: Multiple references to objects

If we make any in-place changes to the list object through any of the variables `x`, `y`, or `z`, then that change will be visible in the other two variables also because all three of them share the same object. In the case of immutable objects, these types of side effects will not occur because they cannot be changed in-place. This distinction is very important to understand, and it will become clearer as we proceed through the chapters and cover some of the mutable and immutable types in detail.

2.21 Functions and methods

We will talk about functions and methods in detail later, but since we will be using built-in functions and methods in the next few chapters, you need to have a general idea of what they are and how you can use them.

A function is a reusable piece of code with a name, and it can perform certain operations for you. You can give it some values called arguments; it performs some work for you, and it might give you a value back. The built-in functions are the functions that are already written for us and are always available, so we can easily use them. We have already used some built-in functions, like `print`, `input`, `type`, and `id`. We know that to call them, we need to write their name followed by parentheses, which can include some arguments. Arguments are values that provide some information to the function for performing its work. If the function doesn't need any arguments, then the parentheses remain empty. Here are some examples of built-in functions:

`abs(x)` - Returns absolute value of `x`

`bin(x)` - Returns binary equivalent of `x`

`oct(x)` - Returns octal equivalent of `x`

`hex(x)` - Returns hexadecimal equivalent of `x`

`max(a, b, c, .....)` - Returns maximum value among the provided arguments

`min(a, b, c, .....)` - Returns minimum value among the provided arguments

A method is like a function, but it is specific to a type, and we access it by using a dot. To call a method, we write the variable name or a literal followed by a dot, then the method name, and then the parentheses, which can include arguments.

```
'hello'.upper()
```

```
list1.append(10)
```

Here, we are calling the method `upper` on a string literal, and we are calling the method `append` on a variable named `list1` that refers to a `list` object.

Functions are not specific to any type, so they are called independently without the dot syntax. You can think of functions as generic operations that can work with multiple types. For example, the built-in function `len` can be used to find the length of a string, list, or a dictionary. Methods are type specific operations that are attached to types and can act on an object of a specific type only. For example, the method `upper` can act only on an object of type `str`, and the method `append` can act only on an object of type `list`. We will discuss most of the methods related to the types that we will see in the coming chapters.

A type defines many methods, and it is not possible to remember all methods associated with a particular type. So whenever required, you can go to the interactive prompt and get a listing of all the methods. To know about the methods available for a data type, just type `dir(typoname)` on the interactive prompt, and it will show you all the methods available for that type. For example, to see all the methods for the `list` type, we can write:

```
>>> dir(list)

['__add__', '__class__', '__class_getitem__',
 '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__',
 '__gt__', '__hash__', '__iadd__', '__imul__',
 '__init__', '__init_subclass__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__sizeof__',
 '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert',
 'pop', 'remove', 'reverse', 'sort']
```

This is the result that we get. There will be lots of methods with leading and trailing underscores, and these methods represent the implementation details of the type and help in customization. The methods towards the end are the ones without any underscore, and these are the methods that we will be mostly using.

This command shows you the names of methods. If you want to know more about a particular method, you can use `help`. On the interactive prompt, write `help`, then inside parentheses, write `typename` followed by a dot and then the method name.

```
>>> help(list.append)
Help on method_descriptor:
append(self, object, /)

    Append object to the end of the list.
```

```
>>> help(str.upper)
Help on method_descriptor:
upper(self, /)

    Return a copy of the string converted to
    uppercase.
```

Here, we are getting help on the `append` method of `list` type and the `upper` method of `string` type.

These functions `dir()` and `help()` accept both the type name or a variable name. So, suppose you have a variable `s` referring to a string object, you can use `dir` and `help` on `s` also. If you write `help(typename)` then it will show you the description of all the methods.

2.22 Importing

There are many predefined functions in the standard library that we can use in our program, but unlike built-in functions, these functions are not automatically available in our program. These functions are organized in modules (Python files), and we have to import them to make them available in our program. For example, the `math` module contains many mathematical functions. The `random` module provides functions for randomization. In the following code, we are importing and using `sqrt` and `trunc` functions from the `math` module.

```
from math import sqrt, trunc
```



```
x = 34
y = 23.4
print(sqrt(34))
print(trunc(23.4))
```

Output-

```
5.830951894845301
23
```

If you import a module by writing `import modulename`, then all the names in that module can be used in your program, but they have to be preceded by the module name and a dot.

```
import math
x = 34
y = 23.4
print(math.sqrt(34))
print(math.trunc(23.4))
```

We can import modules from the rich standard library and make use of lots of pre-existing functionality, and that is why the term 'batteries included' is used for Python. You can see a list of standard library modules in the official Python documentation, and to know more about a module, import it on the shell and use `help` on it.

```
>>> import math
>>> help(math)
```

To see all the available names in a module, you can use the `dir` function after importing it.

```
>>> dir(math)

['__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'acos', 'acosh', 'asin',
 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil',
 'comb', 'copysign', 'cos', 'cosh', 'degrees',
 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1',
 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
```

```
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',  
'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm',  
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',  
'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',  
'prod', 'radians', 'remainder', 'sin', 'sinh',  
'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

To get help on a specific name from the module, use help on that name.

```
>>> import math  
>>> help(math.floor)  
Help on built-in function floor in module math:  
floor(x, /)  
    Return the floor of x as an Integral.
```

2.23 Revisiting interactive mode

We know that when we enter a statement on the shell prompt, it will be executed, and when we enter an expression, it will be evaluated, and its value will be printed. This automatic printing is there only in interactive mode. In script mode, you need to use the `print` function to print the value of expressions. The `print` function works in the interactive mode also, but is not required so you can save some typing.

```
>>> a = 10  
>>> a  
10  
>>> print(a)  
10
```

A single underscore is a valid identifier name in Python, and in scripts, it is used to ignore values, as we will see later. In interactive mode, a single underscore (`_`) is a special variable name that stores the result of the last expression that was evaluated. You can use this variable in another expression on the prompt.

```
>>> a = 5
```

```
>>> a + 2
7
>>> _
7
>>> _ + 5
12
```

When you enter multiline statements on the interactive prompt, the prompt string changes from ">>>" to "....".

```
>>> total_marks = marks_science + marks_maths \
...               + marks_english + marks_socials \
...               + grace_marks
>>> months = [ 'January', 'February', 'March',
'April',
...           'May', 'June', 'July', 'August',
...           'September', 'October', 'November',
'December' ]
```

So, when you type something that occupies more than one line, the prompt changes to three dots.

While experimenting in the interactive mode, you would often make mistakes and get errors. In those cases, you would want to edit and rerun your previous command. You can do this without retyping the previous command by making use of the command history. Each interactive session maintains a history of all the commands that you type at the shell prompt. You can scroll through these commands by pressing "Alt+P" (for the previous command and "Alt+N" (for the next command). On Mac, you have to use Command-P and Command-N. Up and down arrow keys can also be used on some systems for scrolling through commands. By using arrow keys, take the cursor on the desired command and then press Enter to select that command. You can also click on a previous command and press Enter to get that command on your prompt. Once you get a command displayed on your prompt, you can either edit it and then execute it, or you can just execute the command as it is.

We know that when a program is executed, its output appears in the shell window. When the execution of the program is over, the Shell window retains focus and displays a shell prompt. Now, you can explore the result of the program execution on this prompt. For example, you can see the final values of variables you defined in the program. Any names that you had defined in the program would be available in the Shell window after the execution of the program.

2.24 Errors

As you start writing programs, you will encounter many errors in your programs. Understanding and fixing errors is a part of the learning process. It improves our understanding of the language and problem-solving skills. We can broadly categorize errors into three types – syntax errors, run time errors, and logical errors.

Syntax is a set of rules that define how the code instructions should be written in a language. In the previous chapter, we saw that our source code is compiled before being executed by PVM. During the compilation step, the compiler checks the syntax of each instruction and translates it to bytecode. When it finds anything written in the wrong syntax, it stops the translation and displays an error message. These errors are called syntax errors or parsing errors, and they occur due to the incorrect syntax of the code. For example, you might miss a colon or a quote or use an unbalanced pair of parentheses. When there is a syntax error in your program, and you try to run the program, IDLE shows a dialog box, and it also highlights the location where the syntax error is detected in your program. You need to fix the error by making changes in your code and running the program again. As a beginner, you will find yourself making many syntax errors, but as you get used to the language, their frequency will reduce. It is generally not very difficult to identify and remove these errors from your program. Some development environments (not IDLE) underline the syntax errors as you type the code.

If there are no syntax errors, the byte code is generated, and your program enters run time. The byte code goes through the Python Virtual Machine, which executes it by converting it to machine code. Run time is the time when your program is executing; during this time, your program will interact

with the user and might be connected with multiple external resources. If an error occurs during this time, then the execution of the program stops immediately, and it is terminated with an error message. Any error that occurs at this run time is called a run time error. There are some run time errors that are caused due to some mistake in your code, and they can be removed by modifying your code. Some run time errors occur due to unusual events at run time, and they are not under the control of your program. To handle them, you have to write the error handling code, which we will discuss in *Chapter 20*.

Logical errors occur when your program runs smoothly and gives you the output, but the output that it gives is not what was intended, so your program works, but it doesn't do what you expect it to do. These errors occur due to the wrong logic of the code that you have written. The problem is not with the code. The program does exactly what it has been told to do. The problem is that the programmer was not able to communicate properly the solution in the form of code, or maybe the solution that the programmer has come up with is not correct. It could be due to things like a missing assignment, the use of a wrong operator, or an incorrect algorithm. These types of errors are not reported by the interpreter. The programmer has to identify them, so these errors are the most difficult to detect and remove. You have to examine your code and debug the program, and at times, take the help of a debugger.

2.25 PEP8

Python Enhancement Proposals (PEPs) are documents that describe a new Python feature or provide information to the community. There are many PEPs that are listed in PEP0 document, which is the index of PEPs and can be accessed at <https://peps.python.org/pep-0000/>. Of these PEPs, the most useful for Python programmers is PEP8, which is a style guide for writing Python code.

PEP8 was written by Guido van Rossum, Barry Warsaw, and Nick Coghlan in 2001. You can read it online at <https://peps.python.org/pep-0008/>. This document provides various coding conventions and best practices to write readable and consistent code in Python. According to Guido van Rossum, "Code is read much more often than it is written", and according to Zen of Python, "Readability counts." Readability and consistency are important

because the code is written once but read many times by different people for various reasons, like collaborating on a project or debugging and adding new features. Writing PEP8-compliant code will make it easier for you and others to read and understand your Python code.

The guidelines in the document are only recommendations; if you write code that does not conform to PEP8, your code will still work as long as it follows the syntax of the language but might not be considered professional by the Python community. Therefore, it is good to be aware of the best practices and develop a habit of writing code that adheres to the community guidelines.

The PEP8 document includes coding conventions for indentation, whitespaces, naming things, and other coding constructs that we have yet to learn. We will see the conventions as we get introduced to the coding structures. However, I would recommend you to read the document at least once. There are many tools and IDEs that will automatically format your code according to PEP8.

Exercise

1. Which of the following cannot be used as a variable name?

(A) Null (C) Nil

(B) None (D) Not

2. Which of the following is not a valid identifier name?

(A) min_marks

(B) marks2

(C) net-sales

3. Which of these are the literal values of bool type?

(A) true, false

(B) TRUE, FALSE

(C) True, False

4. Python is a case-sensitive language.

(A) True (B) False

5. `x = 56.6`

What will be the type of `x`?

(A) rational (C) float

(B) int (D) decimal

6. All keywords in Python are in lowercase.

(A) True (B) False

7. Which of the following is not a valid `int` literal in Python?

(A) `0o356`

(B) `0x1009`

(C) `10,000`

8. Which of the following will give an error?

(A) `a = 0x3F`

(B) `b = 0o496`

(C) `c = 0b110`

9. Which of the following is not a valid float literal?

(A) `.98` (C) `9e8`

(B) `9.8` (D) All are valid

10. `x = "True"`

What will be the type of `x`?

(A) `int`

(B) `str`

(C) `bool`

11. The value contained in an object cannot be changed if the object belongs to _____ type, and the contained value can be changed if the object belongs to _____ type.

(A) a mutable, an immutable

(B) an immutable, a mutable

12. `x = 960`

`y = x`

Do x and y reference the same memory location?

(A) Yes (B) No

13. `4e-5` is equivalent to

(A) `0.000004`

(B) `400000.0`

(C) `0.00004`

14. Python is a _____ typed language.

(A) statically (B) dynamically

15. Which of these functions can be used to get the identity of an object?

(A) `identify()`

(B) `id()`

(C) `identity()`

16. A Python object can be dynamically assigned to any variable in Python.

(A) True (B) False

17. Existence of a variable name in Python begins with _____

(A) a declaration

(B) an assignment statement

18. To delete a variable named `x`, what will you write?

(A) `delete x`

(B) `del x`

(C) `remove x`

19. Which of these operators can be used both as a unary operator and a binary operator.

(A) `%` (C) `/`

(B) `-` (D) `*`

20. Which of these is the exponentiation operator in Python.

(A) `%` (C) `*`

(B) `^` (D) `**`

21. Which of the following expression evaluates to False

(A) `3 == 3` (C) `3 <= 3`

(B) `3 != 3` (D) `3 >= 3`

22. What will be the value of expression `23 / 2`?

(A) `11` (C) `11.5`

(B) `11.0` (D) `12`

23. What will be the values of expressions `23//2` and `-23//2` ?

(A) `11`, `-12`

(B) `11.0`, `-12.0`

(C) `11.5`, `-11.5`

24. Value of expression `36 ** 0.5` is

(A) `6.0`

(B) `6`

(C) `12.0`

25. Value of expression `23 % 3` is

(A) `2`

(B) `3`

(C) `7`

26. `x = 10 // 3`

What will be the type of `x`?

(A) `int` (B) `float`

27. What is the value of the expression `not (4 > 8)` ?
(A) True (B) False
28. Which one is an equivalent logical expression for `not (a > b)` ?
(A) `a < b`
(B) `a > b`
(C) `a <= b`
29. Which one is an equivalent logical expression for `a < 50` and `a > 4`?
(A) `4 < a < 50`
(B) `50 < a < 4`
(C) `a < 4 < 50`
30. The expression `p <= q < r <= s` is equivalent to
(A) `p <= q` and `q < r` and `r <= s`
(B) `p <= q` or `q < r` or `r <= s`
31. `x == y` will return `True` only when both `x` and `y` refer to the same object
(A) Yes (B) No
32. If `a == b` is `True`, the expression `a is b` will definitely be `True`.
(A) Yes (B) No
33. A single line comment in Python begins with _____
(A) `$` (C) `#`
(B) `/*` (D) `//`
34. What is the value of the expression `2 ** 2 ** 3`?
(A) 64 (B) 256
35. What is the value of the expression `27 / 3 / 3`?
(A) 27.0
(B) 9.0

(C) 3.0

36. Which of these operators has right to left associativity?

(A) + (C) **

(B) * (D) //

37. Which of these symbols is the line continuation symbol?

(A) # (C) /

(B) \$ (D) \

38. Which of the following expressions will give error?

(A) 2+30) / (5-3)

(B) (4+3) (3-5)

(C) None of these

39. What will be the output of the following print call?

`print(2,000,000)`

(A) 2,000,000 (C) 2e6

(B) 2000000 (D) 2 0 0

40. Which of the following expression shows explicit type conversion?

(A) `int(9.8) + 7.3`

(B) `3.4 + 5.4`

(C) `7 % 2`

(D) `17.5 % 3`

41. Which of the following expression involves implicit type conversion?

(A) `int(9.8) + 7.3`

(B) `7 % 2`

(C) `17.5 % 3`

42. What will be the values of expressions?

`3.5/0.2, int(3.5)/0.2, int(3.5/0.2)`

43. What will be the output of the following print call
`print(3.0e250 * 1.6e150)`

(A) `4.8e+400` (B) `inf`

44. What will be the output of the following print call `print(2.4e-250 / 1.2e200)`

(A) `2.0e-450` (B) `0.0`

45. An object can have only one name associated with it.

(A) True (B) False

46. In Python, types are associated with _____

(A) Objects (B) Variables

47. `del` statement deletes

(A) Variable names (B) objects

48. What is the value of the expression `35 == '35'` ?

(A) True (B) False

49. Correct the following print call so that it correctly prints the strings literals and values of variables.

```
name = 'Devank'
```

```
age = 10
```

```
print('My name is, name, and age is, age')
```

50. What will be the values of expressions `11//3`, `int(11//3)` , `-11//3` and `int(-11//3)` ?

What will be the output of code given in questions 51 to 65?

51. `a = 5`

```
print(3 < a < 10)
```

52. `x = 5`

```
x++
```

```
print(x)
```

53. `m = 12`

`n = m = m-10`

`print(m, n)`

54. `n = 5`

`n *= n-1`

`print(n)`

55. `x = 2`

`y = 4`

`x + 4`

`y + 5`

`print(x, y)`

56. `n1 = 9`

`n2 = 3`

`n3 = 6`

`average = n1 + n2 + n3 / 3`

`print(average)`

57. `a = 2`

`b = 3`

`a+1 = b`

`print(a, b)`

58. `x = 0581`

`x +=1`

`print(x)`

59. `x = 2`

`y = 3`

`print(x =< y)`

```
60. salary = 1000
    raise = 100
    new_salary = salary + raise
    print(new_salary)
```

```
61. x = 5
    y = 6
    print(x + y)
```

```
62. print('Hello', end = ',')
    print('Hi', end = ',')
    print('Hey', end = ',')
```

```
63. a = 5
    b = 6
    c = 11
    print(a<b or b<10 and c<a)
```

```
64. x = +92
    y = -92
    print(x, y)
```

```
65. print('Hello world')
    print = 4
    print(2 + 5)
```

66. What will be the output of the following program if numbers 2 and 5 are entered when it is executed?

```
n1 = input('Enter first number : ')
n2 = input('Enter second number : ')
x = n1 + n2 * 3
print(x)
```

67. Write a program that enters mass in grams and displays it in grams and kilograms.

68. Write a program that inputs temperature in Celsius and converts it to Fahrenheit. The formula for conversion is -

$$\text{Temperature in Fahrenheit} = \text{Temperature in Celsius} * 1.8 + 32$$

69. Write a program that prompts the user to input his/her weight in kgs and height in cms, and calculates the body mass index (BMI). BMI is calculated by dividing body weight in kgs by the square of height in meters. For example, if weight is 70 kg, and height is 170 cm, then BMI is $70/(1.7 * 1.7) = 24.2$

70. Write a program that inputs radius of a circle, and displays its area and circumference.

$$\text{Area of a circle} = \pi * \text{radius} * \text{radius}$$

$$\text{Circumference} = 2 * \pi * \text{radius}$$

Import the value of pi(π) from the `math` module.

71. Write a program that enters a phone number and prints its last 3 digits.

72. Write a program that accepts an integer in decimal form and prints it in binary, octal, and hexadecimal. Use built-in functions `bin`, `oct`, and `hex`.

73. Write a program that enters 4 numbers and prints the largest and smallest number. Use built-in functions `max` and `min`.

74. Write a program that enters two numbers and finds the greatest common divisor of those two numbers. Use `gcd` function from the `math` module.

75. Write a program that enters two numbers and generates a random number between those two numbers. Use `randint` function from the `random` module.

76. Write a program that enters the base and height of a right-angled triangle and finds its hypotenuse According to Pythagoras theorem $\text{Hypotenuse}^2 = \text{Base}^2 + \text{Height}^2$

Use `sqrt` function from the `math` module.

Data comes in many forms; the most common form is textual data. Almost every program that does something useful must input, store, process, and output text. In programming, textual data is handled with the help of strings.

A string is a sequence of characters. In Python, the type `str` is used to represent a string. In your program, you can specify a string literal by enclosing a sequence of characters in either single quotes or double quotes. A string literal can contain zero or more characters, including letters, digits, special characters, and space. The enclosing quotation marks are not stored as part of the string; they are used to delimit the string. Here are some examples of string literals:

<code>' '</code>	Empty string
<code>'abc'</code>	String with 3 characters
<code>'a'</code>	String with 1 character
<code>' '</code>	String containing a single space
<code>'123abc!'</code>	String with both alphabetic and nonalphabetic characters
<code>'456'</code>	String containing digits
<code>"cdf"</code>	String literal enclosed in double quotes
<code>"don't shout"</code>	Single quotes inside a double-quoted string
<code>'Book "C in depth" 3 ed'</code>	Double quotes inside a single-quoted string

Table 3.1: *String literals*

If the single quote has to be used as an actual character inside the string, the string can be enclosed in double-quotes. If a double quote must be used as an actual character inside the string, the string can be enclosed in single quotes.

You can use single or double quotes to enclose the string literals in your program. Whichever style you choose, it is better to stick to it. It is not a good idea to mix the two styles. We will be mostly using single quotes in this book. Python also supports triple-quoted strings, which we will discuss later.

In Python, there is no character type that represents a single character. Single characters enclosed in quotes are considered strings of size 1.

A string literal can be assigned to a variable, and then various string-related operations can be performed on that variable.

```
>>> s1 = 'Morning'
>>> s2 = "Evening"
```

These assignments make variables `s1` and `s2` refer to string objects. The type of these objects is `str`.



Figure 3.1: Objects of type str

The interactive interpreter shows the string enclosed in single quotes, even if we define the literal using double quotes.

```
>>> s1
'Morning'
>>> s2
'Evening'
```

If we print the string using the `print` function, the enclosing quotes are not displayed.

```
>>> print(s1)
Morning
```

A string is a sequence of single characters. Other types of sequences in Python consist of lists and tuples, both representing sequences of objects. Sequences are types that maintain a left-to-right ordering among the

elements they contain. These sequence types have some similarities and share some capabilities. Operations like indexing, slicing, concatenation, and repetition apply to all sequence types. The knowledge of slicing will also come in handy while using advanced Python libraries like NumPy and Pandas. So, make sure that you understand these concepts and practice them thoroughly.

3.1 Indexing

To access a single character inside the string, we must specify a numeric index inside square brackets. Indexing is 0 based, so the first index is 0.

```
>>> s = 'quintessence'
```

If we want to access the individual characters of our string `s`, we can write `s[0]` for accessing the first character, `s[1]` for the second character, `s[2]` for the third character, and so on. If a string has `n` characters, the valid index values are from 0 to `n-1`. The string example that we have taken has 12 characters, so the valid index values are from 0 to 11, and thus `s[0]`, `s[1]`, ... , `s[11]` are valid expressions that give us individual characters of the string. `s[11]` will give us the last character of the string.

```
>>> s[0]
'q'
>>> s[11]
'e'
```

Any index value larger than 11 will give an error. It will be an error to write `s[12]` or `s[13]` or any other index greater than 11.

```
>>> s[12]
IndexError: string index out of range
```

Inside the square brackets, we can use any variable name or expression, provided the expression evaluates to an integer.

```
>>> i = 5
>>> s[i]
'e'
```

```
>>> s[i-3]
'i'
```

The built-in function `len` gives the length of the string, which is equal to the total number of characters in the string.

```
>>> len(s)
12
```

The expression `len(s) - 1` can be used as an index to access the last character of the string.

```
>>> s[len(s) - 1]
'e'
```

The length of string `s` is 12, so inside brackets, we will have 11, and we know that `s[11]` will give us the last character. Similarly, to get the second last character, we can write `s[len(s) - 2]`, and to access the third last character, we can write `s[len(s) - 3]`, and so on.

In Python, there is a shortcut for accessing characters from the end of the string. Instead of writing the expression `s[len(s) - 1]`, we can simply write `s[-1]`. So, if we want to access the last character of any string, we do not need to know the length of the string. We can access the character at index -1. Similarly, we can write `s[-2]`, which is equivalent to `s[len(s) - 2]` and hence gives the second last character.

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
quintessence											
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s[-1]
'e'
>>> s[-12]
'q'
>>> s[-13]
```

`IndexError: string index out of range`

Thus, in Python, it is not an error to write negative indices. We can go backward in a string using these negative index values. In general, if we have a string of length n , the valid indices are 0 to $n-1$ and -1 to $-n$.

Writing an index greater than or equal to n or less than $-n$ will raise an `IndexError`. For our string `s`, if we write any index greater than or equal to 12 or less than -12 , then an `IndexError` will be raised.

Indexing a string gives us a one-character string. In languages like C or C++, there is a separate character type to represent single characters, but in Python, there is no such type. A single character inside quotes is of type `str`.

3.2 Strings are immutable

In the previous chapter, we learned about the mutability of objects. An object of immutable type cannot be modified, while an object of mutable type is modifiable. Strings are immutable, meaning you cannot change a string object in any way. Suppose you have a string object, and the variable `s` refers to it.

```
>>> s = 'ring'
```



Figure 3.2: Variable `s` referring to a string object

```
>>> s[0] = 'p'
```

```
TypeError: 'str' object does not support item  
assignment
```

You cannot use the square brackets on the left side of the assignment operator to change any character inside the string. This is due to the immutability of the string object. Once it has been created, it cannot be altered in any manner. You cannot delete any character from the string, insert new characters, or replace anything. However, you can create a new string object and assign it to the same variable name.

```
>>> s = 'ping'
```

When this statement executes, a new string object is created, and the variable `s` starts referring to that new object.



Figure 3.3: Variable `s` referring to a new string object

We know that a variable in Python is just a name, and it can be reassigned any number of times and can refer to any type of object. So, the statement `s = 'ping'` is a valid statement since a new string object is created and the name `s` is reassigned. This statement does not change the string object in any way. It appears that we are changing the string, but we have just reassigned the variable.

The statement `s[0] = 'p'` is not valid since it is trying to change an immutable object in-place. This concept of objects, names, and assigning is very important to understand in Python. If you want to change a string, the only way is to assign the variable name to a new string object. New string objects can be created in many ways, like slicing, concatenating, or calling string methods.

By reassigning a string variable, you can change a string variable without violating the immutability of the string object. It might seem inefficient that a new string object is created every time a string must be changed. However, practically, it is not so, as Python's garbage collector will automatically reclaim the space occupied by any unused objects.

3.3 String Slicing

We have seen how to get a single character from a string by specifying an index using square brackets. Using the same square brackets, we can also access a portion of the string. It is called slicing the string. To extract a part of the string, we must specify 2 integers inside square brackets.

```
s[i:j]
```

Inside the square brackets, we have two integers, `i` and `j`, separated by a colon. The expression `s[i:j]` is a slice of the string; it gives us a new string object that is a copy of the portion of the string `s` from index `i` to index `j-1`. Note that the first index is included while the second index is excluded. So, the slice `s[i:j]` returns a new string object that contains all the characters of string `s`, from index `i` up to (but not including) index `j`. The original string object does not change. Let us see some examples:

```
>>> s = 'homogeneous'
>>> s[2:6]
'moge'
```

The expression `s[2:6]` gives us a new string object that contains all the characters of string `s` from index 2 to index 5. The sliced object can be assigned to a name.

```
>>> s1 = s[4:7]
>>> s1
'gen'
```

The name `s1` refers to the sliced object returned by the expression `s[4:7]`. The original object referred to by `s` remains unchanged.

```
>>> s
'homogeneous'
>>> id(s)
2182966396016
```

Now we make the name `s` refer to a new sliced object.

```
>>> s = s[3:7]
>>> s
'ogen'
>>> id(s)
2182965695664
```

`id` of `s` has changed, which shows that it refers to a new object.

While writing the slicing expression, we can omit the first or the second number or both. If we omit the first index, it is assumed to be 0, i.e., the beginning of the list. So, the slice `s[:j]` indicates a part of the string `s` from index 0 to index `j-1`. It is equivalent to writing `s[0:j]`. If we omit the second index, it is assumed to be the end of the string. So, the slice `s[i:]` indicates a part of the string `s` from index `i` to index `n-1` where `n` is the length of the string. It is equivalent to writing `s[i:n]`.

`s[:j]` Part of string `s` from index 0 to index `j-1` (same as `s[0:j]`)

`s[:7]` Part of string `s` from index 0 to index 6 (same as `s[0:7]`)

`s[i:]` Part of string `s` from index `i` to index `n-1` (same as `s[i:n]`, `n` is length of string)

`s[3:]` Part of string `s` from index 3 to index `n-1` (same as `s[3:n]`, `n` is length of string)

We can omit both the indices inside the brackets. Therefore, the slice `s[:]` extracts the entire string from the beginning till the end. It gives an exact copy of the entire string. It is the same as writing `s[0:n]`.

`s[:]` Part of string `s` from index 0 to index `n-1` (same as `s[0:n]`, `n` is length of string)

So, when slicing from the start of the string, we can omit zero, and when slicing to the end of the string, we can omit `n`, as they are redundant. Here are some examples:

```
>>> s = 'homogeneous'
>>> s[:4]
'homo'
>>> s[5:]
'eneous'
>>> s[:]
'homogeneous'
```

Omitting both indexes gives us a string object that is an exact copy of the string. So, if we must make a new string that is a copy of the string, we can do it this way.

```
>>> scopy = s[:]
>>> scopy
'homogeneous'
```

You can specify a negative index also while slicing.

`s[0:-1]` Part of string `s` from index 0 to index `-2` (same as `s[0:n-1]`)

The slice `s[0:-1]` indicates a part of the string from index 0 to index `-1-1` i.e. `-2`. As we have seen earlier, writing 0 as the first integer is redundant, so you can omit the zero and just write it as `s[:-1]`.

The slice `s[:-1]` represents the whole string, excluding the last character. If you want a part of the string that excludes the last two characters, you can use the slice `s[:-2]`. In general, `s[:-m]` gives us a string that excludes the last `m` characters.

```
>>> s = 'homogeneous'
>>> s[:-1]
'homogeneou'
```

This gives a string object that contains the whole string except the last character. If you want a string object in which the last three characters are removed, you can write this `s[:-3]`.

```
>>> s[:-3]
'homogene'
```

Now, let us write a slice with a negative number as the first index.

```
>>> s[-5:]
'neous'
```

The slice `s[-5:]` starts at index `-5` and goes up to the last index, so it gives you the last 5 characters of the string. Similarly, the slice `s[-3:]` will give you the last 3 characters of the string.

When both the indexes are equal, we get an empty string.

```
>>> s[3:3]
''
```

We have seen that if we index a string and give an invalid index inside square brackets, an `IndexError` occurs. Let us see what happens if we provide a bad index in slicing.

```
>>> s[2:100]
'mogeneous'
```

The end index is greater than the size of the string, but we did not get any `IndexError`. We got a slice from index 2 to the end of the string. So, if the index is greater than or equal to `n` (length of the string), it means the end of the list. Similarly, if the first index is less than or equal to `-n`, it means the start of the string.

```
>>> s[-50:6]
'homoge'
```

Here, the first index is assumed to be at the start of the string. You can see that slicing is more forgiving than indexing. While indexing, if you give such bad indexes, then you will get an error.

```
>>> s[100]
IndexError: string index out of range
```

While slicing, you can also use a third integer inside the square brackets, which is the stride or step of the slice.

`s[i:j:k]` Part of the string `s` from index `i` to index `j-1`, with a step of `k`

`s[3:10:2]` Part of the string containing characters at indexes 3,5,7,9

`s[3:18:3]` Part of the string containing characters at indexes 3,6,9,12,15

`s[i:j:1]` Equivalent to `s[i:j]`

`s[6:1:-1]` Part of the string containing characters at indexes 6,5,4,3,2

`s[20:5:-2]` Part of the string containing characters at indexes
20,18,16,14,12,10,8,6

`s[::-1]` String in reverse order

The slice `s[i:j:k]` will extract characters from index `i` to index `j-1`, with each subsequent index incremented by `k`. When the step is omitted, it is assumed to be 1, so `s[i:j:1]` is equivalent to `s[i:j]`. In the previous examples that we had written, it was assumed to be 1. We can give negative steps also. In the slice `s[6:1:-1]` we start at 6 and add -1 each time, so we get indexes 6,5,4,3,2. Thus, the effect of using a negative slice is that we get the items in reverse order. The slice `s[::-1]` will give the whole string in reverse order. Here are some examples:

```
>>> s = 'Today is the day.'  
>>> s[3:13:2]  
'a ste'
```

Each alternate character of the string from index 3 to index 12 is displayed.

```
>>> s[::2]  
'Tdyi h a.'
```

Each alternate character of the whole string is displayed.

```
>>> s[::3]  
'Tait y'
```

The whole string is displayed with a step of three characters.

```
>>> s[::-1]  
' .yad eht si yadoT'
```

This gives the reverse of the whole string.

3.4 String Concatenation and Repetition

We know that when the operators `+` and `*` are used on numeric types, they add and multiply numbers. These operators can also be used on strings, but they are interpreted differently. The operator `+` performs string concatenation, and the operator `*` performs string repetition.

String literals or string variables can be combined by using the + operator.

```
>>> 'ab' + 'cd'
'abcd'
>>> name = 'Dev'
>>> 'Hello' + name
'HelloDev'
```

In the first example, we have combined two string literals. In the second one, we have combined a string literal with a string variable. In both these cases, a new string object is created, which is displayed at the prompt. In the second example, no space is added between the two words. If you want a space, you must add it explicitly.

```
>>> 'Hello' + ' ' + name
'Hello Dev'
```

The new string object returned after concatenation can be assigned to a name.

```
>>> s = 'Hello' + ' ' + name
>>> s
'Hello Dev'
>>> name = name + 'raj'
>>> name
'Devraj'
```

The asterisk symbol, when used with a string and integer, acts as a repetition operator. We can use the repetition operator to repeat a string.

```
>>> name = 'Dev'
>>> name * 3
'DevDevDev'
```

The expression `name * 3` returns a string object that contains the characters of the string `name` repeated three times. The integer denotes the number of times the string is repeated. You can think of it as an abbreviation for `n times concatenation`. `name * 3` is same as `name + name +`

name. The expression `3 * name` also has the same effect but is less intuitive.

```
>>> 'Hello ' * 5
'Hello Hello Hello Hello Hello '
>>> print('-' * 40)
```

```
-----
>>> s = 'Hee..'
>>> s = s * 3
>>> s
'Hee..Hee..Hee..'
```

In the statement `s = s * 3`, we assign the string object returned by the expression `s * 3` to the variable `s`.

Augmented assignment syntax can be used for both concatenation and repetition operators.

```
>>> s = 'butter '
>>> s += 'scotch '
>>> s
'butter scotch '
>>> s *= 3
>>> s
'butter scotch butter scotch butter scotch '
```

`s += 'scotch'` is equivalent to `s = s + 'scotch'` and the `s *= 3` is equivalent to `s = s * 3`

The augmented assignment does not make any changes to the original object. It reassigns the variable name to a new object.

Here are some more examples:

```
>>> s1 = 'Good Morning !'
>>> s2 = 'Bye Bye See you'
```

We have these two strings, and we must make a string by concatenating the first four characters of the first string and the first three characters of the second string. We can do this by combining the slices of the two strings.

```
>>> s3 = s1[:4] + s2[:3]
>>> s3
'GoodBye'
```

This slice `s1[:4]` gives a string object that contains the first four characters of the string `s1`, and the slice `s2[:3]` gives a new object that contains the first three characters of the string `s2`. When these objects are combined using the `+` operator, we get a new string object assigned to the name `s3`.

Now, we want to make a new string from the string `s1`, such that the first four characters are repeated three times, and the last character is repeated five times.

```
>>> s4 = s1[:4] * 3 + s1[4:-1] + s1[-1] * 5
>>> s4
'GoodGoodGood Morning !!!!!'
```

If we assign the result to the name `s1`, we get the effect of changing the string `s1`.

```
>>> s1 = s1[:4] * 3 + s1[4:-1] + s1[-1] * 5
>>> s1
'GoodGoodGood Morning !!!!!'
```

String literals can also be combined by writing them one after the other.

```
>>> 'abc' 'def' 'hij'
'abcdefhij'
```

Hence, adjacent string literals are concatenated. This feature is applicable only for literals. You cannot join string variables or expressions by using this feature. It is useful when you want to break long string literals.

3.5 Checking membership

The `in` and `not in` operators can be used to test for the existence of a character or substring inside a string. The `in` operator returns `True` if a character or substring exists in the given string; otherwise, it returns `False`. The `not in` operator returns `True` if a character or substring is not present in the string.

```
>>> s = 'good morning !'
>>> 'ing' in s
True
>>> '?' in s
False
>>> 'good morning !' in s
True
>>> 'Good' in s
False
>>> 'you' not in s
True
>>> 'morning' not in s
False
```

3.6 Adding whitespace to strings

You can add whitespace to your string to organize and present it in a readable way. Whitespace in programming includes tabs, newlines, and spaces. The character combination `'\n'` adds a newline, and the combination `'\t'` adds a tab to your string.

```
>>> print('Sun\tMon\tTue')
Sun      Mon      Tue
>>> print('Sun\nMon\nTue\n')
Sun
Mon
```

```
Tue
>>> print('Days : \n\tSun\n\tMon\n\tTue\n')
Days :
    Sun
    Mon
    Tue
```

A single `print` call gives multiple lines of output due to the inclusion of `'\n'` character. This way, we can generate multiple lines of output with only a few lines of code. However, some programmers prefer writing separate `print` calls as the `'\n'` embedded inside a string is difficult to read.

3.7 Creating multiline strings

A string literal enclosed in single or double quotes cannot span more than one line of a program. Such a string should be contained in a single line only. The ending quote should appear on the same line as the starting quote. You will get a syntax error if you try to write a multiline string inside single or double quotes.

```
>>> s = 'Let us get up and get going,
... With a strong heart for whatever may come our
way.
... Keep working, keep trying,
... Learn to work hard and be patient each day.'
...
SyntaxError: unterminated string literal (detected
at line 1)
```

If you want a string literal that spans across multiple physical lines, you can use the continuation character.

```
>>> s = 'Let us get up and get going,\
... With a strong heart for whatever may come our
way.\
```

```
... Keep working, keep trying,\n... Learn to work hard and be patient each day.'\n>>> s\n'Let us get up and get going,With a strong heart\nfor whatever may come our way.Keep working, keep\ntrying,Learn to work hard and be patient each day.'\n>>> print(s)\nLet us get up and get going,With a strong heart for\nwhatever may come our way.Keep working, keep\ntrying,Learn to work hard and be patient each day.
```

The backslash indicates that the string is continued on the next line. Now, we could define the string literal on multiple lines, but when this string is printed, we do not get the literal printed on different lines. To achieve this, we can include newline characters in between the literal. We already know that `'\\n'` is the newline control character used to begin a new line on a screen, so we can use it inside the string.

```
>>> s = 'Let us get up and get going,\\n\\n\n... With a strong heart for whatever may come our\nway.\\n\\n\n... Keep working, keep trying,\\n\\n\n... Learn to work hard and be patient each day.'
```

The `'\\n'` adds a newline character, and the backslash indicates that the string is continued on the next line.

```
>>> print(s)\nLet us get up and get going,\nWith a strong heart for whatever may come our way.\nKeep working, keep trying,\nLearn to work hard and be patient each day.
```

A better and more common way is to use triple-quoted strings. If you put a string literal inside triple quotes, it spans across multiple lines naturally. The triple quotes can consist of three consecutive single quotes(`'''abc'''`) or three consecutive double quotes(`"""abc"""`).


```
s = '''Let us get up and get going,  
With a strong heart for whatever may come our way.  
Keep working, keep trying,  
Learn to work hard and be patient each day.'''
```

If your literal starts with a triple quote, you can keep adding text to it on multiple lines. The literal ends with terminating triple quotes.

```
>>> print(s)  
Let us get up and get going,  
With a strong heart for whatever may come our way.  
Keep working, keep trying,  
Learn to work hard and be patient each day.
```

The newline characters are naturally embedded in a string delimited by triple quotes. Any spaces at the beginning of the line will also be included in the string. If we display the string on the prompt instead of printing it using the `print` function, we will see the newline characters.

```
>>> s  
'Let us get up and get going,\nWith a strong heart  
for whatever may come our way.\nKeep working, keep  
trying,\nLearn to work hard and be patient each  
day.'
```

When you delimit a string literal inside triple quotes, Python adds a newline character at the end of each line. When you print such a string with the `print` function, you can see the original lines because each newline character is interpreted.

When we used backslash to join the lines, then the newline was not added automatically. If you want to prevent some newlines in a triple-quoted string, add a backslash at the end of those particular lines.

```
>>> s = '''Let us get up and get going,  
... With a strong heart for whatever may come our  
way.\n... Keep working, keep trying,
```

```
... Learn to work hard and be patient each day.'''
```

```
>>> print(s)
```

```
Let us get up and get going,
```

```
With a strong heart for whatever may come our  
way.Keep working, keep trying,
```

```
Learn to work hard and be patient each day.
```

Python supports triple-quoted strings so that we can write multiline strings. Using triple quotes improves the readability of long multiline strings in the source code. Generally, these are used in docstrings, that we will discuss later. Another advantage of triple-quoted strings is that we can use them to write string literals that have to include both single and double quotes.

```
>>> print('''My height is 5'3" ''')
```

```
My height is 5'3"
```

We have seen that in Python, adjacent string literals are concatenated. If we place more than one string literal adjacent to each other on a line (with optional whitespace in between), then they will be automatically concatenated.

```
>>>'abc' 'def' 'hij'
```

```
'abcdefghij'
```

If you write the string literals on separate lines and enclose them in parentheses, even then, they are considered adjacent and will be concatenated.

```
>>> s = ('Let us get up and get going,'
```

```
... 'With a strong heart for whatever may come our  
way.'
```

```
... 'Keep working, keep trying,'
```

```
... 'Learn to work hard and be patient each day. ')
```

```
>>> print(s)
```

```
Let us get up and get going,With a strong heart for  
whatever may come our way.Keep working, keep  
trying,Learn to work hard and be patient each day.
```

This can be another way of writing strings that span multiple lines. This approach does not add any newline characters in the string. If you need newlines, you need to add the newline character explicitly in the literals.

This approach can be helpful if you need to add comments to separate lines of the string.

```
>>> s = ('Let us get up and get going, '
... 'With a strong heart for whatever may come our
way.' # prepared for anything
... 'Keep working, keep trying, '
... 'Learn to work hard and be patient each day. ')
# patience is the key
>>> print(s)
```

```
Let us get up and get going,With a strong heart for
whatever may come our way.Keep working, keep
trying,Learn to work hard and be patient each day.
```

The comments are not included in the string. We do not see them when we print the string. In triple-quoted strings, if you try to add comments like this, those comments will be added to the string.

In the previous chapter, we had seen that for adding a multiline comment, we had to precede each line with a # sign. We can also use single triple quotes or double triple quotes to insert multiline comments in our code.

```
# This is a multiline comment
# It explains the code
# It has no effect on the code
''' This is also a multiline comment
    It explains the code
    It has no effect on the code
'''
```

The triple-quoted string is written all by itself. We are not printing it or assigning it to any variable. It is an unused string, so we can use it as a comment. However, this style of writing comments is not recommended, and

in most places, you will find comments that use the `#` sign. The triple-quoted strings are used for docstrings, which we will discuss later.

3.8 String methods

The `str` type supports many methods that can be dot suffixed to the name of the string. We have seen that `str` is an immutable type, so it does not provide any methods that change the original string object. All methods that seem to make changes in the string are designed such that they return a new modified string object. They do not touch the original string object because they are not able to, as objects of type `str` are immutable.

Let us understand this with the help of an example. The method `upper()` is used to change the letters in a string to uppercase. We have a string variable `s`.

```
>>> s = 'Hello'
```

When we call the method `upper` on the variable `s`, it returns a new object that contains all letters of this string in uppercase.

```
>>> s.upper()  
'HELLO'
```

The original object to which `s` was referring remains unchanged. We can make another variable refer to the object returned by `upper`.

```
>>> s1 = s.upper()  
>>> s1  
'HELLO'
```

Now, `s1` refers to the object returned by the method `upper`. If we want to make `s` refer to this new object, we can write `s = s.upper()`, then `s` will refer to this new object.

So, you cannot change the string object using any method, but you can assign the new object returned by the method to the string variable referring to the original string object. `str` type has lots of methods; we will explore some of the common ones here. You can try them on the interactive prompt. To get an up-to-date list of methods, you can call `dir(str)` or

`help(str)` on the interactive prompt. To get the description of a particular method, type `help(str.methodname)` on the prompt.

3.9 Case-changing methods

The following five case-changing methods can be used to perform case conversions in strings. All of them return a new object, which is a copy of string `s` with some changes in the case of the contained letters.

<code>s.lower()</code>	Returns a copy of <code>s</code> , in which each letter is converted to lowercase
<code>s.upper()</code>	Returns a copy of <code>s</code> , in which each letter is converted to uppercase
<code>s.swapcase()</code>	Returns a copy of <code>s</code> , in which each lowercase letter is converted to uppercase and vice versa
<code>s.capitalize()</code>	Returns a copy of <code>s</code> , in which the first letter of the string is capitalized, and the rest of the letters are changed to lowercase
<code>s.title()</code>	Returns a copy of <code>s</code> , in which the first letter of each word is capitalized, and the rest of the letters are changed to lowercase

Table 3.2: Case-changing methods

Let us try some of them at the prompt.

```
>>> s = 'Life is a journey, not a race'
>>> s.lower()
'life is a journey, not a race'
>>> s
'Life is a journey, not a race'
```

We must assign the returned object to the original variable name to see the required change.

```
>>> s = s.title()
>>> s
'Life Is A Journey, Not A Race'
```

Similarly, while using other methods, if you want to see a change in your string, you need to reassign it.

When checking for membership or comparing strings, you can ignore the case by using the `upper` or `lower` methods.

```

>>> 'out' in 'Output'.lower()
True
>>> s = 'telephone'
>>> s[0].upper() in 'AEIOU'
False
>>> response = input('Enter yes or no : ')
Enter yes or no : Yes
>>> response.lower() == 'yes'
True

```

3.10 Character classification methods

The methods in this group check the contents of the string, and they return either True or False. All of them start with 'is', and their names are self-explanatory.

<code>s.isalnum()</code>	Returns True if all characters in <code>s</code> are alphanumeric
<code>s.isalpha()</code>	Returns True if all characters in <code>s</code> are alphabetic
<code>s.isdecimal()</code>	Returns True if there are only decimal characters in <code>s</code>
<code>s.isdigit()</code>	Returns True if all characters in <code>s</code> are digits
<code>s.isidentifier()</code>	Returns True if <code>s</code> is a valid identifier
<code>s.islower()</code>	Returns True if all letters in <code>s</code> are lowercase
<code>s.isupper()</code>	Returns True if all letters in <code>s</code> are uppercase
<code>s.istitle()</code>	Returns True if <code>s</code> is a title cased string
<code>s.isnumeric()</code>	Returns True if all characters in <code>s</code> are numeric
<code>s.isprintable()</code>	Returns True if all characters in <code>s</code> are printable
<code>s.isspace()</code>	Returns True if all characters in <code>s</code> are whitespace

Table 3.3: Character classification methods

Here are some examples:

```

>>> s = 'Yes Sir'
>>> s.isalpha()

```

```
False
>>> s.isupper()
False
>>> s.istitle()
True
```

3.11 Aligning text within strings

The following three methods justify a string into a given field size, and by default, the padding is done with spaces.

<code>s.ljust(size)</code>	Returns the string left justified in a string of length <code>size</code>
<code>s.rjust(size)</code>	Returns the string right justified in a string of length <code>size</code>
<code>s.center(size)</code>	Returns the string centered in a string of length <code>size</code>

Table 3.4: Text alignment methods

The methods `ljust()`, `rjust()`, and `center()` left justify, right justify, or center a string, respectively, such that the string fits within the number of spaces provided by the argument `size`. Here, `size` is the total length of the string after padding. These methods can be used in printing tabular data.

```
>>> s = 'Be a voice, not an echo'
>>> s.ljust(40)
'Be a voice, not an echo                '
>>> s.rjust(40)
'                Be a voice, not an echo'
>>> s.center(40)
'        Be a voice, not an echo        '
```

If `size` is less than the length of the string, there is no change.

```
>>> s.center(4)
'Be a voice, not an echo'
```

You can specify a fill character for padding instead of default spaces.

```
>>> s.center(40, '*')
'*****Be a voice, not an echo*****'
```

The string is center justified in a field width of 40, and the padding is done with an asterisk symbol instead of spaces.

The interactive prompt displays the string object returned by a particular method. As we have seen before, if we want to see the change in the original string, we need to assign this string object to the original string variable.

3.12 Removing unwanted leading and trailing characters

The `str` type provides methods to remove leading and trailing whitespaces or other characters. These methods can be used to sanitize data for further processing. For example, data read from somewhere or input by the user can be cleaned before storing or processing.

<code>s.lstrip(chars)</code>	Returns a copy of the string with leading characters removed
<code>s.rstrip(chars)</code>	Returns a copy of the string with trailing characters removed
<code>s.strip(chars)</code>	Returns a copy of the string with both leading and trailing characters removed

Table 3.5: *Methods to remove leading and trailing characters*

`rstrip()` and `rstrip()` remove characters from the left and right sides of the string, respectively, while `strip()` removes characters from both the left and the right sides. The set of characters to be removed is specified as a string argument. All the characters present in the string argument will be removed from the left, right, or both sides of the string. Here are some examples:

```
>>> '!!!..Imagine .. believe .. achieve ..!!! ?'
'.rstrip('!?. ')
'!!!..Imagine .. believe .. achieve'
```

The argument string is `'!?. '` so all exclamation marks, question marks, full stops, and spaces are removed from the right of the string.


```
>>> '!!!..Imagine .. believe .. achieve ...!! ?  
' .lstrip('!?. ')  
'Imagine .. believe .. achieve ...!! ? '
```

Now we have called `lstrip`, so the characters contained in the argument string are removed from the left of the string.

```
>>> '!!!..Imagine .. believe .. achieve ...!! ?  
' .strip('!?. ')  
'Imagine .. believe .. achieve'
```

Now, the characters are removed from both the left and right of the string, as we have called `strip`.

If the argument is omitted or is `None`, the whitespace characters are removed.

```
>>> s = '    All is well    '  
>>> s.lstrip()  
'All is well    '  
>>> s.rstrip()  
'    All is well'  
>>> s.strip()  
'All is well'
```

These methods return a new object, allowing us to chain subsequent method calls. In the next example, we have called the method `upper` on the object returned by the method `strip()`.

```
>>> s.strip().upper()  
'ALL IS WELL'
```

As you know, to see these changes in `s`, you must reassign `s` to the new object.

The methods `removeprefix` and `removesuffix` can be used to remove a prefix or suffix from the string. If the prefix or suffix is not present then a copy of the original string is returned.

```
>>> 'PyTorch'.removeprefix('Py')
```

```

'Torch'
>>> 'Numpy'.removesuffix('Py')
'Numpy'
>>> 'Numpy'.removesuffix('py')
'Num'

```

3.13 Searching and replacing substrings

One of the essential programming tasks is to search your data for specific information. Python provides many useful methods for searching and replacing information in a string.

<code>s.find(substr)</code>	Returns index of the first occurrence of the given substring. If not found, returns -1
<code>s.index(substr)</code>	Returns index of the first occurrence of the given substring. If not found, raises <code>ValueError</code>
<code>s.rfind(substr)</code>	Returns index of the last occurrence of the given substring. If not found, returns -1
<code>s.rindex(substr)</code>	Returns index of the last occurrence of the given substring. If not found, raises <code>ValueError</code>

Table 3.6: *Methods to search a substring*

The methods `find` and `index` return the index of the first occurrence of the given substring. If not found, `find` returns -1 while `index` raises a `ValueError`. The methods `rfind` and `rindex` are the same as `find` and `index` except that they search through the string backward, i.e., from right to left, so they find the last occurrence of the substring. Here are some more methods:

<code>s.count(substr)</code>	Returns the number of occurrences of the specified substring in <code>s</code>
<code>s.startswith(substr)</code>	Returns <code>True</code> if <code>s</code> starts with the specified substring, <code>False</code> otherwise
<code>s.endswith(substr)</code>	Returns <code>True</code> if <code>s</code> ends with the specified substring, <code>False</code> otherwise
<code>s.replace(s1,s2)</code>	Returns a copy of the string with all occurrences of the first string replaced with the second string

Table 3.7: *String methods*

In all these methods, you can restrict the search by specifying, optional arguments `start` and `end`, as in a slice.

To substitute a substring with another we can use the method `replace`. It returns a copy of the string with all occurrences of the first string replaced with the second string. As usual, the original string remains unchanged, and a new string object is returned. You can restrict the number of replacements by providing a third argument. That argument represents the number of occurrences that have to be replaced. Let us use these methods to understand them better.

```
>>> s = '''Focus on present, not on past or future
... Focus on yourself, not on others
... Focus on the process, not on outcome
... Focus on what you can control, not on what you
cannot control'''
>>> s.find('Focus')
0
```

This call returns the index of the first occurrence of the substring `'Focus'` in the string `s`. We get 0 here because this substring is present in string `s` at the 0th index.

```
>>> s.rfind('Focus')
111
```

This method `rfind` returns the index of the last occurrence of the substring in the string.

```
>>> s.find('focus')
-1
```

We get -1 because `'focus'` with `f` in lowercase is not present in the string `s`.

The methods `index` and `rindex` are similar to `find` and `rfind`, but instead of returning -1, they raise a `ValueError` if the substring is not found. In the previous call, if we use the `index` method, then instead of -1, we get a `ValueError`.

```
>>> s.index('focus')
ValueError: substring not found
>>> s.count('on')
10
```

The substring 'on' comes 10 times in the string s.

```
>>> s.startswith('Focus')
True
>>> s.endswith('?')
False
```

In these methods that we have seen, we can give the start and end index, where the search will be performed.

```
>>> s.find('Focus', 20, 100)
40
```

The search is performed in the string portion from index 20 to index 99. These start and end indexes are interpreted as in slice notation. Similarly, we can use the start and end indexes in all the other methods of this category.

Now, suppose we have a string s.

```
>>> s = 'Dev; 22; male; graduate; Bareilly'
```

We want a string that contains everything after the first occurrence of semicolon. We can get it by using the index method in the slice notation.

```
>>> s[s.index(';'):]
'; 22; male; graduate; Bareilly'
```

s.index(';') gives us the index of the first occurrence of the semicolon, which is 3, and the expression s[s.index(';'):] gives us a slice from index 3 till the end. So, we get everything after the first occurrence of the semicolon. The semicolon itself is included. If we do not want it, we can specify s.index(';')+1 as the start index for the slice.

```
>>> s[s.index(';')+1:]
' 22; male; graduate; Bareilly'
```

Now, we assign this slice object to the name `s2`.

```
>>> s2 = s[s.index(';')+1:]
>>> s2
' 22; male; graduate; Bareilly'
```

So, `s2` is a string that contains everything after the first occurrence of the semicolon. Now, instead of `index`, let us write `rindex`.

```
>>> s2 = s[s.rindex(';')+1:]
>>> s2
' Bareilly'
```

Now we get a string that contains everything after the last occurrence of semicolon. Let us combine both `index` and `rindex` in this slice.

```
>>> s2 = s[s.index(';')+1: s.rindex(';')]
>>> s2
' 22; male; graduate'
```

This gives us everything between the first and last occurrence of the semicolon. We could have also used the `find()` method here, but it is better to use the `index()` method in these types of cases, as `find` returns `-1` if the substring is not found, and `-1` is a valid index value in Python. We might get incorrect results if we use `find()`. Let us understand this with the help of an example. Suppose we want everything from the beginning of the string `s` to the first occurrence of the substring `xy`.

```
>>> s2 = s[: s.index('xy')]
ValueError: substring not found
```

The substring `'xy'` is not present in `s`, so we get this error. Now, instead of `index`, let us use `find` and see.

```
>>> s2 = s[: s.find('xy')]
```

This does not give any error. Let us see what is `s2`.

```
>>> s2
'Dev; 22; male; graduate; Bareill'
```

The `find` method returned `-1` since the substring was not present. So, this slice represents the whole string from starting to index `-2`.

Now, let us try the `replace` method. Again, we take this multiline string `s`.

```
>>> s = '''Focus on present, not on past or future
... Focus on yourself, not on others
... Focus on the process, not on outcome
... Focus on what you can control, not on what you
can't control'''
```

```
>>> s2 = s.replace('Focus', 'Concentrate')
```

```
>>> print(s2)
```

```
Concentrate on present, not on past or future
Concentrate on yourself, not on others
Concentrate on the process, not on outcome
Concentrate on what you can control, not on what
you can't control
```

All the occurrences of `'Focus'` are replaced with `'Concentrate'`.

```
>>> s2 = s.replace('Focus', 'Concentrate', 3)
```

```
>>> print(s2)
```

```
Concentrate on present, not on past or future
Concentrate on yourself, not on others
Concentrate on the process, not on outcome
Focus on what you can control, not on what you
can't control
```

Now, only the first three occurrences are replaced. By replacing it with an empty string, we can delete characters from the string.

```
>>> s2 = s.replace('not', '')
```

```
>>> print(s2)
```

```
Focus on present,  on past or future
Focus on yourself,  on others
Focus on the process,  on outcome
```

Focus on what you can control, on what you can't control

All occurrences of substring 'not' were removed. As a result of removal, we get double spaces in many places. We want only one space in those places. For this, we can replace double spaces with a single space by making one more call to `replace` method.

```
>>> s2 = s.replace('not', '').replace('  ', ' ')
>>> print(s2)
```

Focus on present, on past or future

Focus on yourself, on others

Focus on the process, on outcome

Focus on what you can control, on what you can't control

This chained call works because the `replace` method returns a string object.

3.14 Chaining method calls

Most string methods return a string object, so you can apply multiple methods to a string to get the desired result. We saw this while using the `rstrip` method and the `replace` method. Here is one more example:

```
>>> s = '  hello      '
>>> s = s.strip().upper().center(20, '*')
>>> s
'*****HELLO*****'
```

The methods are executed from left to right, one at a time. In this example, the method `strip` is called on the string `s`, then the method `upper` is called on the string returned by `strip` and the method `center` is called on the string returned by the method `upper`. The string object returned by `center` is assigned to `s`. The order of the methods matter; the output might change if the order is changed.

```
>>> s = '  hello      '
```

```
>>> s = s.center(20, '*').upper().strip()
>>> s
'*** HELLO      ***'
```

3.15 String comparison

The operators `is` and `is not` are used to compare the identity of strings (and other objects). They check whether the two strings occupy the same space in memory.

The comparison operators `==`, `!=`, `<`, `>`, `<=` and `>=` are used to compare strings. As usual, they return a Boolean value `True` or `False`. Two strings are considered equal if their content is exactly the same.

```
>>> s1 = 'Python'
>>> s2 = 'Python'
>>> s1 == s2
True
>>> s1 != s2
False
```

The comparisons performed by the comparison operators are case-sensitive. For example, `'Python'` and `'python'` will not be considered equal. To ignore case and perform case-insensitive comparisons, you can convert both strings to either lowercase or uppercase by using the `upper` and `lower` methods, as we discussed in [section 3.9](#).

```
>>> s1 = 'Python'
>>> s2 = 'python'
>>> s1 == s2
False
>>> s1.lower() == s2.lower()
True
>>> s1.upper() == s2.upper()
True
```


The `casefold()` method can also be used for caseless matching of the strings, as it returns a casefolded copy of the string. This method will work properly even if your string contains Unicode characters.

```
>>> s1.casefold() == s2.casefold()
True
```

The comparison operators compare the individual characters according to the ASCII or Unicode value (code point). Lowercase letters are considered larger than the corresponding uppercase letters as the lowercase letters have a bigger code point than the uppercase ones.

```
>>> ord('P')
80
>>> ord('p')
112
>>> 'Python' < 'python'
True
```

When the string contains all lowercase or all uppercase letters, the comparison is done in regular alphabetic order as in a dictionary.

3.16 String conversions

A type can be converted to another type using the type name as a function if the conversion is supported. Suppose you have a string that represents a number.

```
>>> s = '23'
```

The type of this variable `s` is `str`, so you cannot perform any arithmetic operation supported by `int` type.

```
>>> s + 1
TypeError: can only concatenate str (not "int") to
str
```

However, you can perform operations by converting `s` to `int` or `float`.

```
>>> x = int(s)
```

```
>>> x + 1
24
>>> float(s) / 2
11.5
>>> s = 'UP05788'
>>> n = int(s[2:])
>>> n + 1
5789
```

You can similarly convert strings to other types like `list` or `set`. We will see these types in the coming chapters. The conversion to `int` is a bit different from others as it can take a second argument also (we have discussed this in the previous chapter).

The type name `str` can be used as a function to create string objects. If the argument you send is a string, the function `str` returns a new string object that is a copy of the string. If the argument is a non-string type, it returns a string object that represents the string form of the argument, provided the argument is convertible to a string.

If we try to concatenate a string with a number, a `TypeError` is raised. The number must be converted to a string by using the `str` function.

```
>>> s1 = 'UP05'
>>> n = 2456
>>> s1 + n
TypeError: can only concatenate str (not "int") to str
>>> s1 + str(n)
'UP052456'
```

The functions `bin`, `oct`, and `hex` can also convert a number to a string in an appropriate base.

```
>>> bin(100)
'0b1100100'
>>> oct(100)
```

```
'0o144'  
>>> hex(100)  
'0x64'
```

3.17 Escape Sequences

Inside a string, the backslash (\) is considered an escape character. It is used to indicate that the following character has special meaning, so it should not be treated in the regular way. We have already seen how to include a newline and a tab using the character combinations '\n' and '\t'. These character combinations are examples of escape sequences. The combination '\n' or '\t' is considered a single character known as an escape character. Here is a list of more escape sequences:

\<newline>	Backslash and newline ignored
\'	Single Quote
\"	Double Quote
\\	Backslash character(\)
\n	New Line
\t	Horizontal Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\a	Bell
\ooo	Character with octal value ooo
\xhh	Character with hex value hh
\N{name}	Character named name in the Unicode database
\uxxxx	Unicode character with a 16-bit hex value xxxx
\Uxxxxxxxx	Unicode character with a 32-bit hex value xxxxxxxx

Table 3.8: *Escape sequences*

Escape sequences are special character representations that are represented by a combination of characters where the first character is a backslash, followed by one or more characters. When they appear inside a string, they

are replaced by the single character that they represent. Escape sequences let us embed special non-printing characters (that cannot be typed on a keyboard) in a string. They also resolve ambiguity, such as printing a single quote inside a single quoted string.

Let us use these escape sequences in our strings. We know that `'\n'` represents a newline character, and when it is written inside a string, it will start a new line on the screen.

```
>>> print('How\nare\nyou')
```

```
How
are
you
```

Here we are printing a string that contains the escape sequence `'\n'`. We can see that each `'\n'` is replaced with a newline character; it is printed in the form of a newline. So, you can print the text inside a single string in multiple lines. Let us see the length of this string.

```
>>> len('How\nare\nyou')
```

```
11
```

The escape sequence `'\n'` is counted as just one character, so we have $3+1+3+1+3$, which is 11. If we use the escape sequence `'\t'`, then it is replaced by a tab character which provides space between 2 values.

```
>>> print('How\tare\nyou')
```

```
How      are
you
```

An escape sequence is called so as it escapes the usual meaning of a letter or character (like `n` in `'\n'`) and gives it a whole new meaning.

When Python does not recognize the character after a backslash as an escape code, it just keeps the backslash literally in the string. For example:

```
>>> print('H\el\lo')
```

```
H\el\lo
```

Here, `e` and `l` are not escape codes, so the backslash is literally included in the string. This means that the backslash is included as itself in the string and

is not treated specially. The replacement is done only when the backslash is followed by a valid escape code.

Now, suppose we want to print or use a string that contains some Windows Path.

```
>>> print('C:\textfiles\newFile')
C:      extfiles
ewFile
```

Both '`\t`' and backslash '`\n`' are recognized as escape sequences. So, they are replaced by their respective characters. However, we do not want this replacement to be done in this case. We want to print the backslash literally, even when followed by an escape code. To print a literal backslash character, you must use double backslashes.

```
>>> print('C:\\textfiles\\newFile')
C:\textfiles\newFile
```

Now, the backslashes are printed literally. We could also use raw strings, as we will discuss shortly.

If we try to print a string containing a single quote and enclosed inside single quotes, we will get a syntax error.

```
>>> print('Don't run')
SyntaxError: unterminated string literal
```

One solution to this problem is to enclose the whole string inside double quotes instead of single quotes. Another solution is to use an escape sequence.

```
>>> print('Don\'t run')
Don't run
```

Here, the interpreter sees that the single quote is preceded by a backslash, so it will print a single quote; it will not use this single quote to end the string. This way, you can insert a single quote inside a string enclosed in single quotes, and similarly, you can insert a double quote inside a double-quoted string.

3.18 Raw string literals

If you want to turn off the backslash escape mechanism in a string, you can precede the string literal with the letter `r`. These are called raw strings. They treat backslash as a literal character and not as an escape character. Every character inside a raw string stays the way it is written inside the string. Here are some examples:

```
>>> s = r'hello\n'
>>> print(s)
hello\n
```

Raw strings can be helpful when you have strings that contain many backslashes like Windows path and regular expressions.

```
>>> print(r'C:\Deepali\newFiles')
C:\Deepali\newFiles
```

Here, `'\n'` is not considered an escape sequence. Since the string is preceded by `r`, it is a raw string. The interpreter considers the backslash as a normal character of the string and not as a start of an escape sequence. If we remove `r`, then `'\n'` is considered an escape sequence.

```
>>> print('C:\Deepali\newFiles')
C:\Deepali
ewFiles
```

3.19 String formatting

We have the following 3 variables of type `str`, `int`, and `float`.

```
>>> name = 'Raj'
>>> age = 23
>>> wt = 43.567
```

We know that we can create a string by concatenating strings literal and variables.

```
>>> s = 'My name is ' + name + ', I am ' + str(age)
+ ' years old and my weight is ' + str(wt) + ' kg'
>>> s
'My name is Raj, I am 23 years old and my weight is
43.567 kg'
```

We had to use the conversion functions to convert non-string variables to string, and using the + operator many times was not very readable. Another way is to use the `print` function, in which you can send the strings and variables separated by commas.

```
>>> print('My name is', name, ', I am', age, 'years
old and my weight is', wt, 'kg')
My name is Raj , I am 23 years old and my weight is
43.567 kg
```

Here, we have all the string literals and variables separated by commas. Till now, we have been using these simple approaches for displaying our data, but these approaches were not very readable. Python has different formatting styles that we can use to do more value formatting and display the output in an organized way.

We need to format strings to present data in a better way. This is required when data is to be displayed to the program's user in a readable and understandable manner. In the following image, you can clearly see the difference between the data displayed without any formatting and after formatting.

Name	City	Percentage	Name	City	Percentage
Amit	Bangalore	90.0	Amit	Bangalore	90.00
Vineeta	Delhi	93.33333333333333	Vineeta	Delhi	93.33
Dev	Agra	88.0	Dev	Agra	88.00
Rajneesh	Jaipur	88.66666666666667	Rajneesh	Jaipur	88.67

Figure 3.4: Unformatted and formatted data

String formatting also allows us to interpolate values of variables into strings, which means that we can insert values inside strings using different

formats. You need to format strings for better display on the screen. String formatting is also required when you need to substitute variables.

There are three ways of formatting strings in Python. There is no need to learn all of them, but knowing them is good as you might encounter them in someone else's code. The first is the old-style formatting, which uses the % operator like C language. This style is still supported but is deprecated.

```
>>> name = 'Raj'
>>> age = 23
>>> wt = 47.5

>>> s = 'My name is %s, I am %d years old and my
weight is %f kg' % (name, age, wt)
>>> s
'My name is Raj, I am 23 years old and my weight is
47.500000 kg'
```

In Python 3, a newer style was introduced, which used the `format` method of string class. This was introduced in Python 3 but was backported to Python 2.6.

```
>>> name = 'Raj'
>>> age = 23
>>> wt = 47.5

>>> s = 'My name is {}, I am {} years old and my
weight is {} kg'.format(name, age, wt)
>>> print(s)
My name is Raj, I am 23 years old and my weight is
47.5 kg
```

The curly braces act as placeholders for the data, and the values are sent as arguments to the `format` method.

In Python 3.6, a new formatting approach was introduced that used formatted string literals, also called f-strings.

```
>>> name = 'Raj'
>>> age = 23
```



```
>>> wt = 47.5
>>> s = f'My name is {name}, I am {age} years old
and my weight is {wt} kg'
>>> print(s)
My name is Raj, I am 23 years old and my weight is
47.5 kg
```

Using these f-string literals, you can embed Python expressions inside a string literal using curly braces. They are called f-strings because you get a formatted string literal by prefixing a string with the letter f.

So, when we have a string literal prefixed with f, any variable inside curly braces is substituted with its value. You can see that this style is much clearer than the previous two. It is the simplest one because you can directly insert the names inside the string literal. In this book, we will mostly use the f-string formatting. You might encounter the `format` method style in some other code, so it is discussed in the next section. In the rest of this section, we will discuss f-strings.

Using f-strings, you can simply write your string; whenever you want to substitute the value of a variable, just put it inside curly braces. You can even write Python expressions inside curly braces or call functions and methods directly.

```
>>> name = 'Raj'
>>> age = 23
>>> wt = 47.567
>>> f'After 10 years {name.upper()} will be {age +
10} years old'
'After 10 years RAJ will be 33 years old'
```

We have called the `str` method `upper` and used the expression `age + 10`. Curly braces are used to hold the variables or expressions; they are not displayed. If you want to print left and right curly braces, double them up.

```
>>> f'He is {{ {name}, {age} }}'
'He is { Raj, 23 }'
```

The double curly braces are displayed as a single curly brace.

You can specify a field width where the given value will be displayed.

```
>>> f'His name is {name:8} and he is {age:6} years old'
```

```
'His name is Raj          and he is          23 years old'
```

The numbers 8 and 6 represent the field width, so the variable name is displayed in a width of 8, and age is displayed in a field width of 6. By default, the text is left-aligned, and numbers are right-aligned in their field. We can force left alignment by using less than sign <. Similarly, the right alignment can be forced using the greater than sign > and center alignment by caret ^ sign.

```
>>> f'His name is {name:>8} and he is {age:<6} years old'
```

```
'His name is          Raj and he is 23          years old'
```

Now name is left-aligned, and age is right-aligned.

```
>>> f'His name is {name:^8} and he is {age:^6} years old'
```

```
'His name is    Raj    and he is    23    years old'
```

Now, both name and age are center-aligned in their fields.

To print an integer in a fixed point format, write :f.

```
>>> f'Age is {age:f} and weight is {wt}'
```

```
'Age is 23.000000 and weight is 47.567'
```

The variable age is an integer, but since we have included :f, it is printed with a point. We can also control the number of digits that are displayed.

```
>>> f'Age is {age:.3f} and weight is {wt}'
```

```
'Age is 23.000 and weight is 47.567'
```

Now, only three decimal digits are displayed. We can also specify the width.

```
>>> f'Age is {age:<10.3f} and weight is {wt}'
```

```
'Age is 23.000          and weight is 47.567'
```

The number 10 is the field width, and the less than symbol is for left justification. Now, let us format the float value wt.

```
>>> f'Age is {age:<10.3f} and weight is {wt:.3f}'
'Age is 23.000      and weight is 47.6'
```

We have specified a colon, a dot, and the number 3. This number represents the total number of digits displayed. So, we can see that a total of three digits are displayed. Let us specify a width for it.

```
>>> f'Age is {age:<10.3f} and weight is {wt:8.3f}'
'Age is 23.000      and weight is      47.6'
```

Now, eight spaces are reserved to display this value. If you want to control the number of digits displayed after the decimal, use the letter `f`.

```
>>> f'Age is {age:<10.3f} and weight is {wt:8.3f}'
'Age is 23.000      and weight is    47.567'
```

The number 3 represents the number of digits displayed after the decimal.

By default, your output fields will be padded using spaces; if you want a character to be used for padding, you can place it just after the colon before the alignment specifier. The character is used to display data when the data is too small to fit in the assigned field width. It is called the fill character, which can be any character except `'{'` or `'}'`.

```
>>> f'My name is {name:*^10} and age is {age:->12}'
'My name is ***Raj*** and age is -----23'
```

The variable `name` is center-aligned in a field width of 10, while the asterisk is a fill character. The variable `age` is right-aligned in a field width of 12, and the dash is a fill character. The fill character must be specified before the alignment specifier, and if you want to specify a fill character, it is necessary to specify an alignment specifier. We know that numbers are right-justified by default, but we have still specified the right alignment specifier because we wanted padding done by dashes instead of spaces.

Escape sequences are interpreted as usual inside f-strings also. If you want to suppress the escape mechanism, you can write raw f strings.

```
>>> print(fr'\name: {name}')
```

```
\name: Raj
```

This `\n` is not considered an escape sequence here.

We can write triple-quoted f-strings that span multiple lines.

```
>>> s = f'''My name is {name}, I am {age} years old
... and my weight is {wt} kg'''
>>> s
'My name is Raj, I am 23 years old \nand my weight
is 47.567 kg'
>>> print(s)
My name is Raj, I am 23 years old
and my weight is 47.567 kg
```

An integer can be displayed in hexadecimal, octal, or binary base.

```
>>> num = 1247
>>> f'{num:x} {num:o} {num:b}'
'4df 2337 10011011111'
```

We can use lowercase e or uppercase E to display a number in exponential notation.

```
>>> num1 = 0.00000082478
>>> num2 = 3345600000000
>>> f'{num1:e} {num2:e} {num1:E} {num2:E}'
'8.247800e-07 3.345600e+12 8.247800E-07
3.345600E+12'
```

If we have a big number and want to print the thousands separator, we can write a comma after the colon.

```
>>> f'{num2:,}'
'3,345,600,000,000'
```

Many times, in our programs, we need to display the value of variables and expressions with their names.

```
>>> name = 'Raj'
>>> age = 23
>>> print(f'name = {name}, age = {age}')
```

```

name = Raj, age = 23
>>> a = 14
>>> b = 12
>>> print(f'a + b = {a + b} , a - b = {a - b}')
a + b = 26, a - b = 2
>>> print(f'min(a,b) = {min(a,b)}, max(a,b) =
{max(a,b)}')
min(a,b) = 12, max(a,b) = 14

```

Instead of duplicating the name of the thing to be printed, we can specify it once with an equal to sign, inside the curly braces.

```

>>> print(f'{name = }, {age = }')
name = 'Raj', age = 23
>>> print(f'{a + b = }, {a - b = }')
a + b = 26, a - b = 2
>>> print(f'{min(a,b) = }, {max(a,b) = }')
min(a,b) = 12, max(a,b) = 14

```

3.20 String formatting using the format() method of string class

f-strings were introduced in Python 3.6. If you are using an older version, you have to use the `format` method to format strings.

```

>>> name = 'Raj'
>>> age = 23
>>> wt = 47.567
>>> s = 'My name is {}, I am {} years old and my
weight is {} kg'.format(name, age, wt)
>>> s
'My name is Raj, I am 23 years old and my weight is
47.567 kg'

```

When the curly braces are empty, the interpreter will substitute based on the order of arguments sent in the `format` method. In the above example, the first pair of curly braces are replaced with `name`, the second pair with `age`, and the third pair with `wt`.

We can use index numbers inside curly braces to decide what goes where while substituting values inside the string.

```
>>> s = 'My name is {0}, I am {1} years old and my  
weight is {2} kg'.format(name, age, wt)
```

```
>>> s
```

```
'My name is Raj, I am 23 years old and my weight is  
47.567 kg'
```

The value 0 refers to the first argument, 1 refers to the second argument, and 2 refers to the third argument. This way, you can change the order of the variables and use a data value even more than once.

```
>>> s = 'Age {1} years, Name {0}, weight {2} kg,  
bye from {0}'.format(name, age, wt)
```

```
>>> s
```

```
'Age 23 years, Name Raj, weight 47.567 kg, bye from  
Raj'
```

In addition to positional arguments, we can send keyword arguments also. These keyword arguments are called by their name.

```
>>> s = '{msg}, my name is {n}, I am {a} years  
old'.format(n=name, a=age, msg='Hello')
```

```
>>> s
```

```
'Hello, my name is Raj, I am 23 years old'
```

We can mix both positional and keyword arguments in the same string.

```
>>> s = '{msg}, I am {1} years old and my weight is  
{0} kg'.format(wt, age, msg='Hello')
```

```
>>> s
```

```
'Hello, I am 23 years old and my weight is 47.567  
kg'
```

We can use conversion codes `s`, `d`, or `f`; the code `s` to display the value as a string; `d` to display the values as a decimal integer (base 10), and `f` to display the value as a float with decimal places. When using `f` conversion for values, you can limit the number of digits displayed after the decimal point. This can be done by adding a dot followed by the number of digits after the decimal you want displayed.

```
>>> num1 = 123
>>> num2 = 345.43678
>>> print('number1 is {:.2f}'.format(num1))
number1 is 123.00
>>> print('number2 is {:.2f}'.format(num2))
number2 is 345.44
```

The float value will be rounded off if it has more decimal places than the number of places we want to display.

You can use `0` if you do not want any decimal places to be displayed.

```
>>> print('number2 is {:.0f}'.format(num2))
number2 is 345
```

You can specify a width in which a given value is displayed.

```
>>> name = 'Raj'
>>> age = 23
>>> print('My name is {:8} and I am {:6} years
old'.format(name,age))
My name is Raj          and I am      23 years old
```

By default, strings are left-justified in their width, and numbers are right-justified. To change the justification, you can use symbols `<`, `>` or `^`.

`<` for left justification

`>` for right justification

`^` for center justification

```
>>> print('My name is {:^8} and I am {:<6} years
old'.format(name, age))
```

My name is Raj and I am 23 years old

In the following example, a total of four digits of number are displayed in a width of 10.

```
>>> number = 78.386367
>>> print('number is {:10.4}'.format(number))
number is      78.39
```

In the next example, number is displayed in a width of 10 with four decimal places.

```
>>> print('number is {:10.4f}'.format(number))
number is      78.3864
```

If you want, you can specify a fill character for padding within the given field. By default, this padding is done with spaces. The alignment specifier should be provided to specify a padding character.

```
>>> print('My name is {:*^8} and age is
{:.>6}'.format(name, age))
My name is **Raj** and age is ....23
```

If there is a sign character, padding is done after that. A 0 preceding the width performs zero padding.

```
>>> print('My name is {:*^8} and age is
{:>06}'.format(name, age))
My name is **Raj** and age is 000023
```

You can provide a sign for numeric values.

+ Positive numbers have a + sign, and negative numbers have a - sign

- Negative numbers have a minus sign

<space> Positive numbers preceded with space and negative numbers with a - sign.

To specify an output type, you can use any of the following characters.

String - s

Integers - b for binary, d for decimal base 10 notation, x or X for hexadecimal, o for octal notation

Floating point - e or E for exponential notation, f for fixed point notation

```
>>> num = 246
>>> print('{:x}'.format(num))
f6
>>> print('{:X}'.format(num))
F6
>>> print('{:o}'.format(num))
366
>>> print('{:b}'.format(num))
11110110
>>> num1 = 0.000000000412
>>> num2 = 124300000000000
>>> print('{:e}'.format(num1))
4.120000e-10
>>> print('{:e}'.format(num2))
1.243000e+14
```

You can display your numeric data with a comma as the thousands separator.

```
>>> print('{:,}'.format(num2))
124,300,000,000,000
```

3.21 Representation of text - character encodings

For beginners in programming, this might seem like a complicated topic. If you have mastered the string processing and formatting concepts presented in the chapter, you can skip this part and move on to the next chapter without losing continuity. However, understanding encodings is important when sending or receiving data over the internet and dealing with text that includes

symbols, emojis, or different languages like Hindi, Russian, or Korean. This section will give you a basic understanding of encodings and how computers handle text as binary data. You can always come back to it later, but make sure to read it before you dive into the chapter on working with files. If you are curious about how computers handle text, you might find this section interesting. Before reading this section, it will be good to have a basic idea of the binary, decimal, and hexadecimal number systems.

Computers understand only 0s and 1s, so all forms of data, whether numbers, text, or pictures, are represented and stored in binary form inside the computer. Textual data is a sequence of characters like letters, digits, symbols, punctuation marks, etc. Humans understand these characters, but for a computer, each character is a number represented in binary form. So, to represent different characters on a computer, each one must be assigned a unique number. These numbers can be represented and stored as a sequence of bits (0s and 1s) inside the computer.

Since data has to be transferred between computers, it is essential that different computers use the same numeric codes for characters. This helps ensure that text displayed or processed on one system can be correctly understood and rendered on another. Thus, for effective communication between devices, there needs to be a uniform and universal way of encoding characters. To achieve this, the American Standard Code for Information Interchange (ASCII) was introduced in the 1960s. This standard defines numeric codes for 128 unique characters. It uses integers from 0 to 127 to represent different characters like uppercase letters, lowercase letters, digits, punctuation symbols, spacing characters, and other non-printing control characters. For example, the ASCII code for uppercase A is 65(hex 0x41), for lowercase a is 97(hex 0x61), and for digit 1 is 49(hex 0x31).

In ASCII, each character translates to an integer from 0 to 127. These 128 numbers can be represented by using 7 bits – 0000000(0) to 1111111(127). Thus, ASCII is a 7-bit encoding which can be implemented with only 7 bits. The basic storage unit of a computer is a byte, which is a group of 8 bits. With 8 bits, 256(2^8) unique characters can be represented (00000000(0) to 11111111(255)). The 8th bit is not utilized while using ASCII coding. If that bit is also used, 128 more characters could be represented. This resulted in different inconsistent encodings, which used the remaining 128 numbers (128 to 255) in different ways. Different countries and organizations started

using these spare 128 numbers to represent their own language symbols. ASCII was a universal standard, but these new encodings clashed and were not standardized.

Thus, there was a need for a universal coding standard that could accommodate characters from different scripts and languages used in the world. This led to the development of UNICODE in the 1990s. It is maintained by the Unicode Consortium, and its latest version, Unicode 15.1, contains a total of 149,813 characters, which include symbols from different languages of the world and even emojis. The Unicode specifications are continually updated to add new characters.

Each Unicode character is given a unique name and identification number called a code point. The Unicode code points are usually written in hexadecimal notation (4 to 6 hex characters) preceded by U+. For example, the code point for character A is written as U+0041, and its name is LATIN CAPITAL LETTER A; the code point for digit 1 is U+0031, and its name is DIGIT ONE. The hexadecimal number system is used for code points as it provides a compact representation of large numbers and a more human-friendly representation of binary data.

The Unicode standard contains many tables that list characters and their corresponding code points and names. The first 128 characters of the Unicode standard are the same as in the ASCII table, so ASCII is a subset of Unicode. You can get the Unicode symbols, their names, and code points from the Unicode website or the charmap utility in Windows. Note that the Unicode names are not case-sensitive.

Unicode is a text encoding standard like ASCII; both define unique numbers for different characters. They do not specify anything about the implementation, i.e., how these unique numbers should be stored in memory or transmitted over the network. Implementation of ASCII characters is simple as they are small in number (only 128), so each character can fit in a single byte. However, Unicode characters are large in number; thus, a single byte is not sufficient to represent each Unicode character. There could be different ways to represent a Unicode character as binary data. Thus, to represent Unicode characters as bit patterns, different Unicode encoding schemes are used. Unicode standard specifies the code points for various characters, while these schemes provide the format for representing a character in one or more bytes. These schemes specify how a Unicode

character will be represented in memory, files, or during data transmission. Some schemes are fixed length schemes while others are variable length. Fixed length schemes use the same number of bytes to represent each character, while variable length encodings represent different characters with different numbers of bytes.

Unicode standard is implemented by different encoding schemes like UTF-8, UTF-16, and UTF-32. The scheme UTF-32 is a fixed length encoding scheme that uses four bytes to represent each Unicode character. This encoding is not efficient in terms of space as characters that could be represented in one or two bytes also occupy four bytes. This encoding wastes a lot of space for representing common characters and thus is rarely used. UTF-16 and UTF-8 are variable-length encoding schemes, and from these, UTF-8 (Unicode Transformation Format -8) is more widely used. It is supported by most programming languages, websites, and operating systems.

UTF-8 is a variable-length encoding that uses one to four bytes to represent each Unicode character, depending on the character's code point value. The first 128 code points are represented with a single byte per character, which means that the ASCII characters are encoded in the same way in UTF-8, making it compatible with existing ASCII text. Since UTF-8 is backward compatible with ASCII, using UTF-8 will not break any software based on ASCII. Any valid UTF-8 text is also valid ASCII text.

For other non-ASCII characters, UTF-8 uses two, three, or four bytes per character. Thus, storing ASCII text is efficient since only one byte per character is taken. Less commonly used characters are represented using three or four bytes. UTF-8 is popular because it is compatible with ASCII and requires less space for English text and other Western languages.

The `str` type in Python is a sequence of Unicode characters so that we can include all characters listed in the Unicode standard in our Python strings. In the following strings, we have some Unicode characters that are not in ASCII. You can copy and paste them from somewhere if unavailable on your keyboard.

```
>>> s = 'Hello World ☺'
```

```
>>> c = 'Copyright © '
```

```
>>> greeting = '🙏 नमस्कार मेरा नाम दीपाली है 🙏'
>>> message = 'ನಾನು ಬೆಂಗಳೂರನ್ನು ಪ್ರೀತಿಸುತ್ತೇನೆ ❤️'
>>> bday_wish = '생일 축하해 신의 축복이 있기를 🎂🌸'
```

The Unicode characters can also be placed inside string literals with the help of escape sequences. We can insert a character by its code point by using the escape sequences `\xhh`, `\uxxxx`, `\Uxxxxxxxx`. Smaller numbers can be written using `\x`, and bigger ones using `\u` and `\U`. If you write smaller numbers with `\u` and `\U` you must do the left padding with zeros.

Characters can also be included by their Unicode name if we use the escape sequence `\N{name}`.

```
>>> '100\xA5'
'100¥'
>>> '\u2660\u2663\u2665\u2666'
'♠♣♥♦'
>>> '\N{Black Smiling Face} Hello World \N{White Smiling Face}'
'☺ Hello World ☺'
>>> '\U0001F929\U0001F607\U0001F60E\N{rolling on the floor laughing}'
'🤪😇😎🤪'
>>> '\xA9\u00A9\U000000A9\N{Copyright sign}'
'©©©©'
```

The module `unicodedata` contains a function named `name` that takes a Unicode character and returns its Unicode name in uppercase, and the function `lookup` that takes a case-insensitive name and returns a Unicode character.

```
>>> import unicodedata
>>> unicodedata.name('♠')
'BLACK SPADE SUIT'
>>> unicodedata.lookup('black spade suit')
```

'♠'

To see the names of all the characters used in a string, we can write the following loop. Do not worry about how the loop works. We will study the details of loops in the coming chapters.

```
>>> import unicodedata
>>> s = 'नमस्ते Hello 🙏'
>>> for i in range(len(s)):
...     print(unicodedata.name(s[i]))
DEVANAGARI LETTER NA
DEVANAGARI LETTER MA
DEVANAGARI LETTER SA
DEVANAGARI SIGN VIRAMA
DEVANAGARI LETTER TA
DEVANAGARI VOWEL SIGN E
SPACE
LATIN CAPITAL LETTER H
LATIN SMALL LETTER E
LATIN SMALL LETTER L
LATIN SMALL LETTER L
LATIN SMALL LETTER O
SPACE
PERSON WITH FOLDED HANDS
```

In Python 3, a string of type `str` is a sequence of Unicode characters. There is no encoding scheme associated with the string. When the string is stored in memory or disk or passed over a network, it is encoded using an encoding scheme. The interpreter will do most things for us, and we do not have to worry about encoding as long as we are doing regular string processing operations on our computer. When we exchange data with other sources, we need to be aware of the encoding schemes used by the source and our system.

Most of the Python implementations use the UTF-8 encoding scheme by default. So, the default encoding for Python source files (.py files) is UTF-8. You can use another encoding by inserting a comment of this form at the beginning of your .py file.

```
# -*- coding: encoding-name -*-  
# -*- coding: ascii -*-  
# -*- coding: windows-1252 -*-
```

We can use the built-in functions `ord` and `chr` to convert a character to a code point and vice versa. The function `ord` returns the Unicode code point for a one-character string, and the function `chr` returns a Unicode string of one character representing the Unicode code point provided to it. The `ord` function will raise a `TypeError` if you send a string of length longer than one.

```
>>> ord('A')  
65  
>>> ord('🙏')  
128591  
>>> hex(ord('🙏'))  
'0x1f64f'  
>>> chr(0x1f64f)  
'🙏'  
>>> chr(65)  
'A'
```

We know that the `str` type strings are immutable sequences of Unicode characters (or code points). Python also supports strings made up of raw bytes. The type for these strings is `bytes`, and they are immutable sequences of plain bytes or 8-bit values. 8-bit values can range from 0 to 255, so each element in a `bytes` string is an integer in the range 0 to 255. The `bytes` type is used to manipulate raw binary data. You can write a `bytes` literal like a `str` literal by enclosing it in single, double, or triple quotes, but with the letter `b` prefixed before the opening quote.

```
>>> y = b'\x44\x35\xC8'
```

```
>>> type(y)
<class 'bytes'>
```

`b'\x44\x35\xc8'` is a `bytes` literal that contains three bytes that we have specified with `\x` escape sequence in hexadecimal notation.

```
>>> y
b'D5\xc8'
```

When a `bytes` value is displayed, ASCII printable characters and escape sequences like `\n`, `\t` are printed while other bytes are shown with hexadecimal escape sequence `\x`. This is why, while displaying the above `bytes` string, the ASCII-compatible characters `D` and `5` are represented as characters while the last byte is displayed with an escape sequence. This is the reason why `str` strings and `bytes` strings that contain only ASCII characters will look similar when displayed using `print` or on an interactive prompt.

We can also specify ASCII characters in a `bytes` literal, so we could write the above literal as:

```
>>> y = b'D5\xc8'
```

The `len` built-in function, when used with `bytes` type, returns the number of bytes contained.

```
>>> len(y)
3
```

We can convert a regular `str` string to `bytes` string by calling the `encode()` method on the string. To convert the encoded plain bytes to a Unicode string of type `str` we can use the `decode()` method on a `bytes` string. These methods take an encoding argument according to which they will do the encoding or decoding.

```
>>> 'AS😄'.encode('utf-8') # bytes representation
of string according to utf-8 encoding
b'AS\xf0\x9f\x98\x84'
```

```
>>> b'AS\xf0\x9f\x98\x84'.decode('utf-8') #
converting encoded bytes back to Unicode string
```



```
'AS😄'
```

```
>>> 'AS😄'.encode('utf-32')
```

```
b'\xff\xfe\x00\x00A\x00\x00\x00S\x00\x00\x00\x04\xff\x01\x00'
```

```
>>>
```

```
b'\xff\xfe\x00\x00A\x00\x00\x00S\x00\x00\x00\x04\xff\x01\x00'.decode('utf-32')
```

```
'AS😄'
```

If the encoding is not specified, the default coding is utf-8, but it is better to be explicit and always specify the encoding argument.

Attempting to encode a string that contains characters not specified in the encoding results in a `UnicodeEncodeError`. For example, we cannot encode 'AS😄' using the `ascii` or `latin-1` encoding as these encodings do not have the '😄' character.

```
>>> 'AS😄'.encode('ascii')
```

```
UnicodeEncodeError: 'ascii' codec can't encode character '\U0001f604' in position 2: ordinal not in range(128)
```

```
>>> 'AS😄'.encode('latin-1')
```

```
UnicodeEncodeError: 'latin-1' codec can't encode character '\U0001f604' in position 2: ordinal not in range(256) '
```

We can use a second argument to ignore the characters that cannot be encoded or replace them with a question mark.

```
>>> 'AS😄'.encode('ascii', 'ignore')
```

```
b'AS'
```

```
>>> 'AS😄'.encode('ascii', 'replace')
```

```
b'AS?'
```

We have seen that the `decode` method returns a string by decoding the bytes in the `bytes` string according to the specified encoding. The decoding should be done using the same encoding scheme used to encode that data. If

not, you might get wrong, garbled text or `UnicodeDecodeError`. For example, if the binary data we get from some source was encoded in UTF-16 and we try to decode it using UTF-8 or any other encoding, we will get an error or sometimes wrong text.

```
>>> data = 'AS😄'
>>> binary_data = data.encode('utf-32')
>>> binary_data.decode('utf-8')
UnicodeDecodeError: 'utf-8' codec can't decode byte
0xff in position 0: invalid start byte
>>> data = '±µ'
>>> binary_data = data.encode('utf-8')
>>> binary_data.decode('latin-1')
'Â±Âµ'
>>> data = 'εθ'
>>> binary_data = data.encode('utf-8')
>>> binary_data.decode('utf-16')
'덤룩'
```

The built-in `bytes()` function can also be used to create a `bytes` object from a `str` string according to the encoding specified.

```
>>> bytes('AS😄', 'utf-8')
b'AS\xf0\x9f\x98\x84'
```

The built-in `len` function, when used on `str` type strings, counts the Unicode characters. It does not count bytes.

```
>>> len('µ')
1
```

'µ' is a string of 1 character irrespective of the number of bytes that will be used to store it. The number of bytes will depend on the encoding scheme used.

```
>>> len('µ'.encode('latin-1'))
1
```

```
>>> len('µ'.encode('utf-8'))
2
>>> len('µ'.encode('utf-16'))
4
>>> len('µ'.encode('utf-32'))
8
```

The `len` function, when used on a `bytes` string, returns the number of bytes. The following examples show that UTF-8 is a variable length encoding that uses different numbers for different characters.

```
>>> len('A'.encode('utf-8'))
1
>>> len('µ'.encode('utf-8'))
2
>>> len('₹'.encode('utf-8'))
3
>>> len('😄'.encode('utf-8'))
4
```

There is another type in Python called `bytearray` which is a mutable variant of `bytes`.

Exercise

1. `s = 'Morning'`

The expression `s[len(s)]` will:

- (A) Give last character of the string
- (B) Show error

2. Strings objects are _____

- (A) mutable (B) immutable

3. If `s = 'Rainbow'`, what is `s[2]`?

(A) 'a' (B) 'i'

4. If `s = 'Rainbow'`, what is `s[-2]`?

(A) 'o' (B) 'b'

5. What will this code display?

```
s = 'Hello' + 2
```

```
print(s)
```

(A) Hello2

(B) HelloHello

(C) TypeError

6. Type of 'x' is:

(A) char (B) str

7. The first character of a string `s` is given by:

(A) `s[-1]`

(B) `s[0]`

(C) `s[1]`

8. The last character of a string of length `n` is given by

(A) `s[-1]`

(B) `s[n-1]`

(C) Both

9. If `s = 'rose'`, then the assignment statement

`s[2] = 'p'` will:

(A) change the string to 'rope'

(B) change the string to 'rpse'

(C) give error

10. If `s = 'hello'`, what will be the value of `s.len()`?

(A) 5

(B) 6

(C) Error

11. A variable that is referencing an immutable value cannot be reassigned to another object.

(A) True (B) False

12. If `s = 'hello world'`, what is `s.capitalize()`?

(A) 'Hello World'

(B) 'HELLO WORLD'

(C) 'Hello world'

13. `s = 'Small gains are better than no gains'`

What is the value of `s.count('n', -10)`

(A) 0 (C) 4

(B) 3 (D) Error

14. `s = 'Hello world'`

What is the value of `s.find('word')`

(A) 0 (C) 6

(B) -1 (D) Raises ValueError

15. What is the value of `s` after this assignment:

```
s = 'Good' + ' ' * 2 + 'Evening' + '!' * 3
```

(A) Good*2Evening!*3

(B) 'Good Evening!!!'

(C) Gives Error

16. What will be the output of the following code?

```
s1 = '<>'
```

```
s1 *= 3
```

```
print(s1)
```

(A) <><><>

(B) Gives error, as strings are immutable

17. What is the value of this expression?

`'.... Where ??? '.strip('?.')`

(A) 'Where' (B) ' Where '

(C) '.... Where'

(D) `strip()` does not take arguments

18. The expression `'cd' not in 'abcde'` returns

(A) True

(B) False

(C) Gives Error

19. The expression `s[s.rindex('$'): ]` will give a string that contains everything _____

(A) before the first occurrence of \$ in s

(B) after the first occurrence of \$ in s

(C) before the last occurrence of \$ in s

(D) after the last occurrence of \$ in s

20. `s.find('Count', 20, 70)`

For the above expression, search will be performed in the portion of string

(A) from index 20 to index 69

(B) from index 20 to index 70

21. Which of these represents a newline character?

(A) `'\l'`

(B) `'\i'`

(C) `'\n'`

22. What is the length of this string?

```
len('Hi\tthere\n\n')
```

(A) 10

(B) 13

(C) 14

23. Which statement will display the given text on the screen?

```
E:\python\numbers.py
```

(A) `print('E:\\python\\numbers.py')`

(B) `print('E:\python\numbers.py')`

(C) `print(r'E:\python\numbers.py')`

(D) All of these

24. Which of these will give syntax error:

(A) `print("Let's face it")`

(B) `print('Don't just exist, live')`

(C) `print('It\'s okay to take a break')`

25. By default, text is ____ aligned and numbers are ____ aligned in their field.

(A) left, right

(B) right, left

26. To perform centre alignment of a value in a field width, which symbol is used.

(A) `<` (C) `^`

(B) `>` (D) `&`

27. What will this code display?

```
fruit = 'banana'
```

```
price = 154.25
```

```
print(f'Price of {fruit} is {price:.6f}')
```

(A) Price of banana is 154.250

(B) Price of banana is 154.250000

28. `n = 23414565755`

What will the following statement print?

```
print(f'{n:,}')
```

(A) 23414565755

(B) 23,414,565,755

(C) 234,145,657,55

29. Which statement will you write for displaying the following number in exponential notation?

```
number = 0.00000000354
```

(A) `print(f'{number:e}')`

(B) `print(f'{number:E}')`

(C) Any of these

30. `number = 2455`

Which statement will display the above number in hexadecimal base?

(A) `print(f'{number, x}')`

(B) `print(f'{number:x}')`

(C) `print(f'{number:h}')`

For questions 31 to 46, use the following string

```
s = 'Ideas are easy, execution is hard.'
```

31. Display the first 5 characters of the string.

32. Display the last 5 characters of the string.

33. Display the 5th character of the string.

34. Display the last character of the string using negative index.

35. Display the reverse of the string.

36. Display the string without the last character.

37. Display the string without the last 5 characters.

38. Display the string without the first 5 characters.
39. What will you get when you write `s[100]` .
40. What will you get when you write `s[-40]` .
41. What will you get when you write `s[6:100]` .
42. What will you get when you write `s[-40:5]` .
43. Make another string `s1` that is an exact copy of `s`.
44. Make another string `s2` from `s` by excluding the last 3 characters.
45. What will you get by writing `s[5:5]`
46. Display every alternate character of the string, starting from index 4 to index 14.
47. Write a statement to change a string such that its first character and last characters are exchanged. If the string is 'Hello World', it becomes 'dello Worlh'.
48. Make a string `s3`, by concatenating the last 4 characters of a string `s1` and first 3 characters of a string `s2`.
49. Make a string `s1` from string `s`, in which the first 2 characters are repeated 5 times, and the last character is repeated 3 times. For example, if the string `s` is 'Hello World !', then the string `s1` is 'HeHeHeHeHello World !!!'
50. Write a program that inputs an email id and extracts the username and domain name from the email id. For example, if email is `myname@some site.com` then username is `myname` and domain name is `some site.com`
(Hint : Use `index()` method)
51. Write a program to extract whatever is enclosed inside asterisks in a string. For example, if the string is 'Deepa 35 *9/11/1977* Najibabad', the portion extracted is 9/11/1977.
(Hint : Use `index()` and `rindex()` methods)
52. `s = ' welcome to bengaluru '`

Write a single statement to strip all the whitespaces from left and right of this string `s` and convert it to title case.

53. `s = 'he he that he that he that that he he that'`

Write a single statement to replace all occurrences of 'he' with 'she' and first 3 occurrences of 'that' with 'this'.

54. Make a new string `s1` from a string `s`, such that the first half of the string `s` is changed to uppercase and the second half to lower case. For example, if string `s` is 'Hello World', string `s1` will be 'HELLO world'

55. Write a single statement to check whether a string `s` begins with 'Line' and ends with 'Done'.

56. Write a statement to create a new string named `code` from three strings named `name`, `dob` and `city`. The string `code` should contain every alternate character from string `name` (only up to 8th character), the first two characters and last 2 characters from string `dob` and the first three characters from string `city`. The string `code` will be 11 characters long.

If `name = 'Johny Abraham'` `dob = '09/11/1987'` `city = 'London'` `code` will be 'JhyA0987Lon'

If `name = 'Marie Claire'` `dob = '12/04/1991'` `city = 'Paris'` `code` will be 'MreC1291Par'

57. Write a statement to print a line that contains 80 dashes.

58. Write a statement to print 5 blank lines. ('\n' is the newline character)

59. Write a statement to find the reverse of an integer `n`.

60. `s = ' Python '`

Will the following two statements give same result.

(i) `s.rjust(20, '-').strip()`

(ii) `s.strip().rjust(20, '-')`

61. What will be the output of the following code?

`s = 'Python'`

```
print(s[len(s)-3], s[-3])
```

62. What will be the output of the following code?

```
s = 'And'
```

```
letters = '_abcdefghijklmnopqrstuvwxyz'
```

```
print(letters.index(s[0].lower()),  
letters.index(s[1]), letters.index(s[2]))
```

63. How will you write a print function call that ends in a colon and a newline?

64. What will be the output of the following code?

```
s = "caattt's curiosity killed the cat"
```

```
print(s.removesuffix('cat'))
```

```
print(s.strip('cat'))
```

Lists and Tuples

4

Lists are ordered collections of items. They can be considered similar to arrays in other languages. They are more flexible and powerful as they do not have fixed sizes and can store elements of different types. Lists are the most commonly used sequence types in Python. Here are some examples of list literals:

```
[27, 13, 14, 26, 19]
[ ]
['papaya', 'apple', 'banana']
[10, 15, 'black', None, 3.5, True, 15,]
```

The elements of a list are separated by commas and are enclosed in square brackets. The first example is a list with five elements, and the second represents an empty list. The elements in a list can be of different types. For example, the fourth list contains values of type `int`, `str`, `NoneType`, `float`, and `bool`. Although lists allow mixed types, they are often used to store values of the same type. They are commonly used to represent collections of similar items, such as a list of names or a list of numbers. By storing values of the same type in a list, we can conveniently perform the same operation on all the elements of a particular list.

Values in a list need not be unique; it can have duplicate values. This means that the same value can appear multiple times at different positions in the list. For example, in the fourth list, the value 15 occurs twice.

We can place a trailing comma at the end of the values in a list literal. For example, in our fourth list literal, we have a comma after the last element, 15, just before the closing square bracket. This trailing comma is ignored

and does not cause any syntax error. This can be useful when you want to add elements to a multiline list or rearrange it while editing your code.

Like integer or string literals, list literals can also be assigned to variables.

```
list1 = [12, 43, 21, 67, 54, 11]
```

When this assignment statement executes, Python creates a list object and makes the name `list1` refer to that object.

A list is a referential data structure, which means that it stores references to its elements. Here is how we can visualize `list1`.

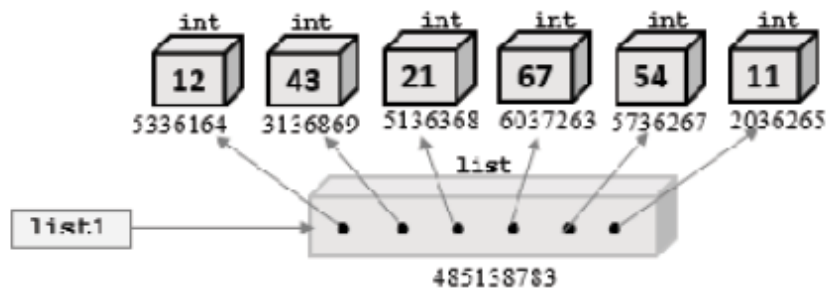


Figure 4.1: List object

The name `list1` refers to the list object, and the list object stores references to different objects that represent the elements of the list. So, although we generally say that a list contains elements, it technically contains references to those elements.

The list type is mutable; this is the first mutable type that we are discussing. ‘Mutable’ means that an object of type `list` can be changed, and its contents can be altered. You can add new elements or delete/overwrite existing elements from the list object. This is why a list can dynamically contract or expand at runtime; its size is not fixed. The interpreter dynamically allocates more memory when required and also dynamically releases the memory no longer required by the list.

We have discussed some properties of a list. Now, before going further, let us discuss why we need the list data type. The list type provides a way to combine related data in order. Let us see an example. Suppose we have this travel itinerary for a 3-week trip:

1. Delhi 2. Bareilly 3. Srinagar 4. Agra 5. Jaipur 6. Mumbai 7. Goa 8. Bangalore 9. Kolkata 10. Varanasi

The order of the destination cities is important here. If we need to manage this trip in our program, then without the list type, we would make ten variables.

```
destination1 = 'Delhi'
destination2 = 'Bareilly'
destination3 = 'Srinagar'
destination4 = 'Agra'
destination5 = 'Jaipur'
destination6 = 'Mumbai'
destination7 = 'Goa'
destination8 = 'Bangalore'
destination9 = 'Kolkata'
destination10 = 'Varanasi'
```

Using a list, we can have all of them in only one data structure and access them using a single name. Since a list is an ordered data structure, the order is preserved.

```
trip = [
    'Delhi', 'Bareilly', 'Srinagar', 'Agra',
    'Jaipur',
    'Mumbai', 'Goa', 'Bangalore', 'Kolkata',
    'Varanasi'
]
```

Now suppose we decide to cut 'Agra', 'Jaipur', and 'Mumbai' from the trip. If we defined 10 variable names, we would have to delete three variable names. This would create confusion, as now, after the name `destination3`, we have the name `destination7`. In the case of a list, we can easily delete the items from anywhere inside the list. Similarly, if we

have to add more cities to the trip, it would be easier if we use a list. Suppose you need to include another trip that involves 20 cities. In that case, you can just make another list instead of defining 20 other names, which is obviously tedious and difficult to maintain in the program.

When we use a list, we can easily insert new items, delete items, replace items, or reorder them. By using a list, we can group related data under one name. Structuring the data inside a list also makes it easier to process it using loops, as discussed in the coming chapters.

Strings, lists, tuples, and range objects are sequences, as they are ordered collections of items. All the sequence operations like indexing, slicing, concatenation, and repetition that we have seen for strings are also valid for lists. However, lists are mutable, so they support other operations that can make in-place changes. This means that you can make changes in the list object itself instead of creating a new changed object, as we had to do in strings.

4.1 Accessing individual elements of a list by indexing

In our program, we can print the whole list by sending the list's name to the `print` function. On the interactive prompt, we can just write the name of the list, and it will be printed. Most of the time, in our program, we would like to access individual elements of the list.

Similar to strings, the elements of a list can be accessed by writing integer index values enclosed in square brackets. Like strings, lists also use zero-based indexing. If `L` is the name of the list, then to access the first element, we write `L[0]`; for the second element `L[1]`, and so on. A list of size `n` has elements indexed from 0 to `n-1`. As in strings, we can also give negative index values to index backward. So, `L[-1]` represents the last element, `L[-2]` the second last element, and so on. For a list of length 6, indices 0,1,2,3,4,5 and -1,-2,-3,-4,-5,-6 are valid indices. Any integer less than -1 or more than 5 will be invalid. If you try to access a list element at an invalid index, the interpreter will raise an `IndexError`.

```
>>> L = [10, 20, 30, 40, 50, 60]
```

```

>>> L
[10, 20, 30, 40, 50, 60]
>>> L[1]
20
>>> L[-1]
60
>>> L[10]
IndexError: list index out of range

```

4.2 Getting parts of a list by slicing

We can extract a portion of the list by slicing. The slice operations that we saw for strings work for lists also in the same way. Slicing a list gives us a part of the list as a new list object. As in strings, we can get a slice of the list by specifying indices separated by colons inside square brackets. The detailed syntax of slicing is not repeated here, as it is exactly the same as in strings. Here are a few examples:

```
L = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
```

L[2:7]	Gives a list that contains elements from index 2 to index 6 [30, 40, 50, 60, 70]
L[2:77]	Gives a list that contains elements from index 2 to index 10 (No IndexError) [30, 40, 50, 60, 70, 80, 90, 100, 110]
L[:5]	Gives a list that contains elements from index 0 to index 4 (first 5 elements) [10, 20, 30, 40, 50]
L[5:]	Gives a list that contains elements from index 5 to index 10 [60, 70, 80, 90, 100, 110]
L[-5:]	Gives a list that contains elements from index -5 to index 10 (last 5 elements) [70, 80, 90, 100, 110]
L[2:9:2]	Gives a list that contains every second element from index 2 to index 8 [30, 50, 70, 90]
L[: :2]	Gives a list that contains every second element starting from first index till last index [10, 30, 50, 70, 90, 110]

<code>L[:]</code>	Gives a list that is an exact copy of the list <code>L</code> [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
<code>L[::-1]</code>	Gives a list that is the reverse of the list <code>L</code> [110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

Table 4.1: Examples of list slicing

If the first number inside the square brackets is omitted, it is considered zero; if the second is omitted, it is considered the last index. The third number represents the step and is optional; if it is omitted, it is considered 1. The slice `L[:]` gives an exact copy of the list, and the slice `L[::-1]` gives the reverse of the list.

You can assign these slices to variable names. For example, if you wish to make a list `L1` that is the reverse of the list `L`, you can write this:

```
L1 = L[::-1]
```

The following statement will make `L2` a copy of the list `L`.

```
L2 = L[:]
```

4.3 Changing an item in a list by index assignment

Since lists are mutable, it is possible to change a list object in-place. We can change any element in the list by assigning it to an index. In the following example, we are changing the element at index 1.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
```

```
>>> L[1] = 50
```

```
>>> L
```

```
[12, 50, 21, 67, 54, 11, 13]
```

If the index is out of range, then an `IndexError` will be raised.

```
>>> L[7] = 100
```

```
IndexError: list assignment index out of range
```

Python performs bounds checking while indexing, so accessing or assigning off the end of a list is an error. The statement `L[7] = 100` will not just silently grow the list; instead, it throws an error. There are specific methods to grow a list, which we will see in a while.

4.4 Changing a Portion of the list by slice assignment

You can modify portions of the list by assigning them to slices. When a list slice is used on the left side of an assignment, the range specified in the slice will be replaced by what is on the right-hand side. Suppose we have this list:

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
```

The following assignment statement replaces the elements at index 2, 3, and 4 with the three elements of the list on the right side:

```
>>> L[2:5] = [300, 400, 500]
```

```
>>> L
```

```
[12, 43, 300, 400, 500, 11, 13]
```

Slice assignment can replace multiple elements of the list in a single step. The length of the list on the right side need not be equal to the length of the slice that is being assigned.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
```

```
>>> L[2:5] = ['a', 'b', 'c', 'd', 'e']
```

```
>>> L
```

```
[12, 43, 'a', 'b', 'c', 'd', 'e', 11, 13]
```

In this example, the length of the slice is three while five items are being assigned, and we can see that all five elements are included in the resultant list. So, the length of the slice and the length of the list that is being assigned need not be the same; the list will shrink or expand to accommodate the new values. This flexibility only exists if you do not provide a step in the slice. When the slice includes a step, the lengths of the slice being assigned to and

the length of the list on the right side should be the same. If the step is not provided, their lengths can be different.

In our example, we have used a list on the right side of an assignment statement. You can use any other iterable; for example, you can have a string or a tuple also on the right side. Let us use a string.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[3:6] = 'abcdef'
>>> L
[12, 43, 21, 'a', 'b', 'c', 'd', 'e', 'f', 13]
```

Now, the specified portion of the list is occupied by characters of the string on the right side.

We can delete a portion of the list by assigning an empty list to a slice.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[3:6] = []
>>> L
[12, 43, 21, 13]
```

Here, all the elements from index 3 to index 5 are deleted from the list.

We know that the slice `L[:]` represents the whole list, so assigning to it will replace the whole list with the list on the right side.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[:] = [1, 2, 3, 4]
>>> L
[1, 2, 3, 4]
```

If you want to clear the whole list, you can write this:

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[:] = []
>>> L
[]
```

```
[]
```

We can insert multiple new elements in a list by squeezing them into an empty slice at the desired location.

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[3:3] = [10, 20, 30, 40]
>>> L
[12, 43, 21, 10, 20, 30, 40, 67, 54, 11, 13]
```

The four elements of the list on the right side are inserted in the list `L` starting at index 3. This way, you can insert new elements without deleting any existing ones.

If you want to add some items to the beginning of the list, you can write this:

```
>>> L = [12, 43, 21, 67, 54, 11, 13]
>>> L[0:0] = [7, 8, 9]          # or L[:0] = [7, 8, 9]
>>> L
[7, 8, 9, 12, 43, 21, 67, 54, 11, 13]
```

When assigning to slices, there should always be an iterable on the right side, even if it contains zero or no elements.

```
>>> L[5:5] = 90
TypeError: can only assign an iterable
>>> L[5:5] = [90]
>>> L
[7, 8, 9, 12, 43, 90, 21, 67, 54, 11, 13]
```

In strings, index assignments and slice assignments are not possible as they are immutable. List objects are mutable, so they can be changed in-place; hence, index and slice assignments are allowed.

These slice assignments are not commonly used in practice as there are specific list methods for performing most insertion and deletion operations. The names of these methods are self-explanatory; hence, using them is

simpler than using slice assignments in most cases. In the following few sections, we will explore these methods.

4.5 Adding an item at the end of the list by using `append()`

The `append()` method adds a single item at the end of the list, and it returns `None`.

```
>>> numbers = [10, 20, 30, 40]
>>> numbers.append(50)
>>> numbers
[10, 20, 30, 40, 50]
```

We have taken a list of four integers and added another integer to it at the end using the `append` method.

4.6 Adding an item anywhere in the list by using `insert()`

The `append` method inserts a new item only at the end of the list. If we want to add a new item at a particular index in the list, we can use the `insert` method. By using this method, we can insert a new element at any place in the list. Like `append`, this method also returns `None`.

In the following list, we have inserted a new element 25 at index 2 using the `insert` method.

```
>>> numbers = [10, 20, 30, 40, 50]
>>> numbers.insert(2, 25)
>>> numbers
[10, 20, 25, 30, 40, 50]
```

The new item is inserted just before the element that was at the index where we want to insert. The element 30 was at index 2, and the new element 25 is

inserted before 30, so now 25 is at index 2. All the elements, including 30 and after it, are shifted right to make room for the new value.

Inserting at index 0 inserts the new item at the beginning of the list.

```
>>> numbers.insert(0, 8)
>>> numbers
[8, 10, 20, 25, 30, 40, 50]
```

If we provide a big index past the end of the list, the element is inserted at the end of the list like `append`. It will not show any error.

```
>>> numbers.insert(1000, 5)
>>> numbers
[8, 10, 20, 25, 30, 40, 50, 5]
```

We got no error, and 5 was inserted at the end of the list.

Adding a new element in between the list or removing it from between the list is costly, as internally, some elements have to be shifted. In case of insertion, some elements might have to be shifted right to make place for a new element. In case of deletion, some elements might have to be shifted left to fill the gap. If the list is large, this shifting can take a lot of time. Insertion or deletion from the end is more efficient, as no shifting is required.

4.7 Adding multiple items at the end by using `extend()` or `+=`

You can add multiple items at the end of the list by using the `extend` method. This method takes an iterable as an argument, and it will add all elements of this iterable to the end of the list. This method also returns `None`. An iterable object is an object that can produce an item on request. All three sequences - lists, strings, and tuples are iterables. Dictionaries and sets are also iterables.

In the following example, we have called the `extend` method on the `numbers` list and sent another list `nums` as the argument.

```
>>> numbers = [10, 20, 30]
```

```
>>> nums = [1, 2, 3, 4]
>>> numbers.extend(nums)
[10, 20, 30, 1, 2, 3, 4]
```

All the elements of `nums` list are added at the end of the `numbers` list. The method `append` will add only one item at the end of the list, while you can use `extend` to add multiple items at the end of the list. So, instead of multiple calls to `append`, you can use the `extend` method as a shorthand. A single `extend` call is more efficient than repeated `append` calls.

If we call `append` and send a list as an argument, that list will be added as one item.

```
>>> numbers = [10, 20, 30]
>>> nums = [1, 2, 3, 4]
>>> numbers.append(nums)
>>> numbers
[10, 20, 30, [1, 2, 3, 4]]
```

We can use other iterable types also in `extend`, like tuple, or string.

```
>>> numbers = [10, 20, 30]
>>> numbers.extend('abcd')
>>> numbers
[10, 20, 30, 'a', 'b', 'c', 'd']
```

All characters of the string argument are added at the end of this list.

The augmented assignment index can also be used to add items from an iterable.

```
>>> numbers += [98, 99, 100]
>>> numbers
[10, 20, 30, 'a', 'b', 'c', 'd', 98, 99, 100]
```

4.8 Removing a single element or a slice by using the del statement

We can use the `del` statement to delete a single element or a slice from the list. `del` is a keyword in Python; it is not a list-specific method like `append` or `extend`.

<code>del L[i]</code>	Removes the element at index <code>i</code>
<code>del L[i:j]</code>	Removes elements from index <code>i</code> to index <code>j-1</code>
<code>del L[i:j:k]</code>	Removes elements from index <code>i</code> to index <code>j-1</code> with a stride of <code>k</code>

Table 4.2: del statement

```
>>> numbers = [10, 20, 30, 40, 50, 60]
>>> del numbers[4] # Deletes element at index 4
>>> numbers
[10, 20, 30, 40, 60]
>>> numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> del numbers[2:7] # Deletes elements from index 2 to index 6
>>> numbers
[10, 20, 80, 90, 100]
```

All the elements after the deleted element are shifted left to fill any gap made by the deleted element. The statement `del numbers[:]` deletes all the elements from the list.

4.9 Removing an element by index and getting it by using pop()

If you want to remove an item from the list and also get the removed item, you can use the method `pop`.

`L.pop(i)` Removes and returns the element at index `i` in the list

`L.pop()` Removes and returns the last element of the list

If we do not specify any index as an argument, then this method removes and returns the last element of the list. So, `pop()` without any argument is the same as `pop(-1)`.

```
>>> numbers = [10, 20, 30, 40, 50, 60, 70, 80]
>>> x = numbers.pop(4)   # removes the element at
index 4
>>> x
50
>>> numbers
[10, 20, 30, 40, 60, 70, 80]
>>> y = numbers.pop(0)   # removes the first element
>>> y
10
>>> numbers
[20, 30, 40, 60, 70, 80]
>>> z = numbers.pop()   # removes the last element
>>> z
80
>>> numbers
[20, 30, 40, 60, 70]
```

If you try to give a non-existent index as an argument, then an `IndexError` will be raised.

The object returned by `pop` is generally assigned to a variable so that it can be used later. If the returned object is not assigned to any variable, then the returned object ceases to exist, and the memory occupied by it is reclaimed by the interpreter.

4.10 Removing an element by value using `remove()`

If you want to remove an element from the list but do not know its index in the list, then you can use the `remove` method. `L.remove(x)` will remove the first occurrence of item `x` from the list `L`, and it returns `None`. If `x` is not found in the list, then it raises `ValueError`.

```
>>> numbers = [10, 20, 30, 40, 50, 60, 70, 80]
>>> numbers.remove(20)
>>> numbers
[10, 30, 40, 50, 60, 70, 80]
>>> numbers.remove(25)
ValueError: list.remove(x): x not in list
```

`numbers.remove(20)` removes the first occurrence of item 20 from the list. If there are multiple occurrences of the item and you want to remove all occurrences, you can use a loop or a list comprehension. We will see how to do this in the coming chapters.

4.11 Removing all the elements by using `clear()`

The method `clear` will remove all items from the list, making it empty.

```
>>> numbers = [10, 20, 30, 40, 50, 60, 70, 80]
>>> numbers.clear()
>>> numbers
[]
```

Let us summarise all the removal methods. If you want to delete an item by index, you can use the `del` statement. If you want to delete an item by index and also want to use the deleted item, use the `pop` method. If you want to

delete an item by value, use the `remove` method. To delete all the elements from the list, use the `clear` method.

The method `clear` was introduced in Python 3. Before that, we could clear the list by using `del` statement or slice assignment only.

```
>>> del numbers[:]
```

```
>>> numbers[:] = []
```

Note that if we assign an empty list to the list name, it does not clear the list in-place.

```
>>> numbers = []    # assigns a new empty list, not
an in-place clearing
```

Clearing the list in-place is important when there are other references referencing the list. For example, when we send the list as an argument to a function, the in-place approach should be used.

4.12 Sorting a List

The elements of a list can be sorted by using the list `sort` method. It sorts the list in-place, which means that it will change your list object. The elements are sorted in ascending order, i.e., they are arranged from smallest to largest. If the elements are strings, they are sorted according to their ASCII values. This method returns `None`.

```
>>> L = [23, 76, 34, 12, 89, 14]
```

```
>>> L.sort()
```

```
>>> L
```

```
[12, 14, 23, 34, 76, 89]
```

To change the sorting order, add the argument `reverse=True`.

```
>>> L = [23, 76, 34, 12, 89, 14]
```

```
>>> L.sort(reverse=True)
```

```
>>> L
```

```
[89, 76, 34, 23, 14, 12]
```

The numbers are now sorted from largest to smallest.

Now, let us use the `sort` method to sort a list of strings.

```
>>> L = ['Cow', 'Zebra', 'Ant', 'Bear', 'Crow',  
         'Wolf']
```

```
>>> L.sort()
```

```
>>> L
```

```
['Ant', 'Bear', 'Cow', 'Crow', 'Wolf', 'Zebra']
```

```
>>> L.sort(reverse=True)
```

```
>>> L
```

```
['Zebra', 'Wolf', 'Crow', 'Cow', 'Bear', 'Ant']
```

We get the results in alphabetical order, but this order will be disturbed if the list contains strings in both lower case and upper case.

```
>>> L = ['Cow', 'Zebra', 'Ant', 'bat', 'crow',  
         'Wolf']
```

```
>>> L.sort()
```

```
>>> L
```

```
['Ant', 'Cow', 'Wolf', 'Zebra', 'bat', 'crow']
```

In the sorted list, we first have all the uppercase strings and then the lowercase strings. This is because the strings are sorted according to their ASCII values. ASCII values of uppercase letters are less than those of lowercase letters, so uppercase letters come before lowercase letters.

Therefore, the `sort` method performs a case-sensitive sort in the case of strings. To perform case insensitive sort, i.e., to ignore the case while sorting, you can send `str.lower` as the argument for the `key` parameter.

```
>>> L = ['Cow', 'Zebra', 'Ant', 'bat', 'crow',  
         'Wolf']
```

```
>>> L.sort(key=str.lower)
```

```
>>> L
```

```
['Ant', 'bat', 'Cow', 'crow', 'Wolf', 'Zebra']
```

Now, the sorting is done in regular alphabetical order, and this is because the sorting is not done on original strings. The `str.lower` function is applied to each string to get a key, and then the sorting is done on those keys. So the sorting is done on these values: 'cow', 'zebra', 'ant', 'bat', 'crow', 'wolf'. The original values of the list remain unchanged; they are not changed to lowercase.

We could do the same thing by sending the `str.upper` function as the argument for the `key` parameter.

```
>>> L = ['Cow', 'Zebra', 'Ant', 'bat', 'crow',  
         'Wolf']
```

```
>>> L.sort(key=str.upper)
```

```
>>> L
```

```
['Ant', 'bat', 'Cow', 'crow', 'Wolf', 'Zebra']
```

Now the sorting is done on these values: 'COW', 'ZEBRA', 'ANT', 'BAT', 'CROW', 'WOLF'

You can send any one-argument function for the `key` parameter, which will be applied to each element of the list to produce its key. The produced key will be used for sorting. In the next example, we will use the `len` function for the `key` parameter.

```
>>> L = ['Cow', 'Zebra', 'Ant', 'bat', 'crow',  
         'Wolf']
```

```
>>> L.sort(key=len)
```

```
>>> L
```

```
['Cow', 'Ant', 'bat', 'crow', 'Wolf', 'Zebra']
```

Now, sorting is done on the following values:

```
len('Cow')->3, len('Zebra')->5, len('Ant')->3,  
len('bat')->3, len('crow')->4, len('Wolf')->4
```

Now, the strings are sorted according to their length.

The `sort` method will not work if the list contains elements of mixed types. If the list contains all strings or all numbers, it is fine, but when a list contains elements of unrelated types, you will get an error. For example, a list of integers and floats will be sorted, but sorting a list of integers and strings will give an error.

```
>>> L1 = [12.4, 12, 13.77, 88, 9.2]
```

```
>>> L1.sort()
```

```
>>> L1
```

```
[9.2, 12, 12.4, 13.77, 88]
```

```
>>> L2 = ['Seven', 'Five', 12, 'Six', 'Two', 300, 99]
```

```
>>> L2.sort()
```

```
TypeError: '<' not supported between instances of  
'int' and 'str'
```

The list `L1` that contains integers and floats is sorted but the list `L2` that contains strings and integers gives `TypeError` because the types are unrelated.

The `sort` method will change the list object in-place, so the original order of the list elements will be lost. If you do not want to modify your original list and want just a sorted copy of the original list, you can use the `sorted()` built-in function. This function does not sort the list in-place, which means that it does not change your list object. It just returns a new list object that is a sorted copy of the list. The returned list object can be assigned to another name.

```
>>> L = [81, 2, 13, 99, 7]
```

```
>>> L1 = sorted(L)
```

```
>>> L1
```

```
[2, 7, 13, 81, 99]
```

```
>>> L
```

```
[81, 2, 13, 99, 7]
```

We can see that the list `L` has not changed. The arguments for `reverse` and key parameters can be used with the `sorted` function also.

4.13 Reversing a List

The `reverse` method reverses the order of the elements of the list in-place. It returns `None`.

```
>>> L = [2, 5, 3, 1, 7, 4]
>>> L.reverse()
>>> L
[4, 7, 1, 3, 5, 2]
```

If you do not want your list to be changed, use the `reversed` built-in function. This function does not return a list. It returns an iterable object that has to be converted to a list.

```
>>> L = [2, 5, 3, 1, 7, 4]
>>> L1 = list(reversed(L))
>>> L1
[4, 7, 1, 3, 5, 2]
>>> L
[2, 5, 3, 1, 7, 4]
```

We have converted the return value of `reversed` function to a list and assigned it to `L1`. The list `L1` contains the elements of list `L` in reversed order, and the list `L` remains unchanged.

As we have seen before, we can get a reversed copy of the list by using the slice `L[::-1]`

```
>>> L = [2, 5, 3, 1, 7, 4]
>>> L1 = L[::-1]
>>> L1
```

```
[4, 7, 1, 3, 5, 2]
```

```
>>> L
```

```
[2, 5, 3, 1, 7, 4]
```

4.14 Finding an item in the list

The membership operators `in` and `not in` can be used to check whether an element is present in the list. If we want to know the index of an element, then we can use the `index` method. It returns the index of the first occurrence of the item in the list. If the item is not present, then it raises `ValueError`. The search can be restricted by providing optional start and end values.

<code>item in L</code>	Returns <code>True</code> if <code>item</code> present in list <code>L</code> , otherwise <code>False</code>
<code>item not in L</code>	Returns <code>True</code> if <code>item</code> not present in list <code>L</code> , otherwise <code>False</code>
<code>L.index(item)</code>	Returns the index of the first occurrence of <code>item</code> in the list
<code>L.index(item,i,j)</code>	Returns the index of the first occurrence of <code>item</code> in a portion of the list starting from index <code>i</code> to index <code>j-1</code>

Table 4.3: *Finding an item in the list*

First, let us check the presence of an item in a list using the `in` operator.

```
>>> numbers = [82, 31, 55, 12, 7, 56, 99, 12, 99, 67, 12]
```

```
>>> 31 in numbers
```

```
True
```

```
>>> 31 not in numbers
```

```
False
```

```
>>> 100 not in numbers
```

```
True
```

Now, let us use the `index` method.

```
>>> numbers.index(12)
```


3

We get the index of the first occurrence of 12 in the list. Let us specify a start value for searching.

```
>>> numbers.index(12, 4)
```

7

Now, item 12 was searched in the portion of the list starting from index 4 till the end of the list. We can specify an end value also.

```
>>> numbers.index(12, 4, 10)
```

7

The search was done from index 4 to index 9. If the searched item is not present in the list, then `ValueError` is raised.

```
>>> numbers.index(100)
```

```
ValueError: 100 is not in list
```

To count the number of occurrences of an item, we can use the method `count`. If the item is not present in the list, it will return 0.

```
>>> numbers.count(12)
```

3

4.15 Comparing Lists

The `==` and `!=` operators can be used to compare two lists for value equality. The `==` operator will evaluate to `True` if the lists have the same content, while the `!=` operator will evaluate to `True` if the contents of the list are different. The lists are compared element by element, starting from the first index till the last index.

```
>>> L1 = [1, 2, 3]
```

```
>>> L2 = [1, 2, 3]
```

```
>>> L3 = [1, 20, 30]
```

```
>>> L1 == L2
```

```
True
```

```
>>> L1 != L3
```

```
True
```

```
>>> L2 == L3
```

```
False
```

If you want to check whether the two lists refer to the same object, you can use the `is` and `is not` operators.

```
>>> L4 = L1
```

```
>>> L1 is L2
```

```
False
```

```
>>> L1 is L4
```

```
True
```

```
>>> L1 is not L4
```

```
False
```

You can also use `<`, `<=`, `>`, and `>=` operators with lists. These operators will work only if the lists contain compatible types of data that support greater-than and less-than comparisons.

```
>>> L1 = [1, 2, 3, 4, 5, 6, 7]
```

```
>>> L2 = [1, 2, 3, 7, 8]
```

```
>>> L1 < L2
```

```
True
```

The two lists are compared element by element till there is a mismatch in the elements being compared. The result will be the result of comparing the two mismatched elements. For example, here, mismatched elements are 4 and 7; since 4 is smaller, `L1` is considered smaller than `L2`.

4.16 Built-in functions used on lists

We have already seen how the built-in functions `sorted` and `reversed` can be used with lists. Here are some more built-in functions that can work with lists.

<code>len(L)</code>	Returns the size of the list
<code>min(L)</code>	Returns the smallest value of the list
<code>max(L)</code>	Returns the largest value of the list
<code>sum(L)</code>	Returns the sum of all the elements of the list if the elements are of numeric type

Table 4.4: Built-in functions

```
>>> numbers = [82, 31, 55, 12, 7, 56, 99, 12, 99,
67, 12]
>>> len(numbers)
11
>>> max(numbers)
99
>>> min(numbers)
7
>>> sum(numbers)
532
>>> average = sum(numbers)/len(numbers)
>>> average
48.36363636363637
```

4.17 Concatenation and Replication

Like strings, we can perform concatenation and repetition in lists using the `+` and `*` operators.

<code>L1 + L2</code>	Returns a new list object which has all elements of lists <code>L1</code> and <code>L2</code>
<code>n * L</code> <code>L * n</code>	Returns a new list object in which all elements of list <code>L</code> are repeated <code>n</code> times

Table 4.5: Concatenation and replication in lists

The + operator combines two lists, and the * operator can be used with a list and an integer to replicate the list a specified number of times. If $n \leq 0$, the result is an empty list.

```
>>> L1 = [1, 2, 3]
>>> L2 = [6, 7, 8]
>>> L3 = L1 + L2
>>> L3
[1, 2, 3, 6, 7, 8]
```

The expression `L1 + L2` returned a new list object which contained all the elements of the first list and then all the elements of the second list, and we have assigned this list object to name `L3`.

```
>>> L1 * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 4 * L1
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The expressions `L1 * 4` and `4 * L1` both give a list object in which the elements of the list `L1` are repeated 4 times.

The augmented assignment statement syntax is also available for these operators.

```
>>> L1 = [1, 2, 3]
>>> L1 += [10, 11, 12, 13]
>>> L1
[1, 2, 3, 10, 11, 12, 13]
>>> L2 = [6, 7, 8]
>>> L2 *= 3
>>> L2
```

```
[6, 7, 8, 6, 7, 8, 6, 7, 8]
```

These statements make in-place changes in the list object, so in the above two examples, list objects `L1` and `L2` are changed in-place.

There is a difference between augmented assignment syntax and simple assignment when used with lists.

Augmented Assignment Simple Assignment

```
L1 += L2
```

```
L1 = L1 + L2
```

```
L *= n
```

```
L = L * n
```

Let us try to understand this difference with an example:

```
>>> L1 = [1, 2, 3]
```

```
>>> id(L1)
```

```
1750909251072
```

```
>>> L1 = L1 + [5, 6, 7]
```

```
>>> L1
```

```
[1, 2, 3, 5, 6, 7]
```

```
>>> id(L1)
```

```
1750909253056
```

```
>>> L2 = [1, 2, 3]
```

```
>>> id(L2)
```

```
1750909253952
```

```
>>> L2 += [5, 6, 7]
```

```
>>> L2
```

```
[1, 2, 3, 5, 6, 7]
```

```
>>> id(L2)
```

```
1750909253952
```

We took two lists `L1` and `L2`; with `L1`, we used the simple assignment, and with `L2`, we used the augmented assignment syntax. We can see that the `id`

of `L1` has changed, but that of `L2` has not changed. This means that in the case of simple assignment, a new object was created which was assigned to `L1`, while in the case of augmented assignment syntax, in-place changes were made to the list object `L2`.

The result is the same whether we use augmented or simple assignments, but the implementation is different. The augmented assignment is more efficient since it makes in-place changes, while in the case of a simple assignment, a new object is created. If you are dealing with large lists, the creation of a new object will require a lot of temporary space. Also, if there are many references referring to the list object, then making in-place changes is the correct approach.

As we have seen in [section 4.7](#), the augmented assignment syntax (`L1+=L2`) is like the `extend` method (`L1.extend(L2)`). It appends all the items of the iterable to the end of the list.

In the case of strings and tuples, the augmented assignment statements work like simple assignment statements since strings and tuples are immutable, and in-place changes are not possible.

4.18 Using a list with functions from the random module

To select a random item from the list or shuffle the list, you can use the `choice` and `shuffle` functions from the `random` module of the standard library.

The `random.choice()` function returns a randomly selected element from the list.

```
>>> import random
>>> colors = ['red', 'blue', 'green', 'yellow']
>>> random.choice(colors)
'blue'
>>> random.choice(colors)
```

```
'green'
```

The `random.shuffle()` function reorders the elements in the list.

```
>>> cities = ['Etah', 'Kasganj', 'Dhampur',  
'Najibabad', 'Bareilly', 'Chennai', 'Bangalore']  
>>> random.shuffle(cities)  
>>> cities  
['Bangalore', 'Kasganj', 'Najibabad', 'Chennai',  
'Bareilly', 'Etah', 'Dhampur']  
>>> random.shuffle(cities)  
>>> cities  
['Bareilly', 'Najibabad', 'Chennai', 'Kasganj',  
'Dhampur', 'Bangalore', 'Etah']
```

This function modifies the list in-place; it does not return a new list.

4.19 Creating a list

We know that the simplest way of creating a list is to write the list literal and make a variable name refer to it.

```
L = [11, 22, 33, 44]
```

You would often like to construct your list dynamically at run time. You can do this by starting with an empty list and adding items at run time using the `append` or `extend` methods.

```
L = []  
item = input('Enter an item : ')  
L.append(item)  
item = input('Enter another item : ')  
L.append(item)
```

If you have an existing iterable that you want in list form, then you can use the `list` function. This function can be used to convert other iterables to a

list.

```
>>> L = list('blue')
```

```
>>> L
```

```
['b', 'l', 'u', 'e']
```

The function call `list('blue')` produces a list of individual characters of the string 'blue'. The `list` function can take any object of iterable type so that you can use other collections like dictionaries, tuples, or sets. You can also make a new list by making a copy of an existing list. We will discuss copying in detail in the following sections.

4.20 Using range to create a list of integers

Lists containing a range of integers are very common. We can use the built-in `range` function to create these types of lists. The `range` function generates a sequence of integers.

```
range(3,10)    3, 4, 5, 6, 7, 8, 9
```

```
range(2,7)     2, 3, 4, 5, 6
```

The call `range(3,10)` generates integers from 3 up to 10. The first number is included, but the second number is excluded. Similarly, the call `range(2,7)` generates integers from 2 up to 7(excluding 7). If we place these calls inside the `list` function, we will get a list of integers.

```
>>> list(range(3, 10))
```

```
[3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(2, 7))
```

```
[2, 3, 4, 5, 6]
```

We can use a step as the third argument to the `range` function, as we had used in slice notation.

```
>>> list(range(1, 20, 2))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```



```
>>> list(range(20, 100, 10))
[20, 30, 40, 50, 60, 70, 80, 90]
>>> list(range(10, 2, -1))
[10, 9, 8, 7, 6, 5, 4, 3]
>>> list(range(100, 20, -10))
[100, 90, 80, 70, 60, 50, 40, 30]
```

In the call `range(1, 20, 2)`, 2 is the step, so we get a list of odd numbers from 1 to 19. In the second example, we have used 10 as the step value. The step can be negative also, as we have in the last two examples.

If there is only one argument in the `range` function, then we get a list from 0 to that number minus 1.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
```

We will use this `range` function for loops also, so it is good to become familiar with it.

4.21 Using the repetition operator to create a list of repeated values

The repetition operator can be used to initialize a list with the same initial value for all the elements. Here are some examples:

```
>>> [0] * 15
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [''] * 5
['', '', '', '', '']
>>> [None] * 4
```

```
[None, None, None, None]
```

The expression `[0] * 15` gives a list of 15 elements, all initialized to 0.

The expression `[''] * 5` creates a list that contains 5 empty strings. In the last example, we get a list of size 4 with all its elements `None`.

There is another Pythonic way of creating a list called List Comprehension. We will discuss that later in a separate chapter.

4.22 Creating a list by splitting a string

We have seen that the `list` function can break a string into individual characters.

```
>>> L = list('I love Python')
>>> L
['I', ' ', 'l', 'o', 'v', 'e', ' ', 'P', 'y', 't', 'h', 'o', 'n']
```

If you want to break the string into words and make a list of words in the string, you can use the `split` method.

`s.split(sep)` splits the string using `sep` as the separator string.

The `split` method of `str` type splits a string on a separator to a list of substrings. If the separator is not specified or is `None`, then any whitespace acts as a separator. Whitespace can be space, tab, or a newline.

```
>>> L = 'I love Python'.split()
>>> L
['I', 'love', 'Python']
```

Here are some more examples:

```
>>> phone = '011-395-343343'
>>> phone.split('-')
['011', '395', '343343']
>>> date = '22/11/1987'
```

```
>>> date.split('/')
['22', '11', '1987']
>>> student = 'Sam      23      Mechanical   A+'
>>> student.split()
['Sam', '23', 'Mechanical', 'A+']
```

In the call `student.split()`, we have not provided any separator, so splitting is done on whitespace characters.

We can limit the number of splits by specifying a second argument.

```
>>> phone = '011-395-343343'
>>> phone.split('-',1)
['011', '395-343343']
```

We have sent 1 as the second argument, so now only one split is done.

If we have a multiline string and we want to break it into single line strings, then we can use either the `split` method with newline character (`'\n'`) as the separator or we can use the `splitlines` method. The method `splitlines()` splits a multiline string into a list of single-line strings. In the next example, we have a multiline string enclosed in triple quotes.

```
>>> quote = '''When failure knocks you down,
... rise again,
... keep moving
... never give up'''
>>> quote.split('\n')
['When failure knocks you down,', 'rise again,',
'keep moving', 'never give up']
>>> quote.splitlines()
['When failure knocks you down,', 'rise again,',
'keep moving', 'never give up']
```

If we call the `split` method on this multiline string without any argument, then splitting will be done on each whitespace character instead of only newline characters.

```
>>> quote.split()

['When', 'failure', 'knocks', 'you', 'down,',
'rise', 'again,', 'keep', 'moving', 'never',
'give', 'up']
```

Note that `split()` and `splitlines()` are string methods, not list methods. They are called on string objects but return list objects.

4.23 Converting a list of strings to a single string using `join()`

The string method named `join()` is the reverse of the `split()` method. It takes a list of strings as an argument and returns a string in which the string elements of the list have been joined by a separator string. The method is called on the separator string, and the list of strings is sent as the argument. Here are some examples:

```
>>> L = ['15', 'May', '2005']
>>> '/'.join(L)
'15/May/2005'
```

We have called the `join` method on the string `'/'` and sent list `L` as the argument. This call gave us a string object in which the elements of the list have been joined by `'/'`. Let us try calling this method on different strings.

```
>>> '.'.join(L)    # joined by dots
'15.May.2005'
>>> ' '.join(L)    # joined by spaces
'15 May 2005'
>>> ''.join(L)     # called on empty string
'15May2005'
```

The list sent as the argument should be a list of strings only, not a list of integers or floats or any other type. Instead of a list, we can use any other iterable that contains strings. So, we can even use a tuple of strings or sets of strings.

If we send a string as the argument, then all the characters of the string are joined.

```
>>> '-'.join('Python')
'P-y-t-h-o-n'
>>> ' '.join('Python')
'P y t h o n'
```

4.24 List of Lists (Nested lists)

Lists can contain elements of any type, including lists. We get a nested list when a list appears as an element in another list. Here is an example of a nested list:

```
L = ['blue', [3, 4, 5], 34]
```

The inner list `[3, 4, 5]` is the second element of the list `L`, so to access it, we can write `L[1]`, and to access the first element of the inner list, we can write `L[1][0]`. To access the second element of the inner list, we can write `L[1][1]` and so on. In the next example, we have a list with all its elements as lists.

```
listA = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [9, 10, 11]]
```

The nested list structure is often used to represent matrices. For example, the following matrix of three rows and four columns can be represented by a nested list of three elements where each element is a list of size 4.

$$\begin{bmatrix} 1 & 4 & 8 & 3 \\ 2 & 5 & 6 & 3 \\ 1 & 9 & 5 & 8 \end{bmatrix}$$

Figure 4.2: Matrix of size 3 X 4

```
>>> A = [  
...     [1, 4, 8, 3],  
...     [2, 5, 6, 3],  
...     [1, 9, 5, 8]  
... ]
```

We can extract a single row using a single index, and a single element of the matrix using double indexes.

```
>>> A[0]  
[1, 4, 8, 3]  
  
>>> A[1]  
[2, 5, 6, 3]  
  
>>> A[1][2]  
6
```

4.25 Copying a list

If we have a list `L`, the statement `L1 = L` does not make an independent copy of the list, and no new object is formed. This assignment only makes an alias. Both `L` and `L1` refer to the same object. If we make any changes in `L` or in `L1`, the changes will be reflected in the other one also. This is called aliasing.

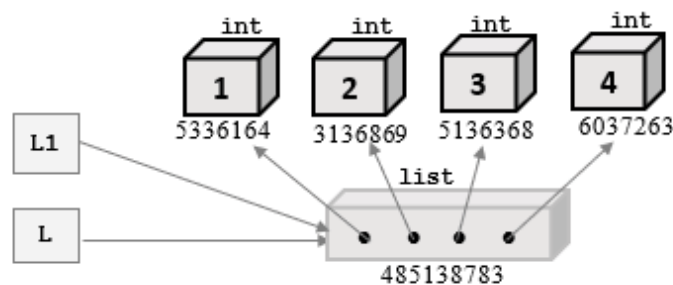


Figure 4.3: Variables `L` and `L1` refer to the same list object

When dealing with objects of immutable types, like integers or strings, then aliasing does not matter much, as neither of the variables can cause a change to the shared object. Immutable objects cannot be changed in any way. But when working with mutable types like lists and dictionaries, this aliasing can lead to unexpected and undesirable behavior, as it can cause unwanted changes in an object. This is because the mutable objects can be changed. With immutable objects, there is no such problem. That is why Python itself aliases small strings for optimization.

If we need to make an independent copy of a list, we have three ways. First is by using the slice notation which we have already seen. The second is by using the `list` function, and the last one by using the `list copy` method. In all these three ways, new list objects are created.

```
>>> L = [1, 2, 3, 4]
>>> L1 = L          # makes an alias
>>> L2 = L[:]       # makes a copy by slice notation
>>> L3 = list(L)    # makes a copy by list function
>>> L4 = L.copy()   # makes a copy by copy method
>>> L1
[1, 2, 3, 4]
>>> L2
[1, 2, 3, 4]
>>> L3
[1, 2, 3, 4]
>>> L4
[1, 2, 3, 4]
```

Let us see the ids of objects that L, L1, L2, L3, L4 are referring to.

```
>>> id(L)
1453383532736
>>> id(L1)
```

```
1453383532736
```

```
>>> id(L2)
```

```
1453383547200
```

```
>>> id(L3)
```

```
1453340650560
```

```
>>> id(L4)
```

```
1453383543744
```

We can see that the ids of `L` and `L1` are the same, which shows that they refer to the same object and, hence, are aliases. ids of `L2`, `L3` and `L4` are all different from the id of `L`, which shows that they are separate independent copies and not aliases. So, in all these three cases, new objects are created. Any changes you make to any of these copies will not be reflected in the original object.

```
>>> L2[0] = 35
```

```
>>> L2
```

```
[35, 2, 3, 4]
```

```
>>> L
```

```
[1, 2, 3, 4]
```

We changed the first element of `L2` to 35, but `L` remains unchanged. Now, let us make some changes in `L3` and `L4`.

```
>>> L3.append(45)
```

```
>>> L3
```

```
[1, 2, 3, 4, 45]
```

```
>>> L
```

```
[1, 2, 3, 4]
```

```
>>> L4[1] += 100
```

```
>>> L4
```



```
[1, 102, 3, 4]
```

```
>>> L
```

```
[1, 2, 3, 4]
```

Any changes made to L2, L3, or L4 do not affect L. However, any changes made to the alias L1 will affect the original object to which L is referring.

```
>>> L1[0] = 99
```

```
>>> L1
```

```
[99, 2, 3, 4]
```

```
>>> L
```

```
[99, 2, 3, 4]
```

We can see that L has changed now. Similarly, any changes made to L will be reflected in the alias L1.

```
>>> L[1] = 1000
```

```
>>> L
```

```
[99, 1000, 3, 4]
```

```
>>> L1
```

```
[99, 1000, 3, 4]
```

We changed list L, and the alias L1 also changed. This change in L will not change L2, L3, or L4 since they are independent copies.

```
>>> L2
```

```
[35, 2, 3, 4]
```

```
>>> L3
```

```
[1, 2, 3, 4, 45]
```

```
>>> L4
```

```
[1, 102, 3, 4]
```

4.26 Shallow copy and deep copy

We saw three ways of copying a list. The copy created in these three ways is a shallow copy; it is just a top-level copy. Let us see what it means.

```
L = [1, 2, 3, 4]
L2 = L[:]          # shallow copy
L2 = list(L)       # shallow copy
L2 = L.copy()      # shallow copy
```

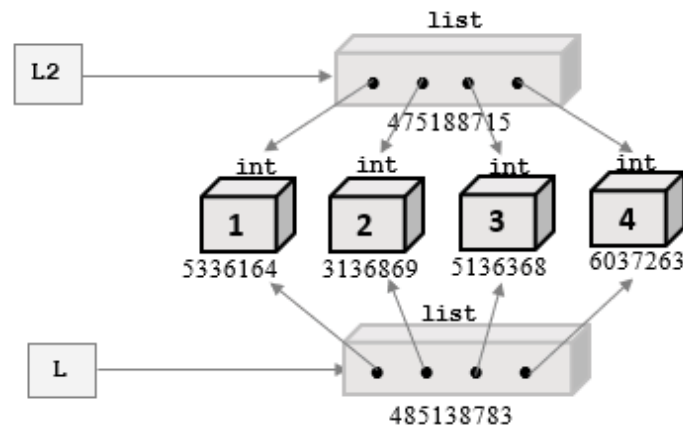


Figure 4.4: Variables L and L2 refer to different list objects

We have a list L, and if we create a copy L2 using any of the three ways we have seen, a new list object is created. This list object contains references to elements from the original list, meaning the contained objects are not copied. This is just a one-level copy. This shallow copy will not create any problems if your list contains only immutable objects, but if your list contains mutable objects, then this shallow copy can produce unwanted results. Let us see how.

Now, suppose our list L contains two integers and a list, and we make a copy L2 by using the `copy` method. We get a new list object that contains references to the three contained objects.

```
>>> L = [12, 13, ['a', 'b']]
>>> L2 = L.copy()
```

```
>>> L2
[12, 13, ['a', 'b']]
```

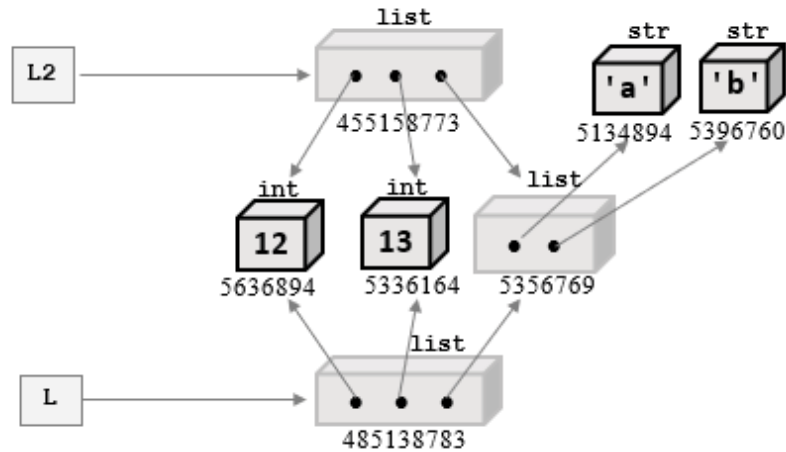


Figure 4.5: Copying the list using the copy method

L and L2 refer to different list objects since we have used the `copy` method. Now, suppose we make in-place changes to the contained list through L2.

```
>>> L2[2].append('c')
>>> L2
[12, 13, ['a', 'b', 'c']]
>>> L
[12, 13, ['a', 'b', 'c']]
```

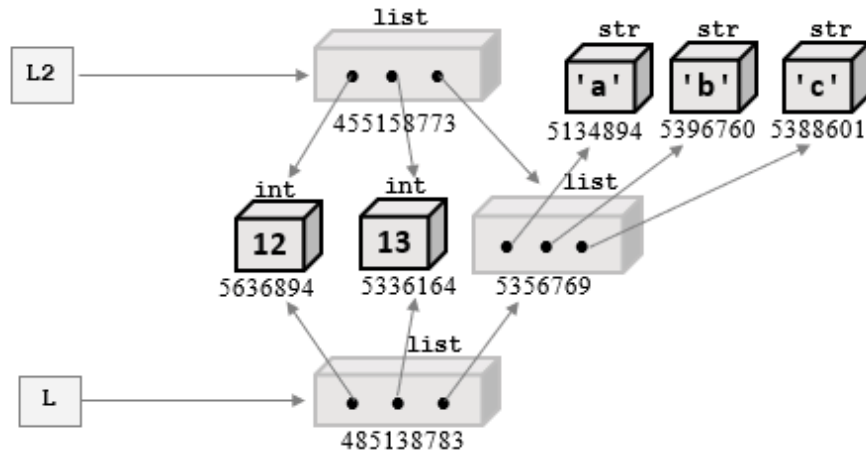


Figure 4.6: In-place changes made to the contained list through L2

`L2[2]` refers to the inner list, so `L2[2].append('c')` calls `append` on the inner list. This call gives us a new element in the inner list. Since the contained list object is shared by both `L` and `L2`, changes made through one are reflected in the other also.

The inner list changed for `L` also because the nested list was not copied; only the reference to it was copied. Immutable contained objects cannot pose any such problem because they cannot be changed in-place. For example, in our list, the integer object is immutable; it cannot be changed in-place, so there is no problem in sharing it. If we write `L2[0] = 22`, there will be no side effect; a new object will be created, and `L2[0]` will refer to this new object now.

```
>>> L2[0] = 22
>>> L2
[22, 13, ['a', 'b', 'c']]
>>> L
[12, 13, ['a', 'b', 'c']]
```

When you have only immutable objects inside your list, a shallow copy is sufficient. When you have mutable objects inside your list, you must perform a deep copy to avoid surprises.

To get a deep copy, you need to use the `deepcopy()` function from the module `copy`. It will give you a complete and independent copy of a deeply

nested data structure. It will recursively traverse objects to copy all their parts. The `deepcopy` function will not do just one level copying; it extends the copying to the last level.

```
>>> L = [12, 13, ['a', 'b']]
>>> L2 = L.copy()
>>> from copy import deepcopy
>>> L3 = deepcopy(L)
```

We made a shallow copy using the `copy` method, then imported the `deepcopy` function from the `copy` module and made a deep copy using this function. Now, let us see the id of the inner list for all three lists.

```
>>> id(L[2])
2038538420416
>>> id(L2[2])
2038538420416
>>> id(L3[2])
2038538490816
```

We can see that `L[2]` and `L2[2]` are referring to the same list object, but `L3[2]` is referring to a new list object. Let us make some changes in the inner list through `L3`.

```
>>> L3[2].append('c')
>>> L3
[12, 13, ['a', 'b', 'c']]
>>> L
[12, 13, ['a', 'b']]
```

Now, there was no change in `L`.

So, we saw the difference between shallow copying and deep copying. In a shallow copy, only object references are copied; the objects themselves are not copied. This leads to the aliasing of contained objects. Most of the time,

shallow copying will be fine; deep copying is required only when you have nested structures like lists within lists or dictionaries within lists.

4.27 Repetition operator with nested lists

When the repetition operator is used with nested lists, we can get unexpected results. Let us understand this with an example.

```
>>> L = [12, ['a', 'b']]
>>> L1 = L * 3
>>> L1
[12, ['a', 'b'], 12, ['a', 'b'], 12, ['a', 'b']]
```

We have a list `L`, and we made another list `L1` by repeating this `L` three times. Now, in the list `L1`, we will make some changes.

```
>>> L1[1][0] = 'z'
>>> L1
[12, ['z', 'b'], 12, ['z', 'b'], 12, ['z', 'b']]
>>> L
[12, ['z', 'b']]
```

We had changed only `L1[1][0]` to `'z'`, but `L1[3][0]` and `L1[5][0]` also have been changed to `'z'`. Even our list `L` has been changed.

This is because when a new list is built using the repetition operator, Python copies each item by reference; it will not create new objects. It just creates references to the same objects. For immutable objects, it is not a problem, but it can be a problem for mutable objects. So, if the list contains mutable objects, using the repetition operator on a list can produce unexpected side effects.

Here is the figure for the example that we have seen:

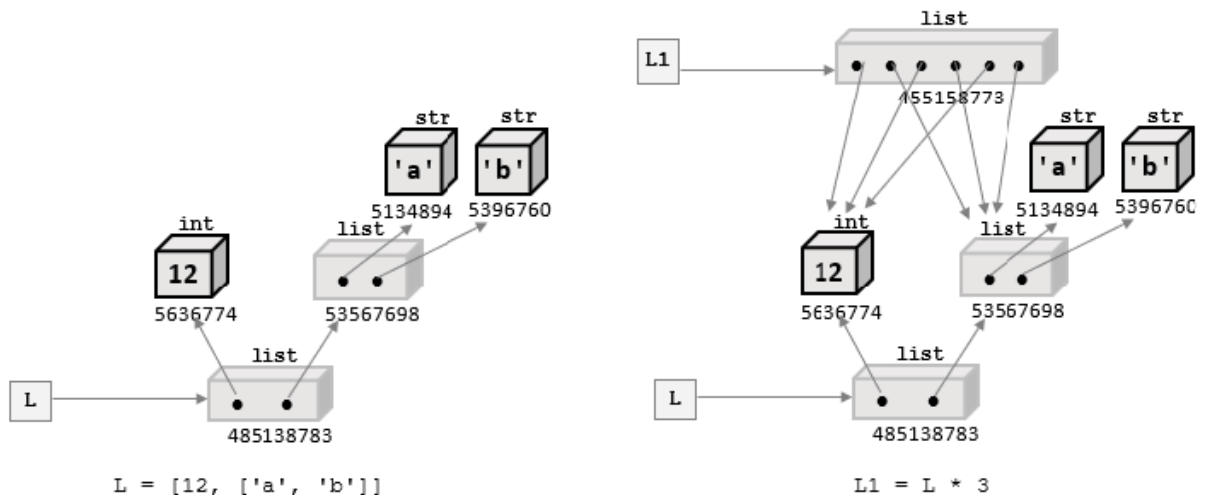


Figure 4.7: List `L1` contains references to the objects of list `L`

We can see that the new list `L1` that we created using the repetition operator contains references to the objects of the original list. The repetition operator does not create any new object. The inner list object has four references referring to it, so any changes made to it through any of the references will be reflected in all four places. We also have four references to the integer object, but this will not create any problem as this is an immutable object, so it cannot be changed in-place.

We can confirm the fact that we have seen in the figure by using the `id` function.

```
>>> id(L[1])
53567698
>>> id(L1[1])
53567698
>>> id(L1[3])
53567698
>>> id(L1[5])
53567698
```

The ids are the same. which means that all of them refer to the same list object.

Let us see one more case where this can create problems. We have seen that we can use the repetition operator to create lists in which all elements have the same initial values. Suppose we want to create a list of empty lists:

```
[[], [], [], []]
```

To get this list, we write the following statement:

```
>>> L = [[]] * 4
```

```
>>> L
```

```
[[], [], [], []]
```

We get a list containing four empty lists, but if we make in-place changes to any of these inner lists, all the inner lists will be affected since they all refer to the same object. Let us append an item to the first sublist.

```
>>> L[0].append(12)
```

```
>>> L [[12], [12], [12], [12]]
```

We appended a value to the first sublist of `L`, but that value has been appended to all the sublists of `L`. This is because we have only one list object, and all the sublists refer to that same object. Let us see one more example:

Suppose we want to create a matrix of size 3 X 4 with all its elements initialized to 0.

```
[ [0, 0, 0, ],  
  [0, 0, 0, ],  
  [0, 0, 0, ]  
]
```

To represent this matrix, we create a nested list using the repetition operator.

```
>>> L = [[0] * 3] * 4
```

```
>>> L
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

We get the properly initialized nested list, but changing any inner list will result in unexpected results.


```
>>> L[1][2] = 34
```

```
>>> L
```

```
[[0, 0, 34], [0, 0, 34], [0, 0, 34], [0, 0, 34]]
```

To avoid these surprising side effects, you should not use the repetition operator with nested lists. You can write the list directly, or if the desired list is big, you can use list comprehensions, which we will discuss in a separate chapter.

Now, let us write the list directly.

```
>>> L = [[], [], [], []]
```

```
>>> L
```

```
[[], [], [], []]
```

```
>>> L[0].append(12)
```

```
>>> L
```

```
[[12], [], [], []]
```

There is no problem now, as all the sublists refer to different objects.

4.28 Tuples

Like lists, tuples are ordered sequences of elements, but they are immutable, which means that once a tuple is defined, it cannot be changed. You cannot dynamically add or remove elements as you do in lists. All the elements have to be defined at the time of creation. The word ‘tuple’ can be pronounced as either ‘toople’ or ‘tupple’. A tuple allows mixed types and can have duplicate values. It is a referential data structure like a list, which means that it contains just references to objects. So, a tuple is like a list, but unlike a list, a tuple is immutable, which means that a tuple object cannot be changed in-place. A tuple object, once created, cannot be modified. For example, the following tuple will always contain the four references in the same order. They will always refer to the same objects. You cannot make these references refer to some other object, nor can you add or remove any item from this tuple.

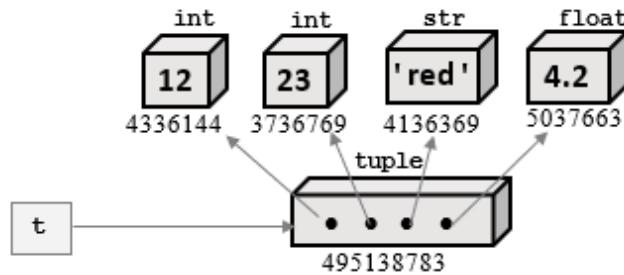


Figure 4.8: *Tuple object*

So, a tuple is a fixed-length data structure whose items cannot be changed. When you have data that needs to be ordered and will not change, put it inside a tuple. Here are some examples of tuple literals.

```
('Joe', 22, 15000)
```

```
('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
```

```
()
```

```
(2,)
```

A tuple literal is written as a comma-separated series of values enclosed in parentheses. An empty pair of parentheses denotes an empty tuple, and a tuple of only one element should contain a comma following that element before the closing parenthesis. If you write the tuple in the last example as `(2)` instead of `(2,)`, it will be wrong because without the trailing comma, the expression is considered a simple parenthesized numeric expression. The expression `(2,)` is of type `tuple`, but the expression `(2)` is of type `int`. So, for a tuple of size 1, you need the trailing comma.

Sometimes, the parentheses enclosing the elements of a tuple can be omitted. For example, the tuple `('Joe', 22, 15000)` can be written as `'Joe', 22, 15000` also. It is better to put the parentheses as it improves clarity and makes the tuple more visible. In some cases, you are not allowed to omit these parentheses, as we will discuss later in the functions chapter when you send a tuple as a function argument.

Tuples are sequences like strings and lists, supporting usual sequence operations like indexing and slicing. Elements of a tuple can be accessed by writing an index inside square brackets or by using slices.

```
>>> days = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu',
            'Fri', 'Sat')
>>> days[2]
'Tue'
>>> days[-1]
'Sat'
>>> days[2:5]
('Tue', 'Wed', 'Thu')
```

Since tuples are immutable, index and slice assignments are not allowed.

```
>>> days[2] = 'Tuesday'
TypeError: 'tuple' object does not support item
assignment
```

We can create a tuple by writing the tuple literal, and the parentheses may be omitted, as we have discussed, or we can use the `tuple` function to convert any iterable to a tuple.

```
>>> numbers = (10, 20, 30, 40)
>>> numbers
(10, 20, 30, 40)
>>> days = 'Sun', 'Mon', 'Tue'
>>> days
('Sun', 'Mon', 'Tue')
>>> t1 = tuple([1, 2, 3])
>>> t1
(1, 2, 3)
>>> t2 = tuple('yes')
>>> t2
('y', 'e', 's')
```

```
>>> t3 = tuple(range(3,7))
```

```
>>> t3
```

```
(3, 4, 5, 6)
```

Tuples support concatenation and repetition like other sequence types.

```
>>> t1 = (1, 2, 3)
```

```
>>> t2 = (4, 5, 6)
```

```
>>> t1 + t2
```

```
(1, 2, 3, 4, 5, 6)
```

```
>>> t1 * 3
```

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

The expression `t1 + t2` gives us a tuple in which we have elements of both `t1` and `t2`, and the expression `t1*3` gives us a tuple in which elements of the tuple `t1` are repeated three times. We can use the augmented assignment syntax also.

```
>> t1 += t2
```

```
>>> t1
```

```
(1, 2, 3, 4, 5, 6)
```

The statement `t1 += t2` does not change the tuple object referred to by `t1`. It just rebinds the name `t1` to a different object. A new object will be created, and that will be assigned to the name `t1`. It is actually equivalent to writing:

```
t1 = t1 + t2
```

Tuples can be compared for their values and identities, and we can use the `in` and `not in` operators with tuples to check the membership of items.

```
>>> t1 = (1, 2, 3, 'black')
```

```
>>> t2 = (1, 2, 3, 'black')
```

```
>>> t1 == t2
```

```
True
```

```
>>> t1 != t2
False
>>> t1 is t2
False
>>> t1 is not t2
True
>>> 2 in t1
True
>>> 2 not in t1
False
```

There are only two methods available for a tuple - `count` and `index`. The call `T.count(x)` returns the number of occurrences of `x` in tuple `T`, and `T.index(x)` returns the index of the first occurrence of `x` in tuple `T`. As in list methods, you can also send additional arguments to restrict your search. Since a tuple is immutable, the `delete`, `append`, or `insert` operations are not defined for tuples. There is no `copy` method for a tuple, so if you want to copy a tuple, you can use the `copy` and `deepcopy` functions of the `copy` module.

Tuples are immutable, so we cannot change a tuple, but if the tuple contains a mutable object, for example, a list, we can change that referenced object. So, a tuple itself cannot be changed, but what it contains can be changed if mutable. Let us see an example:

```
>>> student = ('Ted', 25, [88, 70, 92])
>>> student[1] = 90
TypeError: 'tuple' object does not support item assignment
```

We have a tuple named `student` that contains a string, an integer, and a list. It contains references to three objects. Since a tuple is immutable, it cannot be changed. Its length will always be 3, and it will always contain references to these objects. You cannot make these references refer to any

other object. So that is why when we write `student[1] = 90`, we get an error.

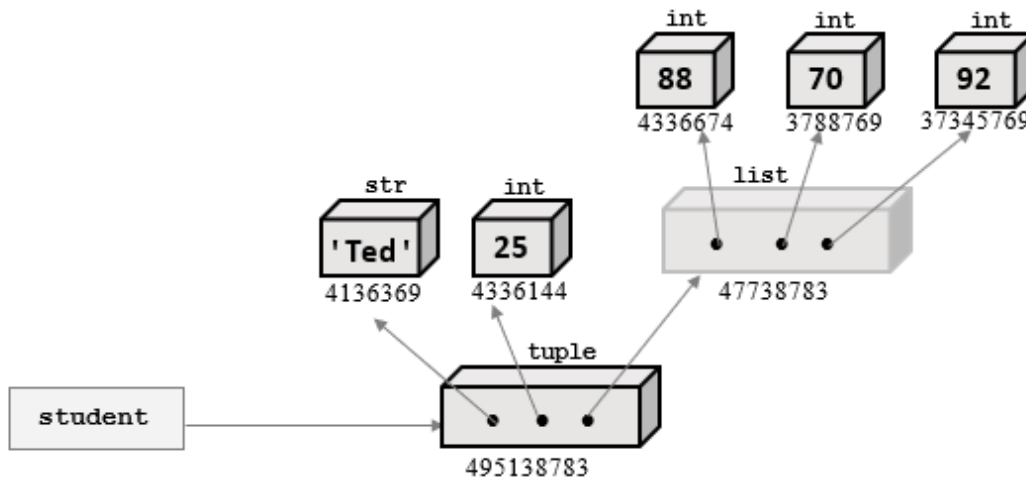


Figure 4.9: Tuple containing a mutable object

From the three objects whose references are contained inside the tuple, the first two (`str` and `int`) are immutable, but the third one, which is a list, is mutable, so it can be changed in-place. So, we can write:

```
>>> student[2][1] = 90
>>> student
('Ted', 25, [88, 90, 92])
```

This is valid because although `student` refers to an immutable object, `student[2]` refers to a mutable object. So, we can make any in-place changes in `student[2]`. Thus, the second reference in the list now refers to a new integer object with a value of 90. When we printed the tuple, we could clearly see a change in it. If a tuple contains a mutable type, we might see a change in it.

The other immutable type that we have seen is `str`. A string can never be changed in any way because it is not a referential structure and does not contain references to characters. It physically holds the characters in contiguous memory.

We have seen that tuples are like lists except for the fact that they are immutable. They have only two methods available. You must be wondering why we need tuples when the list type is already there. The answer is that we

need tuples because of their immutability. Since they are immutable, they provide a sort of safety to your data. If you have a sequence of items, and you create a list out of them and pass that list in the program, chances are that it might be modified at some point in your program because lists are mutable and can be changed. However, if you put your data inside a tuple, it cannot be changed. So, it is safe to use a tuple if you do not want your data to be changed. There can be no aliasing problems in tuples because of their immutability.

Tuples are processed faster than lists. This is because their contents do not change, so Python can implement some optimizations, which make tuples a little faster than lists.

Tuples allow a function to return multiple values. We will discuss this later when we learn about functions. Some built-in methods and functions like `enumerate`, `divmod`, `zip` use this feature and return multiple values in the form of tuples. So, even if you do not create your own tuple, you might have to use tuples that are returned by functions or methods that you use from standard library or other packages.

Tuples can be used as keys in a dictionary. We will learn about dictionaries in the next chapter. Only immutable types like strings and integers can be used as keys of a dictionary. We cannot use a list as a key as it is mutable. A tuple can be used as a key if it contains only immutable elements; if it contains any mutable element directly or indirectly, it cannot be used as a key.

We have seen that tuples are safer and faster than lists, allow us to return more than one thing from a function, and can be used as dictionary keys. So, suppose you have an ordered sequence of values that you are sure will not change. In that case, it is better to use a tuple for better performance and safety. Using a tuple also conveys the message to the reader of your program that you do not intend the sequence of values to be changed.

Although both lists and tuples allow data of mixed type, lists are usually homogeneous, while tuples are usually heterogeneous. In the real world, tuples are mostly used to store records. Lists are generally iterated over using loops, while tuple elements are usually accessed using unpacking. In the next section, we will discuss tuple packing and unpacking.

4.29 Tuple packing and unpacking

The following assignment statement packs data into a tuple.

```
>>> employee = ('Raj', 20, 'Delhi', 15000)
```

The four values are packed into a tuple, and this tuple is assigned to the name `employee`. We could write this statement without the parentheses also.

```
>>> employee = 'Raj', 20, 'Delhi', 15000
```

This is called packing a tuple. Unpacking is the reverse of packing. We can use tuple unpacking to extract data from it.

```
>>> name, age, city, salary = employee
```

In this statement, we are assigning a single tuple to multiple variables. So here, the first value of `employee` tuple is assigned to `name`, second to `age`, third to `city`, and fourth to `salary`. The packing and unpacking can be done at the same time in a single line.

```
>>> name, age, city, salary = ('Raj', 20, 'Delhi', 15000)
```

Here, first, the 4 values that are there on the right side are packed into a tuple, and then they are unpacked. The variable `name` is bound to the string `'Raj'`, `age` is bound to `20`, `city` is bound to `Delhi` and `salary` is bound to `15000`. Parentheses are not necessary, so you can write it like this also.

```
>>>> name, age, city, salary = 'Raj', 20, 'Delhi', 15000
```

This is why you can do multiple assignments in a single statement in Python.

```
>>>> a, b, c = 2, 30, 1
```

When we write a statement like this, multiple assignments are being done. This is also called simultaneous assignment; `a` is assigned value `2`, `b` is assigned value `30`, and `c` is assigned value `1`. We have seen this in the second chapter. What actually happens is that the three values on the right-hand side are automatically packed into a tuple. Then, that tuple is automatically unpacked, with its elements assigned to the three variables on the left-hand

side. So, now you know that behind this multiple assignment technique of Python, there is tuple packing and unpacking going on.

One application of tuple unpacking is swapping the values of two variables without using a temporary variable. In other languages, you would swap the values of two variables, `x` and `y`, like this.

```
temp = x
x = y
y = temp
```

In Python, you can do it in a single statement by using a tuple assignment.

```
x, y = y, x
```

This is the Pythonic way of swapping two values. There was no need to create any temporary variable to hold the data temporarily while swapping the values. The right-hand side is evaluated first, so the two values are packed in a tuple, and then that tuple is unpacked. The first value is assigned to `x`, and the second value is assigned to `y`. So, the old value of `y` is assigned to `x`, and the old value of `x` is assigned to `y`. The unnamed tuple that is automatically packed and unpacked implicitly serves as the temporary variable.

The unpacking works not only for tuples. It can work for any iterable type.

```
>>> x, y, z = [1, 2, 3]
>>> print(x, y, z)
1 2 3
>>> first, second, third = 'not'
>>> print(first, second, third)
n o t
>>> d, m, y = '22/11/1987'.split('/')
>>> print(d, m, y)
22 11 1987
>>> a, b, c, d = range(3, 7)
```

```
>>> print(a, b, c, d)
```

```
3 4 5 6
```

In the first example, we are unpacking a list. `x` gets the value 1, `y` gets 2, and `z` gets 3. Next, we have unpacked a string so the variables `first`, `second` and `third` get values `'n'`, `'o'` and `'t'` respectively. In the next example, the `split` method returns a list, so the variables `d`, `m`, and `y` get the values 22, 11, and 1987, respectively. In the last example, we are using unpacking with the `range` function, so variable `a` is 3, `b` is 4, `c` is 5, and `d` is 6. We can use this trick to assign names to a range of values.

```
>>> black, white, green, blue, red, yellow =  
range(1,7)
```

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
SATURDAY, SUNDAY) = range(7)
```

This example that defines integer constants for days of the week is from the built-in `calendar` module.

We can call the `split` method on the string returned by `input` function, and unpack the list returned by the `split` method.

```
>>> c1, c2, c3 = input('Enter three colours :  

```

```
Enter three colours : red blue green
```

```
>>> print(c1, c2, c3)
```

```
red blue green
```

This way, we can break the input and, therefore, ask the user to enter values for multiple variables using a single input call.

Unpacking will give you an error if the number of variables on the left side is not equal to the number of elements in the right-side collection.

```
>>> x, y = [1, 2, 3]
```

```
ValueError: too many values to unpack (expected 2)
```

```
>>> w, x, y, z = [1, 2, 3]
```

```
ValueError: not enough values to unpack (expected 4, got 3)
```

In the next example, we have a list that contains a string, an integer, and a tuple. If we try to unpack the list with five variables on the left, we get an error because the number of values on the left-hand side is not equal to the number of values in the right-hand side collection.

```
>>> L = ['Dev', 10, (29, 4, 2013)]
```

```
>>> name, age, d, m, y = L
```

```
ValueError: not enough values to unpack (expected 5, got 3)
```

The correct way to unpack is by enclosing variables `d`, `m`, and `y` in parentheses.

```
>>> name, age, (d, m, y) = L
```

```
>>> print(name, age, d, m, y)
```

```
Dev 10 29 4 2013
```

So, `name` gets the value `'Dev'`, `age` gets the value `10`, and variables `d`, `m`, and `y` get the values `29`, `4`, and `2013` respectively. This is one of the examples of situations where parentheses of a tuple cannot be omitted.

While unpacking, we can ignore some values from the tuple if we do not need them. In the following example, we have a tuple named `employee`, and we are unpacking it.

```
>>> employee = ('Raj', 25, 'Delhi', 'raj@abc.com', 'XY289', 15000)
```

```
>>> name, age, city, email, id, salary = employee
```

Suppose we do not want the last two values of this tuple. We want to unpack only the first four values. We have seen that if we write only four variables on the left side, we will get an error as the number of variables on the left is not equal to the number of values on the right side.

```
>>> name, age, city, email = employee
```

```
ValueError: too many values to unpack (expected 4)
```

The solution to this problem is to give any dummy name to satisfy the syntax. The convention is to use an

underscore, which is a valid name in Python.

```
>>> name, age, city, email, _ , _ = employee
```

```
>>> name, _ , city, email, _ , salary = employee
```

In the first statement, we have ignored the last 2 values; in the second statement, we have ignored the second and fifth values. This way, we can ignore some values and satisfy the interpreter.

You could use any other variable name here instead of the underscore. There is nothing special about this underscore. For example, you can use the name `dummy` here.

```
>>> name, dummy, city, email, dummy, salary =  
employee
```

However, using an underscore is a convention and it is easier to type a single underscore than typing any other variable name. If you want to ignore multiple adjacent values, you can use an asterisk before a variable name.

```
>>> name, *_ , salary = employee
```

Here, `name` will get the first value of the tuple, `salary` will get the last value, and all other values in between are ignored. Again, using an underscore here is the convention. You can use any other variable also. For example, we have used the name `skip` here.

```
>>> name, *skip, salary = employee
```

All the values that we have ignored will actually be collected in a list named `skip`. In the previous statement where we used `*_`, the name of the list will be `_`, which is a valid name.

You might not always want to throw the values, so in that case, you can use a meaningful variable name instead of the throwaway variable `_`.

```
>>> record = ('Ted', 25, 'Paris', 'Java', 'C++',  
'C', 'Python')
```

```
>>> name, age, city, *languages = record
```

```
>>> languages
['Java', 'C++', 'C', 'Python']
```

Here, we know that the first element is name, the second is age, the third is city, and after that, every element is a language. So, we have collected the remaining elements in the list named `languages`. Here is another example:

```
>>> author = ('Learn C', 'Python Programming',
'Data structures', 'Alex', 'alex@gmail.com')
>>> *books, name, email = author
>>> books
['Learn C', 'Python Programming', 'Data
structures']
```

Here, we know that the second last element is name, the last is email, and before that, everything is the name of a book. So, we have placed the starred variable in the beginning.

Exercise

What will be the output of the code given in Questions 1 to 48?

```
1. listA = [11, 22, 33, 44]
   print(listA[2.0])
2. listA = [1, 2, 3, 4]
   listA[3] = 100
   print(listA)
3. listA = [4, 5, 6, 7, 8, 9, 10, 11,
            12, 1, 3, 14, 15, 16, 17]
   print(listA[2:9:2])
4. listB = [10, 20, 30]
   listB[3] = 40
```

```
    print(listB)
5. listA = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    listA[2:4] = [10, 20, 30, 40, 50]
    print(listA)
6. listA = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    listA[3:5] = []
    print(listA)
7. listA = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    listA[3] = []
    print(listA)
8. listA = [1, 2, 3, 4, 5, 6]
    print(listA[4:4])
9. listA = [1, 2, 3, 4, 5]
    listA[3:3] = 'abcd'
    print(listA)
10. numbers = [2, 4, 11, 6, 3, 9, 19]
    print(10 not in numbers)
11. listA = [1, 2, 3]
    listA = listA * 3
    print(listA)
12. listA = ['ab', 'cd', 'ef', 'gh']
    x = sum(listA)
    print(x)
13. L = [''] * 3
    print(L)
14. listA = list('Welcome')
```

```
    print(listA)
15.L = list(range(5))
    print(L)
16.L = list(range(100, 0, 10))
    print(L)
17.L = list(range(3, 15, 3))
    print(L)
18.avengers = 'Thor,Iron man,Hulk,Ant-Man'
    listA = avengers.split(',')
    print(listA)
19.print('ab-cd-de-fg-hi-jk'.split('-',3))
20.listA = [1, 2, 3]
    numbers = [10, listA, 20]
    del listA
    print(numbers)
21.a = 1
    b = 2
    c = 3
    list1 = [a, b, c]
    b = 100
    print(list1)
22.L = [[]] * 3
    L[2].append('x')
    print(L)
23.names = ['Ami', 'Sam', 'Amitabh', 'Jim']
    print(names[-2][-3])
```

```
24.names = ['Ami', 'Jim', 'Tim', 'Ron']
    names.append(['Dev', 'Raj', 'Sam'])
    print(len(names))
25.listX = [0] * 5
    listX[1] = 45
    print(listX)
26.listA = [[0]] * 4
    listA[1].extend([4,5])
    listA[2].append(9)
    print(listA)
27.x = [[11, 2, 6], [5, 9, 1]] * 3
    x[0].sort()
    x[1] = sorted(x[1])
    print(x)
28.t = (6, 7, 8)
    t = t * 2
    print(t)
29.t = (1, 2, 3, 4)
    x, y, z = t
    print(x, y, z)
30.t = (1, 2, 3, 4, 5, 6, 7, 8)
    x, _, y, *_ = t
    print(x, y, _)
31.listA = [4, 5, 6, 7, 8, 9, 10]
    listA[2:5] = []
    print(listA, end='    ')
```



```

    listA[2] = []
    print(listA)
32. listA = [4, 3, 2, 6]
    listA = listA.sort(reverse=True)
    print(listA, end=' ')
    listB = [9, 4, 3]
    listB = listB.append(5)
    print(listB)
33. numbers = [1, 2, 3]
    numbers.extend([4, 5, 6])
    print(len(numbers), end=' ')
    numbers.append([7, 8, 9])
    print(len(numbers))
34. x = [1, 2, 3]
    y = [x] * 4
    z = x * 4
    print(y, z)
35. date = '09/08/1973'
    print('-'.join(date.split('/')))
36. t2 = 4, 5, 6
    print(type(t2))
37. t1 = ('hello')
    t2 = ('hello',)
    print(type(t1), type(t2))
38. a, b, c = range(1, 3)
    print(a, b, c)

```

```
39.t = (1, 2, 3, 4, 5, 6)
    a, b, _, c, d, e = t
    print(_)
40.t = (1, 2, 3, 4, 5, 6)
    a, b, *_ , e = t
    print(_)
41.numbers = [1, 2, 3, 4]
    print(numbers[:], numbers[::-1])
42.x = list(range(1, 6, 2))
    y = list(range(1, 7, 2))
    print(x == y)
43.print([10, 20, 30, 40, 50, 60][2:4][1])
44.x = 1, 2, 3
    a, b, c = 1, 2, 3
    print(x, a, b, c)
45.L1 = [1, 2, 3, 4]
    L1.append([])
    L2 = [1, 2, 3, 4]
    L2.extend([])
    print(L1, L2)
46.L1 = [3, 2, 5]
    L2 = [6, 8, 1, 9]
    x = sorted(L1) + sorted(L2)
    y = sorted(L1 + L2)
    print(x, y)
47.L1 = [1, 2, 3]
```

```
L1 += 100
```

```
L2 = [1, 2, 3]
```

```
L2[1] += 100
```

```
print(L1, L2)
```

48. `numbers = [98, 11, 22, 9, 6, 32, 5]`

```
print(sorted(numbers)[2:4])
```

49. What are the valid indices for a list of length 4?

(A) 1, 2, 3, 4 (C) 0, 1, 2, 3, -1, -2, -3

(B) 0, 1, 2, 3 (D) 0, 1, 2, 3, -1, -2, -3, -4

50. `fruits = ['fig', 'apple', 'mango', 'orange']`

What is the result of `fruits.index('banana')` ?

(A) Returns -1 (C) Raises ValueError

(B) Returns None (D) Raises IndexError

51. `marks = [86, 93, 93, 67, 92, 89, 92, 93, 52, 92, 91]`

What is the value of `marks.count(max(marks))` ?

(A) 93 (C) 3

(B) 92 (D) 0

52. Which of these expressions will search for element 12 in last 5 elements of a list L?

(A) `L.index(12, 5)` (B) `L.index(12, -5)`

53. `listA = [3, 4, 5, 6]`

The expression `listA += [10]`

(A) reassigns `listA` to a different object (B) makes in-place changes in `listA`

54. What is the value of the following expression?

```
[1, 2, 3] + 'abc'
```

- (A) [1, 2, 3, a, b, c]
(B) '123abc'

(C) Raises TypeError

55. Which one of these will create an empty list?

- (A) listA = []
(B) listA = list()
(C) Both

56. Which of these is not a tuple?

- (A) (23) (C) (23,)
(B) (23, 5)

57. t = (1, 2, 3, 4)

Which of these are valid operations for tuple t?

- (i) t[1] = 100 (ii) t = t + (100,)
(A) only (i) is valid (C) both (i) and (ii) valid
(B) only (ii) is valid (D) both (i) and (ii) invalid

58. student = ('Dev', 32, [12, 13, 14], (88, 98))

Which one of these is a valid operation?

- (A) student[0] = 'Joseph' (C) student[2][1] = 34
(B) student[0][1] = 'r' (D) student[3][1] = 34

59. Will this code give an error?

```
L = ['Dev', 25, (12,)]  
name, age, d = L
```

- (A) Yes (B) No

In questions 60 to 77, write statements to perform the given operations on the following list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

60. Change the second last element of the list to 200.
61. Replace the elements 3, 4, 5, 6 with elements 30, 40, 50, 60, 70, 80
62. Replace all the elements from index 3 onwards with the characters of the string 'pqr'.
- Resulting list should be [1, 2, 3, 'p', 'q', 'r']
63. Insert new elements 10, 20, 30, 40, 50 starting at index 5.
- Resulting list should be [1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 6, 7, 8]
64. Delete all elements from index 2 to index 5. Resulting list should be [1, 2, 7, 8]
65. Make a new list named `cpy` that is a copy of the `numbers` list.
66. Make a new list named `rev` that is reverse of the `numbers` list
67. Add 100 at the end of the list
68. Add 200 in the beginning of the list
69. Add 150 at index 3
70. Add 12, 13, 14, 15 at the end of the list in one step.
71. Delete element 5 from the list.
72. Delete the last element from the list
73. Delete the element at index 5 and store it in a variable
74. Delete the first element from the list
75. Delete all the elements of the list
76. Use the `del` keyword to delete the element at index 5.
77. Use the `del` keyword to delete the last 3 elements.

Use the following list for questions 78 to 92

```
numbers = [12, 32, 55, 67, 3, 55, 68, 22, 55, 89, 55, 1, 19, 32]
```

Write code to perform the following operations:

78. Find the number of occurrences of 55 in the list.
79. Find the index of the first occurrence of 55 in the list.
80. Find the index of last occurrence of 55 in the list.
81. Find the index of the first occurrence of 55 in a portion of the list, starting from index 4 to index 9.
82. Find the index of the smallest element of the list.
83. Replace the largest element of the list with 1000.
84. Find the second largest and third smallest elements from the list.
85. Make a new list that contains the three largest elements of the list.
86. Find the sum of the five smallest elements of the list.
87. Find the minimum value of the first half of the list.
88. Find the average of all the elements of the list.
89. Make a new list that contains the 5 largest elements from the list.
90. Make a new list that contains the 5 smallest elements from the list.
91. Sort the list in descending order.
92. Make a new list that contains all the elements of the `numbers` list in ascending order. The original list should not change.
93. Sort this list of strings based on their length.

```
fruits = ['banana', 'fig', 'Mango',  
          'pomegranate', 'Apple']
```

94. Perform case insensitive sort on this list of strings.

```
fruits = ['banana', 'fig', 'Mango',  
          'pomegranate', 'Apple']
```

95. Write a statement to create a list of size 20 with all elements initialized to `None`.
96. Create the following list by using the `range` function.

```
[1000, 900, 800, 700, 600, 500, 400, 300, 200,  
100]
```

97. Create a list of all multiples of 7 greater than 50 and less than 150, using the range function.

98. `listD = ['Pluto', 'Goofy', 'Donald Duck', 'Alice']`

Create a string using the join method in which all these strings are joined by a comma

99. Write an expression that will give you the reverse of the string present at index 2 of the following list.

```
fruits = ['apple', 'banana', 'grapes', 'guava']
```

100. What will be the output of this code?

```
student = ('John', 25, [88, 90, 92])
student[2].extend([89, 98])
print(student)
```

101. What will be the output?

```
L = [1, 2, 3]
X = ['a', L]
X[1][0] = 100
print(L)
```

What can you do to avoid the side effect that is seen in this code?

102. How would you write this code in Pythonic way?

```
x = 3
y = 2
temp = x    # save old value of x
x = y * x   # change x
y = temp    # set y to old value of x
print(x, y)
```

103. What is the difference between `L1 = L1 + L2` and `L1.extend(L2)` ?
104. What is the difference between `listA.clear()` , `del listA` and `del listA[:]` .?
105. What is the difference between `L1 = L.sort()` and `L1 = sorted(L)` ?
106. What is the difference between `L[:3] = []` , `L[3]=[]` and `L[3:]=[]` ?
107. Rewrite the following code using tuple unpacking.
- ```
employee = ('Ken', 'London', 26, 4000)
name = employee[0]
city = employee[1]
age = employee[2]
salary = employee[3]
```
108. Write code to swap first and last values of a list L.
109. Use `input` function and `split` method to input 5 colours, separated by hyphens(-). Collect the input in a list.



# Dictionaries and Sets

---

## 5

In the previous chapter, we discussed how to store data using lists and tuples. In this chapter, we will discuss two more data structures named dictionaries and sets. Dictionaries help you organize and structure your data in a better way. It is easier to represent real-world data using a dictionary. Both dictionaries and sets are internally implemented in such a way that they perform very fast searching.

## 5.1 Dictionaries

The dictionary data structure is a collection of key-value pairs. Each element of a dictionary is a key-value pair which is also known as an item. Here is an example of a dictionary literal:

```
countries = {'IN': 'India', 'GR': 'Germany', 'MX': 'Mexico', 'JP': 'Japan'}
```

This dictionary contains four key-value pairs. The strings 'IN', 'GR', 'MX', and 'JP' are keys, and the strings 'India', 'Germany', 'Mexico', and 'Japan' are the corresponding values. The key-value pairs are separated by commas and are enclosed inside curly braces. In each pair, the key and the value are separated by a colon. The dictionary literal has been assigned to the name `countries`. Typing the name of the dictionary on the shell prompt or printing it by using the `print` function will display all its contents.

```
>>> countries
{'IN': 'India', 'GR': 'Germany', 'MX': 'Mexico', 'JP': 'Japan'}
```

In our example dictionary, both keys and values are of `str` type. They can be of other types also, but there is a restriction on the type of keys. The keys can be of immutable type only; you cannot have a key of mutable type.

Therefore, a key can be a string, an integer, a tuple, or any other immutable type; however, most of the time, it is a string. There is no such restriction on values; they can be of mutable or immutable types. So, a value in a dictionary could be a string, integer, list, tuple dictionary, or any other type.

The other restriction on keys is that they must be unique; duplicate keys are not allowed. Again, there is no such restriction on values. They can be duplicated, and the same value can be associated with any number of keys. So, you cannot have key-value pairs where the keys are the same, but you can have key-value pairs where the values are the same. A key can appear only once, while a value can occur many times.

Like lists and tuples, you can have a trailing comma in a dictionary literal also.

```
countries = {'IN': 'India', 'GR': 'Germany', 'MX':
'Mexico', 'JP': 'Japan', }
```

Dictionaries are mutable data structures like lists, so a dictionary can shrink or grow at run time, and its elements can be changed. Like a list, a dictionary is also a referential data structure which means that it contains references to objects; both keys and values are object references.

Searching in dictionaries is performed by keys. You can provide the name of the key to retrieve the value associated with that key. For example, in our `countries` dictionary, we can get the name of a country from its abbreviation, which is used as the key. Dictionaries are highly optimized, so this lookup is very fast. If we try to structure our data of country names and abbreviations by using a list, it would be difficult to implement and also would be inefficient.

Now let us discuss how we can access a value corresponding to a given key. In lists and strings, we use an integer index inside the square brackets to access a value; in dictionaries, we will use a key inside the square brackets to retrieve a value. For example, the expression `countries['IN']` will give us the value associated with the key `'IN'`.

```
>>> countries['IN']
```

```
'India'
>>> countries['MX']
'Mexico'
```

Let us discuss some more examples where dictionaries can be used. You will generally need to create a dictionary when you have some data that is in tabular form. In [Figure 5.1](#), we have some data samples written in tables. The first one is the record of a student; the left column is the field name, and the right column is the value of that field. In the second table, the left column is the product name, and the right column is its price, and in the third one, the left column contains the designation, and the right column represents the associated salary.

| Student   |            | Price     |    | Salary     |       |
|-----------|------------|-----------|----|------------|-------|
| name      | John       | pencil    | 10 | programmer | 10000 |
| gender    | M          | pen       | 22 | manager    | 20000 |
| city      | Paris      | eraser    | 12 | accountant | 15000 |
| age       | 21         | sharpener | 13 |            |       |
| marks     | 89, 78, 91 | marker    | 32 |            |       |
| is_sporty | True       |           |    |            |       |

*Figure 5.1: Data in tabular form*

First, let us represent the student data using a list.

```
student = ['John', 'M', 'Paris', 21, [89, 78, 91],
True]
```

When we need to access a student's name, we will write `student[0]`, and when we need the student's age, we will write `student[3]`. The problem with this representation is that we must remember that the name is at index location 0, the gender is at index location 1, and so on. All the values are there in the list, but there is no information about the values, so a list is possibly not the best choice here. When the values are identified by their names, we need to use a dictionary. Let us put the same data in a dictionary.

```
>>> student = {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
```

```
 'age': 21,
 'marks': [89, 78, 91],
 'is_sporty': True
 }
```

Now there is more information, and we know what each value represents. The keys are used to describe the data, and values represent the actual data. To increase the readability of our dictionary, we have placed each key-value pair on a separate line.

As we have seen before, we can get the value associated with a key by writing the dictionary name with the key inside the square brackets.

```
>>> student['name']
'John'
>>> student['city']
'Paris'
>>> student['grade']
KeyError: 'grade'
>>> student['marks'][1]
78
```

If the key we specify is not present in the dictionary, then we get a `KeyError`. 'grade' is not a key in the dictionary, so the expression `student['grade']` raises an error. The last expression will give us a value of 78 since `students['marks']` is a list, and we can use numeric indexing on that list.

We can see that by using a key as the index instead of an integer index, the code becomes more readable and self-documenting. Therefore, when item names are more meaningful than item positions, it makes more sense to have the items in a dictionary.

Here is the dictionary for the next data sample:

```
>>> prices = {'pencil': 10,
 'pen': 22,
 'eraser': 12,
```

```
 'sharpener': 13,
 'marker': 32
 }
```

Here, the product name is the key, and its associated price is the value. If you need to access the price of a marker, you can write `prices['marker']`. If you need to find the total price of 2 pencils, 3 markers, and 5 erasers, you can write this:

```
>>> total = 2 * prices['pencil'] + 3 *
prices['marker'] + 5 * prices['eraser']
```

We can use the built-in function `len` to find the length of the dictionary.

```
>>> len(prices)
5
```

The length of the dictionary is 5, and the `len` function returns the number of items in a dictionary, i.e., the number of key-value pairs.

The following dictionary is for the last table of the figure.

```
>>> salary = {'programmer': 10000,
 'manager': 20000,
 'accountant': 15000
 }
```

Here, keys represent the designation names, and values are the associated salaries.

So, when you have your data in a table, the best data structure for this type of data is a dictionary. We can extract any value from a dictionary by using the associated key inside the square brackets. This data structure is named so, as it resembles a real-life dictionary, in which there is a word and its associated definition; here, we have a key and its associated value. You associate a key with a value, which is also called an associative data structure. It is also known as mapping type since it maps keys to associated values.

Information lookup is faster in dictionaries. They allow faster access to values as there is no need to go through each item sequentially as in a list. Values can be easily located by directly going to the key. This is because of

the highly optimized hashing algorithm used to implement dictionaries. This is the reason why keys of mutable types are not allowed.

Before version 3.7, dictionaries were unordered structures, which means that the items in a dictionary would not necessarily be in the same order in which you defined or inserted them. When printing a dictionary, the items would not necessarily be displayed in the order in which they were defined. Python 3.7 onwards dictionaries are ordered data structures, and dictionary elements are guaranteed to be in insertion order. When you print a dictionary or iterate over it in a loop, you will see that the order of elements is the same in which they were defined or added to the dictionary.

Built-in functions like `max`, `min`, `sorted` work for dictionaries also, but all of them work for keys only. If you need to use them for values, you can do it using lambda functions which is discussed later in this book.

## 5.2 Adding new key-value pairs

The following assignment statement will add a new key-value pair to the dictionary.

```
d[k] = val
```

This will insert the key `k` with the value `val` in the dictionary `d`. Let us insert a new key-value pair in our `prices` dictionary.

```
>>> prices['ruler'] = 30
>>> prices
{'pencil': 10, 'pen': 22, 'eraser': 12,
 'sharpener': 13, 'marker': 32, 'ruler': 30}
```

We know that duplicate keys are not allowed in a dictionary. Let us see what happens when we try to add a new key-value pair and the key already exists in the dictionary.

```
>>> prices['pencil'] = 15
>>> prices
{'pencil': 15, 'pen': 22, 'eraser': 12,
 'sharpener': 13, 'marker': 32, 'ruler': 30}
```

We do not get any error; assigning a value to an existing dictionary key replaces the old value with the new value.

If in a dictionary literal, a key is specified more than once, then also the interpreter will not complain, and it will assign the last occurrence of value to the key.

```
>>> d = {'x': 1, 'y': 2, 'z': 3, 'x': 100}
>>> d
{'x': 100, 'y': 2, 'z': 3}
```

## 5.3 Modifying Values

In the previous section, we already saw how to change the value associated with a particular key. The following assignment will replace the old value associated with key `k` with the new value.

```
d[k] = val
```

Let us change the price of a pen in our `prices` dictionary.

```
>>> prices = {'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32}
>>> prices['pen'] = 25
>>> prices
{'pencil': 10, 'pen': 25, 'eraser': 12, 'sharpener': 13, 'marker': 32}
```

The old value 22 is replaced is replaced with the new value 25. The syntax for adding a new key-value pair and modifying existing values is the same. If the key is not present, the assignment statement `d[k] = val` will insert the key and the value in the dictionary, and if the key is present, it will update the value.

You can also use augmented assignment statements to change the values. For example, in the `salary` dictionary that we had written, suppose you want to increase the salary of the programmer and decrease the salary of the manager. You can write the following augmented assignment statements:

```
>>> salary = {'programmer': 10000, 'manager': 20000, 'accountant': 15000}
```

```
>>> salary['programmer'] += 1000
>>> salary['manager'] -= 1000
>>> salary
{'programmer': 11000, 'manager': 19000,
'accountant': 15000}
```

## 5.4 Getting a value from a key by using the get() method

We have seen that we can access individual values in a dictionary using the key as the index. If we have a dictionary named `d`, we can write `d[k]` to access the value associated with the key `k`. The problem with this approach is that if the key `k` is not present in the dictionary `d`, then a `KeyError` will be raised. To avoid this error, you can use the `get()` method. This method returns the associated value like `d[k]`, but if the key is not found, instead of raising an error, it returns `None`. You can specify another value to be returned instead of `None` if the key is not present. So, if you think there are any chances of the key not existing in the dictionary, it is better to use the `get` method instead of the square bracket notation.

|                            |                                                                                                                          |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>d.get(k)</code>      | Returns the value that is associated with key <code>k</code><br>If <code>k</code> not present, returns <code>None</code> |
| <code>d.get(k, val)</code> | Returns the value that is associated with key <code>k</code><br>If <code>k</code> not present, returns <code>val</code>  |

***Table 5.1: The get method***

The `get` method takes a key as the argument and returns the value associated with it. It takes an optional second argument, which is the value to be returned when the key does not exist.

```
>>> prices = {'pencil': 10, 'pen': 22, 'eraser':12,
'sharpener':13, 'marker':32}
>>> prices['pen']
22
>>> prices.get('pen')
```



22

```
>>> prices['stapler']
KeyError: 'stapler'
>>> prices.get('stapler')
```

When we used the `get` method on a non-existent key, nothing was printed on the prompt. If we use the `print` function, we can see that it returns `None`.

```
>>> print(prices.get('stapler'))
None
```

We can specify any other value to be returned instead of `None`.

```
>>> prices.get('stapler', 0)
0
>>> prices.get('stapler', 5)
5
```

## 5.5 Getting a value from a key by using the `setdefault()` method

The `setdefault` method also accesses the value from a key, but if the key is missing, it will add that key to the dictionary. The value for that key is set to `None` or you can provide your own value also.

|                                   |                                                                                                                                                                                                        |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>d.setdefault(k)</code>      | Returns the value that is associated with key <code>k</code><br>If <code>k</code> is not present, returns <code>None</code> and adds the key <code>k</code> to dictionary with value <code>None</code> |
| <code>d.setdefault(k, val)</code> | Returns the value that is associated with key <code>k</code><br>If <code>k</code> is not present, returns <code>val</code> and adds the key <code>k</code> to dictionary with value <code>val</code>   |

**Table 5.2:** *The `setdefault` method*

The `setdefault` method can take two arguments. The first argument is the key for which you want to retrieve the value. The second argument is optional. It is the value that will be assigned to the key instead of the default `None`.

```
>>> prices = {'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32}
>>> prices.setdefault('pen')
22
>>> prices.setdefault('stapler') # Returns None
>>> prices
{'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32, 'stapler': None}
```

We can see that the key is added with the value `None`. If we want the key to be added with a value other than `None`, we can specify that value.

```
>>> prices.setdefault('gum', 10)
10
>>> prices
{'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32, 'stapler': None, 'gum': 10}
```

## 5.6 Getting all keys, all values, and all key-value pairs

The following three methods return special list-like iterable objects called dictionary views. These objects are dynamic, so any changes in the dictionary are reflected in these objects.

|                         |                                                                                        |
|-------------------------|----------------------------------------------------------------------------------------|
| <code>d.keys()</code>   | Returns an object providing a view on keys of the dictionary <code>d</code>            |
| <code>d.values()</code> | Returns an object providing a view on values of the dictionary <code>d</code>          |
| <code>d.items()</code>  | Returns an object providing a view on keys and values of the dictionary <code>d</code> |

**Table 5.3:** *Methods to get all keys, values, and key-value pairs*

To get all the keys, use the `d.keys()` method; to get all the values, use the `d.values()` method; and to get all the key-value pairs, use the `d.items()` method.

```
>>> prices = {'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32}
>>> prices.keys()
dict_keys(['pencil', 'pen', 'eraser', 'sharpener', 'marker'])
>>> prices.values()
dict_values([10, 22, 12, 13, 32])
>>> prices.items()
dict_items([('pencil', 10), ('pen', 22), ('eraser', 12), ('sharpener', 13), ('marker', 32)])
```

The methods `keys()`, `values()`, and `items()` return a `dict_keys` object, `dict_values` object, and `dict_items` object. You can use the `list` function to convert these objects to `list` type if required.

```
>>> list(prices.keys())
['pencil', 'pen', 'eraser', 'sharpener', 'marker']
>>> list(prices.values())
[10, 22, 12, 13, 32]
>>> list(prices.items())
[('pencil', 10), ('pen', 22), ('eraser', 12), ('sharpener', 13), ('marker', 32)]
```

These three methods do not return lists to save the time and memory used in creating a list that might have no use. For large dictionaries, lists also will be large and hence will consume more space. These methods return a view object, and if you want a list, you can convert explicitly. The dictionary view objects are iterable, and we can use them in a for loop to process all the items of a dictionary. We will discuss this in [Chapter 7](#) that covers loops.

Python 3.8 onwards, the dictionary views are reversible. If we use the built-in `reversed` function, the keys and values will be iterated over in the reverse order of the insertion.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
```

```

{'a': 10, 'b': 20, 'c': 30}
>>> list(reversed(d))
['c', 'b', 'a']
>>> list(reversed(d.keys()))
['c', 'b', 'a']
>>> list(reversed(d.values()))
[30, 20, 10]
>>> list(reversed(d.items()))
[('c', 30), ('b', 20), ('a', 10)]

```

We can also use the `sorted` function on these views.

```

>>> sorted(d.keys())
['a', 'b', 'c']
>>> sorted(d.values())
[10, 20, 30]
>>> sorted(d.items())
[('a', 10), ('b', 20), ('c', 30)]

```

## 5.7 Checking for the existence of a key or a value in a dictionary

In the previous chapters, we saw that the `in` and `not in` operators can check whether a value exists in a list, tuple, or string. These operators can also check whether a key or a value exists in a dictionary. These membership operators can be used with the dictionary view objects to check for membership of keys and values.

|                                                   |                                                                                                          |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>x in d</code><br><code>x in d.keys()</code> | Returns True if <code>x</code> is present as a key in the dictionary <code>d</code> , otherwise False    |
| <code>x in d.values()</code>                      | Returns True if <code>x</code> is present as a value in the dictionary <code>d</code> , otherwise False  |
| <code>(k, val) in d.items()</code>                | Returns True if <code>(k, val)</code> pair is present in the dictionary <code>d</code> , otherwise False |

**Table 5.4:** *Checking for the existence of a key or a value in a dictionary*

If you want to know whether a key is present in the dictionary, you can simply write `x in d` or `x in d.keys()`. To check if `x` is present in the dictionary as a value, you can write `x in d.values()`. To check for a key-value pair, you can use the `items` method.

```
>>> prices = {'pencil': 10, 'pen': 22, 'eraser': 12, 'sharpener': 13, 'marker': 32}
>>> 'pen' in prices
True
>>> 'pen' in prices.keys()
True
>>> 100 in prices.values()
False
>>> 100 not in prices.values()
True
>>> 22 in prices.values()
True
>>> ('pencil', 10) in prices.items()
True
>>> ('pencil', 12) in prices.items()
False
```

## 5.8 Comparing dictionaries

The equality operators `==` and `!=` can be used to compare two dictionaries. The expression `d1==d2` will return `True` if the two dictionaries contain the same key-value pairs. We can also use the `keys()`, `values()`, and `items()` methods with these operators. The other comparison operators (`<`, `>`, `<=`, `>=`) are not defined for a dictionary.

```
>>> d1 = {'x': 1, 'y': 2, 'z': 3}
>>> d2 = {'x': 1, 'y': 2, 'z': 3}
```

```

>>> d3 = {'x': 100, 'y': 200, 'z': 300}
>>> d1 == d2
True
>>> d1 == d3
False
>>> d1.keys() == d3.keys()
True

```

## 5.9 Deleting key-value pairs from a dictionary

The `del` statement can be used to delete a key-value pair from the dictionary. `del d[k]` will remove both the key `k` and its associated value from the dictionary. If the key is not present, then a `KeyError` will be raised.

If you want to delete a key-value pair and store the deleted value in a variable, you can use the `pop` method. This method will delete the key-value pair, and it will return the value associated with the key. If the key is not present, then a `KeyError` will be raised. If you do not want the `KeyError` to be raised in case of missing key, you can send a second argument to the `pop` function, which will be returned if the key is not present. For example, the call `d.pop(k, -1)` will return `-1` if key `k` is not present.

|                            |                                                                                                                                  |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>del d[k]</code>      | Removes key <code>k</code> and its associated value from the dictionary <code>d</code>                                           |
| <code>d.pop(k)</code>      | Removes key <code>k</code> and its associated value from the dictionary <code>d</code> , and returns the value <code>d[k]</code> |
| <code>d.pop(k, val)</code> | Returns <code>val</code> if key <code>k</code> is not present in the dictionary                                                  |

**Table 5.5:** *Deleting key-value pairs*

In lists, you could use the `pop()` method without any argument, and it would give you the last element, but in dictionaries, you cannot use `pop()` without an argument.

```

>>> prices = {'pencil': 10, 'pen': 22, 'eraser': 12, 'gum': 13, 'marker': 32, 'ruler': 30}

```

```
>>> del prices['marker']
>>> prices
{'pencil': 10, 'pen': 22, 'eraser': 12, 'gum': 13,
'ruler': 30}
```

The key value pair corresponding to the key 'marker' has been removed.

```
>>> x = prices.pop('pencil')
>>> x
10
```

```
>>> prices
{'pen': 22, 'eraser': 12, 'gum': 13, 'ruler': 30}
```

This call to method `pop` removed the key-value pair ('pencil', 10), and it also returned the value 10, which we stored in the variable `x`.

We will get a `KeyError` if the key is not present in the dictionary.

```
>>> prices.pop('book')
KeyError: 'book'
```

To avoid this `KeyError`, we can send a second argument.

```
>>> prices.pop('book', 0)
0
```

Now, 0 is returned from the `pop` method for the non-existent key 'book'.

The method `popitem()` removes and returns a random key-value tuple pair from the dictionary.

```
>>> prices
{'pen': 22, 'eraser': 12, 'gum': 13, 'ruler': 30}
>>> prices.popitem()
('ruler', 30)
>>> prices
{'pen': 22, 'eraser': 12, 'gum': 13}
>>> prices.popitem()
('gum', 13)
```

```
>>> prices
{'pen': 22, 'eraser': 12}
```

The method `clear()` removes all key-value pairs from the dictionary and makes it empty.

```
>>> prices.clear()
>>> prices
{}
```

If you try to empty the dictionary by assigning an empty dictionary, then there can be problems if other variables are referring to the dictionary.

```
>>> prices = {}
```

This will not delete all the items from the dictionary. This will create a new empty dictionary and make the name `prices` refer to that empty dictionary.

## 5.10 Creating a Dictionary at run time

We have seen how to create dictionaries by writing dictionary literals. All key-value pairs are written inside the curly braces, with keys and values separated by colons.

```
prices = {'pen': 22, 'eraser': 12, 'gum': 13,
'ruler': 30}
```

Creating a dictionary this way is fine if you know the initial data beforehand. If you want to create your dictionary dynamically at run time, you can start by creating an empty dictionary and adding key-value pairs to it. Let us start with an empty dictionary and add key-value pairs to it.

```
prices = {}
fruit = input('Enter name of a fruit: ')
price = int(input('Enter its price: '))
prices[fruit] = price
fruit = input('Enter name of a fruit: ')
price = int(input('Enter its price: '))
prices[fruit] = price
```



```
fruit = input('Enter name of a fruit: ')
price = int(input('Enter its price: '))
prices[fruit] = price
print(prices)
```

### **Sample run -**

```
Enter name of a fruit: Apple
Enter its price: 50
Enter name of a fruit: Banana
Enter its price: 25
Enter name of a fruit: Guava
Enter its price: 27
{'Apple': 50, 'Banana': 25, 'Guava': 27}
```

In this program, we must repeat the code for entering key-value pairs. In [Chapter 7](#), we will learn how to avoid this code repetition and input multiple keys and values in a dictionary using loops. We can let the users enter the keys and values, or the input can be taken from a file and stored in the dictionary at run time.

## **5.11 Creating a dictionary from existing data by using dict()**

We can create dictionaries from existing data that is present in other data structures like lists or tuples. The `dict()` function can be used to convert a sequence of two value sequences into a dictionary. The first item in the sequence is used as the key, and the second item is the value. For example, suppose we have a list of 2 item lists.

```
>>> list1 = [['a', 1], ['b', 2], ['c', 3]]
>>> d1 = dict(list1)
>>> d1
{'a': 1, 'b': 2, 'c': 3}
```

We sent the list to the `dict` function and got a dictionary. The first item in each inner list is taken as the key, and the second item is taken as the value. Instead of a list of lists, we can have a list of tuples, a tuple of tuples, or a tuple of lists. The main thing is that the length of inner sequences should be exactly 2, as they represent the key-value pairs.

```
>>> t1 = ('x', 4), ('y', 5), ('z', 6)
>>> d2 = dict(t1)
>>> d2
{'x': 4, 'y': 5, 'z': 6}
>>> t2 = ['x', 4], ['y', 5], ['z', 6]
>>> d3 = dict(t2)
>>> print(d3)
{'x': 4, 'y': 5, 'z': 6}
>>> d4 = dict(['x', 4], ['y', 5], ['z', 6])
>>> d4
{'x': 4, 'y': 5, 'z': 6}
```

While defining the two tuples, `t1` and `t2`, we have omitted the enclosing parentheses, but when we send the tuple literal directly inside the `dict` function, we have to put the parentheses.

We can send a list or tuple of strings of length 2, as strings are also sequences.

```
>>> d5 = dict(['X1', 'Y2', 'Z3'])
>>> d5
{'X': '1', 'Y': '2', 'Z': '3'}
```

The first character from the string is taken as the key, and the second character as the value.

We can also create a new dictionary by passing keyword arguments to the `dict` function.

```
>>> d6 = dict(pencil=12, eraser=45, sharpener=30)
>>> d6
{'pencil': 12, 'eraser': 45, 'sharpener': 30}
```

The names will become the keys, and the values will become the corresponding values in the dictionary. But this way you can have only strings as keys.

Dictionaries can also be created by zipping together two sequences. For example, suppose we have the following two lists: the first one contains country names, and the other one contains corresponding capitals at the same offsets.

```
>>> countries = ['France', 'Austria', 'Japan',
 'India']
>>> capitals = ['Paris', 'Vienna', 'Tokyo', 'New
Delhi']
```

We can create a dictionary from these two lists using the `zip` function. This function walks through multiple sequences and creates tuples from items at the same offsets.

```
>>> d7 = dict(zip(countries, capitals))
>>> d7
{'France': 'Paris', 'Austria': 'Vienna', 'Japan':
'Tokyo', 'India': 'New Delhi'}
```

The two lists were sent to the `zip` function and its return value was sent to the `dict` function, and we get a dictionary with keys from the first list and values from the other list.

An empty dictionary can also be created by using the `dict` function, although using empty braces is the preferred style.

```
>>> d8 = dict()
>>> d8
{}
```

## 5.12 Creating a dictionary by using the `fromkeys()` method

`dict.fromkeys(I, value)` creates a new dictionary with keys from iterable `I` and values set to `value`. If `value` is not provided, then the

values for all the keys are set to None.

Suppose we have a list named `stationery`, and we send it to the `fromkeys` method, with the second argument as 0.

```
>>> stationery = ['pencil', 'marker', 'eraser',
'sharpener']
>>> prices = dict.fromkeys(stationery, 0)
>>> print(prices)
{ 'pencil': 0, 'marker': 0, 'eraser': 0,
 'sharpener': 0 }
```

We get this dictionary, in which keys are taken from the list, and the value for all the keys is set to 0. This method is generally used to create default dictionaries. If we do not provide the second argument, all values will be None.

```
>>> d1 = dict.fromkeys(stationery)
>>> print(d1)
{ 'pencil': None, 'marker': None, 'eraser': None,
 'sharpener': None }
>>> d2 = dict.fromkeys(range(7))
>>> print(d2)
{0: None, 1: None, 2: None, 3: None, 4: None, 5:
None, 6: None}
```

This method is usually directly called as `dict.fromkeys()` rather than being called on an existing dictionary. It can also be called using an empty dictionary literal.

```
>>> prices = {}.fromkeys(stationery, 0)
>>> print(prices)
{ 'pencil': 0, 'marker': 0, 'eraser': 0,
 'sharpener': 0 }
```

There is another way of creating a dictionary called dictionary comprehension expression, which we will discuss later in a separate chapter.

## 5.13 Combining dictionaries

We can copy the key-value pairs of a dictionary into another dictionary by using the `update` method. The call `d.update(d1)` merges all entries of dictionary `d1` into dictionary `d`. If there is a key that is present in both dictionaries, the value in dictionary `d` is overwritten by the value in dictionary `d1`.

```
>>> prices1 = {'apple': 10, 'mango': 15, 'banana': 20}
>>> prices2 = {'grapes': 25, 'banana': 17, 'papaya': 12}
>>> prices1.update(prices2)
>>> prices1
{'apple': 10, 'mango': 15, 'banana': 17, 'grapes': 25, 'papaya': 12}
```

All the entries of `prices2` are added to `prices1`. The key `'banana'` was present in both dictionaries, and we can see that the value in `prices1` was overwritten by the value in `prices2`.

The `update` method can also accept an iterable object of key-value pairs.

```
>>> L = [['guava', 23], ['fig', 30], ['mango', 25]]
>>> prices1.update(L)
>>> prices1
{'apple': 10, 'mango': 25, 'banana': 17, 'grapes': 25, 'papaya': 12, 'guava': 23, 'fig': 30}
```

The `update` method can accept keyword arguments also.

```
>>> prices1.update(lemon=15, melon=65)
>>> prices1
{'apple': 10, 'mango': 25, 'banana': 17, 'grapes': 25, 'papaya': 12, 'guava': 23, 'fig': 30, 'lemon': 15, 'melon': 65}
```

Python 3.9 onwards, the two operators `|` and `|=` are also available for the `dict` type.

```
>>> d1 = {'x': 1, 'y': 2, 'c': 8}
>>> d2 = {'a': 3, 'b': 4, 'c': 7}
>>> d1 | d2
{'x': 1, 'y': 2, 'c': 7, 'a': 3, 'b': 4}
```

The expression `d1 | d2` returns a new dictionary with the merged keys and values of `d1` and `d2`. The values of `d2` get priority if `d1` and `d2` have the same keys.

```
>>> d2 | d1
{'a': 3, 'b': 4, 'c': 8, 'x': 1, 'y': 2}
>>> d1 |= d2
>>> d1
{'x': 1, 'y': 2, 'c': 7, 'a': 3, 'b': 4}
```

## 5.14 Nesting of dictionaries

The values in a dictionary can be of any type; they can be of type `dict` also. When we have a dictionary as a value inside a dictionary, we get a nested dictionary. Let us understand this with the help of an example. We have seen the following dictionary that was used to describe a student record. In this dictionary, we have used a list to represent the marks.

```
student = {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
 'age': 21,
 'marks': [89, 78, 91],
 'is_sporty': True
 }
```

We use the following expressions to access marks:

```
student['marks'][0] -> Marks in first subject
```

`student['marks'][1]` -> Marks in second subject  
`student['marks'][2]` -> Marks in third subject

Suppose we want to make the marks field more informative and want to store marks with the name of the subject. For that, we can use a dictionary instead of using a list.

In [Figure 5.2](#), we have a dictionary inside a dictionary. In the `student` dictionary, the value for the key named `'marks'` is again a dictionary. In the inner dictionary, keys are subject names, and values are marks in those subjects. To access the inner dictionary, we will write `student['marks']` because this dictionary is the value corresponding to the `'marks'` key. To access values inside this dictionary, we will use keys `'Maths'`, `'Physics'`, and `'Chemistry'` as indexes.

So, to get marks in Maths, we will write `student['marks']`  
`['Maths']`. Similarly, to get Physics marks, we can write  
`student['marks']['Physics']`.

|           |                                        |
|-----------|----------------------------------------|
| name      | John                                   |
| gender    | M                                      |
| city      | Paris                                  |
| age       | 21                                     |
| marks     | Maths 89<br>Physics 78<br>Chemistry 91 |
| is_sporty | True                                   |

```
student = {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
 'age': 21,
 'marks': {'Maths': 89,
 'Physics': 78,
 'Chemistry': 91
 },
 'is_sporty': True
 }
```

**Figure 5.2:** Nested dictionary

```
>>> student['marks']
{'Maths': 89, 'Physics': 78, 'Chemistry': 91}
>>> student['marks']['Maths']
89
>>> student['marks']['Physics']
78
>>> student['marks']['Chemistry']
```

The dictionary we have just defined represents the record of a single student. There could be records of many students that we would want to store in our program. Instead of giving a name to each record, like `student1`, `student2`, or `student3`, we can store all these records in a collection type. Storing them in a list will result in a longer time for value retrieval, so we can store them in a dictionary, as a dictionary search is more efficient. All the students have a unique student id, so we can have a collection in the form of another dictionary, where the keys are the id numbers and values are the dictionaries that represent student records.

|        |           |                                        |
|--------|-----------|----------------------------------------|
| 105416 | name      | John                                   |
|        | gender    | M                                      |
|        | city      | Paris                                  |
|        | age       | 21                                     |
|        | marks     | Maths 89<br>Physics 78<br>Chemistry 91 |
|        | is_sporty | True                                   |
| 144547 | name      | Dev                                    |
|        | gender    | M                                      |
|        | city      | London                                 |
|        | age       | 23                                     |
|        | marks     | Maths 88<br>Physics 77<br>Chemistry 98 |
|        | is_sporty | False                                  |
| 132399 | name      | Mary                                   |
|        | gender    | F                                      |
|        | city      | Paris                                  |
|        | age       | 22                                     |
|        | marks     | Maths 99<br>Physics 87<br>Chemistry 88 |
|        | is_sporty | True                                   |

```
students = {105416: {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
 'age': 21,
 'marks': {'Maths': 89,
 'Physics': 78,
 'Chemistry': 91},
 'is_sporty': True},
 144547: {'name': 'Dev',
 'gender': 'M',
 'city': 'London',
 'age': 23,
 'marks': {'Maths': 88,
 'Physics': 77,
 'Chemistry': 98},
 'is_sporty': False},
 132399: {'name': 'Mary',
 'gender': 'F',
 'city': 'Paris',
 'age': 22,
 'marks': {'Maths': 99,
 'Physics': 87,
 'Chemistry': 88},
 'is_sporty': True}}
```

**Figure 5.3:** Nested dictionary

Now, we can access a student's details by using the student id. The expression `students[105416]` will give us the first dictionary, `students[144547]` will give us the second dictionary, and



`students[132399]` will give us the third dictionary. To access the data of student with the id 144547, we can write:

```
>>> students[144547]
{'name': 'Dev', 'gender': 'M', 'city': 'London',
 'age': 23, 'marks': {'Maths': 88, 'Physics': 77,
 'Chemistry': 98}, 'is_sporty': False}
```

To access the name of the student with the id 105416, we can write:

```
>>> students[105416]['name']
'John'
```

The following expression gives the chemistry marks of the student with the id 132399:

```
>>> students[132399]['marks']['Chemistry']
88
```

These types of dictionaries will generally not be written in the literal form in the program. The details will be entered by the user or taken from a file. To print these types of complex nested structures in a readable form, we can use the `pp` function from `pprint` module.

```
>>> import pprint
>>> pprint.pp(students)
>>> pprint.pp(students[144547])
```

## 5.15 Aliasing and Shallow vs. Deep Copy

We have discussed aliasing, shallow copying, and deep copying in the previous chapter. Dictionaries are also mutable structures like lists, so you need to be careful about aliasing, and when you have a nested dictionary structure, you need to perform a deep copy instead of a shallow copy. In this section, we will take some examples to understand these concepts in the context of dictionaries.

Suppose you have a fruit shop, and this dictionary stores the fruit names and prices available in your shop.

```
>>> shop1_prices = {'apple': 200, 'mango': 250,
'banana': 100, 'grapes': 90}
```

Now, you open another shop and the same items are available in that shop also, so you decide to copy this dictionary.

```
>>> shop2_prices = shop1_prices
```

This dictionary shop2\_prices stores the prices of fruits in shop2.

```
>>> shop2_prices
{'apple': 200, 'mango': 250, 'banana': 100,
'grapes': 90}
```

This shop2 is a new shop, so sales are less, resulting in a huge stock of apples and bananas, so you decide to drop the price of apples and bananas in shop2.

```
>>> shop2_prices['apple'] -= 40
>>> shop2_prices['banana'] -= 20
>>> shop2_prices
{'apple': 160, 'mango': 250, 'banana': 80,
'grapes': 90}
```

We can see that the prices are reduced in shop2. Now there is a customer in shop1- your old shop - who wants to buy 2 kg apples and 3 kg bananas. This is how you calculate the amount to be paid:

```
>>> bill = 2 * shop1_prices['apple'] + 3 *
shop1_prices['banana']
>>> bill
560
```

The customer pays ₹560, and you suffer a loss of 140 because of aliasing. 2 kg apple and 3 kg banana would cost ₹700 in shop1, but you got only ₹560. The culprit here is the statement shop2\_prices = shop1\_prices which made an alias instead of an independent copy.

We can check the ids of the two dictionaries.

```
>>> id(shop1_prices)
1688566781824
```

```
>>> id(shop2_prices)
1688566781824
```

No new object was created; there is only one dictionary object, and both `shop1_prices` and `shop2_prices` refer to it.

```
>>> shop1_prices
{'apple': 160, 'mango': 250, 'banana': 80,
'grapes': 90}
```

The prices were reduced for `shop1` also. Instead of the assignment statement, we should have used the dictionary `copy` method, as that would give us an independent copy of the dictionary.

```
>>> shop1_prices = {'apple': 200, 'mango': 250,
'banana': 100, 'grapes': 90}
>>> shop2_prices = shop1_prices.copy()
>>> shop2_prices['apple'] -= 40
>>> shop2_prices['banana'] -= 20
>>> bill = 2 * shop1_prices['apple'] + 3 *
shop1_prices['banana']
>>> bill
700
```

We can also use the `dict` function to get an independent copy.

```
>>> shop2_prices = dict(shop1_prices)
```

Now, suppose you have a software company also, and the following nested dictionary structure stores the salary of the employees. The salaries of programmers working with different languages are different.

```
>>> officel_salary = {'manager': 6000,
... 'web designer': 3000,
... 'programmer': {'Python': 5000,
'Java': 4000, 'C#': 4500}}
```

```
... }
```

You open another office, and from your fruit business experience, you know what problems aliasing can cause, so you do not make that mistake again. You make an independent copy by using the `copy` method.

```
>>> office2_salary = office1_salary.copy()
```

To be sure that you have independent objects, you can check the ids also.

```
>>> id(office1_salary)
```

```
2081864102592
```

```
>>> id(office2_salary)
```

```
2081828847232
```

ids are different, which means that we have separate dictionary objects. Python programmers in office1 are performing very well, so you decide to increase their salary.

```
>>> office1_salary['programmer']['Python'] += 500
```

You do not want anything to go wrong, so before printing the salary slips, you can check the two dictionaries. We will use the `pp` function from the `pprint` module to print these nested dictionaries in a readable form.

```
>>> import pprint
```

```
>>> pprint.pp(office1_salary)
```

```
{'manager': 6000,
 'web designer': 3000,
 'programmer': {'Python': 5500, 'Java': 4000, 'C#':
 4500}}
```

Python programmers in office1 now get 5500 instead of 5000 which is what we wanted.

```
>>> pprint.pp(office2_salary)
```

```
{'manager': 6000,
 'web designer': 3000,
 'programmer': {'Python': 5500, 'Java': 4000, 'C#':
 4500}}
```

The salary for Python programmers in office2 has also changed. How is this possible when we have made an independent copy using the `copy` method? The problem is that we have a nested structure, so we need a deep copy instead of a shallow copy. The `copy` method gives us a shallow copy. We can check the ids of inner dictionaries.

```
>>> id(office1_salary['programmer'])
2081827346304
>>> id(office2_salary['programmer'])
2081827346304
```

The same dictionary is shared by both objects. To perform a deep copy, we need to use the `deepcopy` function from the `copy` module.

```
>>> from copy import deepcopy
>>> office2_salary = deepcopy(office1_salary)
>>> id(office2_salary['programmer'])
1975443905984
>>> id(office1_salary['programmer'])
1975399634816
```

## 5.16 Introduction to sets

Searching in a list or tuple takes a long time if they are big in size, and if they have to be searched multiple times, it can lead to poor performance. Another constraint with lists is that they store duplicate values. In some cases, we might need to store only unique values. We can make our list store only unique values, but that will not be efficient since whenever we insert a new value, we have to sequentially scan all the values to check whether the value is already present.

The set data structure is suitable for these types of situations. When you want to store a collection of unique values for faster lookup, you can use a set. Sets are internally implemented in such a way that they can be searched very quickly and they automatically eliminate duplicate entries. However, there are some limitations of sets; the values will not be stored in any

particular order, and you can store only values of immutable type. Let us discuss the definition and syntax of defining sets.

A set is an unordered mutable collection of immutable and unique objects. Sets are unordered, so sets do not maintain any order among their elements. So, they are not of sequence type. Sets are mutable, meaning an object of type set can be changed. We can replace existing elements of the set, add new elements, or remove elements from the set. A set is a collection of immutable objects, meaning it can contain objects of only immutable types like integers, strings, or tuples. It cannot contain mutable type elements like lists or dictionaries. The elements need not be of the same types; a set can contain elements of different types.

The most important point about sets is that it is a collection of unique objects, meaning duplicate elements are not allowed in a set. So, you can see that elements of a set are like keys of a dictionary. They have to be immutable and unique. Here are a few examples of set literals.

```
>>> big_cities = {'London', 'Paris', 'Bangalore',
'Tokyo'}
>>> primes = {2, 3, 5, 7, 11, 13, 17, 19}
>>> colors = {'red', 'blue', 'yellow', 'black',
'white'}
```

On the right side of the assignment, we have a set literal that is assigned to a variable name. The elements of a set are placed inside curly braces and are separated by commas.

Like lists, tuples, and dictionaries, sets are also referential structures, which means that they contain references to objects. The elements are not in any order; there is nothing like the first element or the second element. You can think of a set as just a bag of unique values.

In sequences like strings, lists, and tuples, the elements are ordered so they can be identified by their position; we could access an individual element by applying a numeric index. In dictionaries, elements are identified by keys, so there we can access an element by using a key as the index. But in sets, elements are neither ordered nor there are any keys, so we cannot use indexing to access an individual element of a set. Sets do not support indexing or slicing as they do not have an inherent order.

The most common operation performed on a set is testing the existence of an item. For that, we can use the membership operators `in` and `not in`.

```
>>> 'Paris' in big_cities
True
>>> city = 'Perth'
>>> city not in big_cities
True
>>> number = 11
>>> number not in primes
False
```

You can write these types of expressions, and they will return `True` or `False` depending on whether the given item is present in the set or not. Testing for membership is faster in a set as compared to a list or tuples.

In our example sets, we have created a set of primes of the first 8 prime numbers so we can check whether a given number exists in this set or not. If we want to do something with, suppose, the fifth prime number, we cannot do it because the set has not stored them in order, so we do not know which is the fifth prime number. If we have such a requirement, we must make a list or tuple, which are ordered structures.

Now the question is, how will you know that you need to create a set in your program? You can create a set when you have a collection of values whose order does not matter, and in your program, you will just need to know whether a value belongs to that collection or not. So, when you want to store some unique values whose order does not matter but search efficiency matters, you can use a set.

## 5.17 Creating a set

The call to `set()` function will create an empty set.

```
s = set()
```

We have only one way to create an empty set because empty curly braces `{ }` are used to create an empty dictionary.

```
s = {} # an empty dictionary will be created
```

Dictionaries were introduced in Python before sets, so this syntax is taken by dictionary, and you have to make an empty set by using the `set()` function only.

We can use the `set` function to create sets from other types like strings, lists, tuples, and dictionaries. The duplicate values are discarded in this process as a set can have only unique values.

```
>>> print(set('HELLO'))
{'E', 'L', 'H', 'O'}
>>> L = [1, 2, 3, 1, 2, 3, 4, 5, 4, 3, 2, 1]
>>> print(set(L))
{1, 2, 3, 4, 5}
>>> t = (20, 30, 40, 30, 20)
>>> print(set(t))
{40, 20, 30}
```

In all these examples, we can see that the duplicate values are discarded, and only unique values are placed in the set. The original order is not necessarily preserved, as sets are unordered structures.

If you try to convert a dictionary to a set, you get only a set of keys; the values are lost. To get the set of values, you have to use the `values` method of `dict` type.

```
>>> d = {1:'a', 2:'b', 3:'c', 4:'a', 5:'c'}
>>> set(d)
{1, 2, 3, 4, 5}
>>> set(d.values())
{'a', 'b', 'c'}
```

Sets can be created by using the `range` function also.

```
>>> odds = set(range(1, 20, 2))
>>> odds
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```



If you have an existing list or tuple and you want to search it without duplicates efficiently, you can convert it to a set. If you want to filter out duplicates from a list, you can convert it to set and then back to the list again but the order will be lost in this process.

We can use sets for performing order-neutral equality tests. You can convert a list to a set before testing for equality.

```
>>> L1 = [1, 2, 3, 4]
>>> L2 = [3, 2, 4, 1]
>>> print(L1 == L2)
False
>>> print(set(L1) == set(L2))
True
```

## 5.18 Adding and Removing elements

Here are some methods for adding and removing elements from a set.

|                           |                                                                                                             |
|---------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>s.add(x)</code>     | Adds a new item <code>x</code> to the set <code>s</code>                                                    |
| <code>s.pop()</code>      | Removes an arbitrary element from <code>s</code>                                                            |
| <code>s.remove(x)</code>  | Removes <code>x</code> from set <code>s</code> , raises <code>KeyError</code> if <code>x</code> not present |
| <code>s.discard(x)</code> | Removes <code>x</code> from set <code>s</code> , no effect if <code>x</code> not present                    |
| <code>s.clear()</code>    | Removes all elements from <code>s</code>                                                                    |

**Table 5.6:** *Adding and removing elements from a set*

The `add()` method is used to add a new item to the set, and if this item `x` is already present in the set, then there is no effect. The item that is to be added should be of immutable type. The `pop()` method removes an arbitrary element from `s`. If the set is empty, then it raises a `KeyError`. To remove a specified item, use either `remove()` or `discard()`. Both will remove the element `x` from the set; they just differ in their behavior when `x` is not present. `remove()` will raise `KeyError`, while `discard()` will have no effect if the element to be removed is not present. The `clear()` method removes all elements from the set, and the `copy()` method returns a copy of set `s`. Here are some examples of these methods:

```

>>> cities = {'Cairo', 'Mumbai', 'Agra',
'Bengaluru', 'Rome', 'Perth', 'Bareilly', 'Bern'}
>>> cities.add('Delhi')
>>> cities
{'Bern', 'Agra', 'Mumbai', 'Cairo', 'Perth',
'Bengaluru', 'Bareilly', 'Rome', 'Delhi'}
>>> cities.remove('Bern')
>>> cities
{'Agra', 'Mumbai', 'Cairo', 'Perth', 'Bengaluru',
'Bareilly', 'Rome', 'Delhi'}
>>> cities.remove('Tokyo')
KeyError: 'Tokyo'
>>> cities.discard('Tokyo')
>>> print(cities)
{'Agra', 'Mumbai', 'Cairo', 'Perth', 'Bengaluru',
'Bareilly', 'Rome', 'Delhi'}
>>> city = cities.pop()
>>> print(city)
Agra
>>> print(cities)
{'Mumbai', 'Cairo', 'Perth', 'Bengaluru',
'Bareilly', 'Rome', 'Delhi'}

```

The `copy` method of `set` type returns a shallow copy of the set. The built in functions like `len()`, `sum()`, `max()`, `min()`, `sorted()`, `all()`, `any()` work on sets also.

Some special operations can be performed on sets. These operations correspond to the set theory of mathematics. You might have studied operations like union, intersection, and difference in set theory in maths. These operations are supported by sets of Python. These operations are different from the operations that we have seen in other collections like lists or tuples. These special set operations can make your code shorter and more readable. We will discuss these operations in the next two sections.

## 5.19 Comparing sets

In mathematics, two sets are considered to be disjoint if they have no element in common. In Python, we have the method `isdisjoint` to check whether two sets have any elements in common. The expression `s1.isdisjoint(s2)` returns `True` if sets `s1` and `s2` have no elements in common, otherwise it returns `False`.

```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {5, 6, 7, 3}
>>> s3 = {10, 20, 30, 50}
>>> s1.isdisjoint(s2)
False
>>> s1.isdisjoint(s3)
True
```

Two sets are considered equal if each element of one set is contained in the other set. We can use the equality operators `==` and `!=` with sets to check their equality.

```
>>> s1 = {1, 2, 4, 3, 6}
>>> s2 = {6, 4, 3, 2, 1}
>>> s1 == s2
True
```

Sets `s1` and `s2` have identical elements, so they are equal.

We know that the elements of a set are not in any particular order, so the meaning of operators `<`, `>`, `<=`, and `>=` operators is different from what it is for lists and tuples. These operators are based on the mathematical notion of subsets and supersets. In set theory, a set `s1` is a subset of another set `s2` if every element of `s1` is present in set `s2`. In Python, the method `issubset` and the operator `<=` are used to check for subset relationship.

If `s1` is a subset of `s2`, then we can say that `s2` is a superset of `s1`.

So, a set `s1` is a superset of another set `s2`; if `s1` contains every element of `s2`; it can contain extra elements also. In Python, the method `issuperset()` and the operator `>=` are used to check for superset relationship.

|                                                            |                                                                                    |
|------------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>s1.issubset(s2)</code> or <code>s1 &lt;= s2</code>   | Returns True if <code>s1</code> is a subset of <code>s2</code> , otherwise False   |
| <code>s1.issuperset(s2)</code> or <code>s1 &gt;= s2</code> | Returns True if <code>s1</code> is a superset of <code>s2</code> , otherwise False |

**Table 5.7:** Comparing sets

The method `issubset()` or the expression `s1 <= s2` returns True if every item in set `s1` is also present in set `s2`. Otherwise, it returns False. The method `issuperset()` or the expression `s1 >= s2` returns True if every item in set `s2` is also present in set `s1`. Otherwise, it returns False. If two sets `s1` and `s2` are equal, then `s1` is a subset of `s2`, and it is also a superset of `s2`.

The methods `issubset()` and `issuperset()` can also accept sequential types as arguments. But if we use the operators `<=` and `>=`, then we can compare two sets only. Here are some examples:

```
>>> s1 = {'x', 'y', 'z', 'a', 'b'}
>>> s2 = {'x', 'y', 'z', 'a', 'b', 'c', 'd'}
>>> s3 = {'a', 'b', 'x', 'y', 'z'}
>>> s1.issubset(s2)
True
>>> s1 <= s2
True
>>> s2.issuperset(s1)
True
>>> s2 >= s1
True
```

Sets `s1` and `s3` are equal, so both are considered subset and superset of each other.

```
>>> s1 <= s3
True
```

```
>>> s1 >= s3
```

```
True
```

Every set is considered a subset and superset of itself.

```
>>> s1 <= s1
```

```
True
```

```
>>> s1 >= s1
```

```
True
```

The operators `<` and `>` are used to check for proper subset and proper superset. A proper subset is like a subset, but the two sets cannot be equal. Similarly, a proper superset is like a superset, but the two sets cannot be equal. There are no equivalent methods corresponding to these operators.

```
>>> s1 < s2
```

```
True
```

```
>>> s1 < s3
```

```
False
```

```
>>> s3 > s1
```

```
False
```

The expression `s1 < s3` returns `False` because `s1` is not a proper subset of `s3`. A proper subset is any subset that is not equal to the set. The set `s1` is a subset of `s3`, but since it is equal to `s3`, it is not a proper subset. The expression `s3 > s1` returns `False` because `s3` is not a proper superset of `s1`. A proper superset is any superset that is not equal to the set. This set `s3` is a superset of `s1`, but since it is equal to `s3`, it is not a proper superset.

## 5.20 Union, intersection, and difference of sets

Python `set` type is different from other types that we have seen because it supports all standard mathematical set operations like union, intersection, and difference. These mathematical operations can be used in different types of programming situations. We can use either a method or an equivalent operator for any of these set operations.

`s1.union(s2,s3,...)`     or   `s1 | s2 | s3`

Returns a new set containing all items of sets `s1, s2, s3.....`

`s1.intersection(s2,s3,...)`     or   `s1 & s2 & s3`

Returns a new set containing only the common items of sets `s1, s2, s3.....`

`s1.difference(s2)`     or   `s1 - s2`

Returns a new set containing all items of `s1` that are not in `s2`

`s1.symmetric_difference(s2)`     or   `s1 ^ s2`

Returns a new set containing items that are in set `s1` or `s2`, but not both; so, it actually returns elements of both sets that are not in the intersection.

Let us see some examples. We have taken three sets named `python_programmers`, `java_programmers` and `c_programmers`.

```
>>> python_programmers = {'Nick', 'Sam', 'Peter',
 'Mary', 'Alan', 'Rose', 'Zara', 'Max'}
```

```
>>> java_programmers = {'Ted', 'Sandy', 'Peter',
 'Alan', 'Ross', 'Max', 'Ruby'}
```

```
>>> c_programmers = {'Nick', 'Ted', 'Peter',
 'Abbie', 'Julie', 'Jack', 'Jill'}
```

Suppose we want a set of programmers who can work both in Java and in Python. So, we need names that are common to both the sets `python_programmers` and `java_programmers`, and for that, we can use the `intersection` method or the `&` operator.

```
>>>
python_programmers.intersection(java_programmers)
{'Alan', 'Max', 'Peter'}
```

```
>>> python_programmers & java_programmers
{'Alan', 'Max', 'Peter'}
```

If we want a set of programmers who can work in all the three languages, Java, Python, and C, then we need names common to all three sets, so we can write this:

```
>>>
python_programmers.intersection(java_programmers,
c_programmers)
{'Peter'}
>>> python_programmers & java_programmers &
c_programmers
{'Peter'}
```

Now, suppose we want a set of those programmers who can program either in C or in Python. So now we need a union of the two sets `c_programmers` and `python_programmers`.

```
>>> c_programmers.union(python_programmers)
{'Julie', 'Max', 'Jill', 'Rose', 'Sam', 'Peter',
'Abbie', 'Nick', 'Jack', 'Zara', 'Ted', 'Alan',
'Mary'}
>>> c_programmers | python_programmers
{'Julie', 'Max', 'Jill', 'Rose', 'Sam', 'Peter',
'Abbie', 'Nick', 'Jack', 'Zara', 'Ted', 'Alan',
'Mary'}
```

Now suppose we need a set of programmers who can program in Python but who do not know Java. So, we want the names of all those programmers who are in the `python_programmers` set but not in `java_programmers` set. For getting this set we can use the difference method or the equivalent operator.

```
>>> python_programmers - java_programmers
{'Rose', 'Sam', 'Nick', 'Zara', 'Mary'}
>>> python_programmers.difference(java_programmers)
{'Rose', 'Sam', 'Nick', 'Zara', 'Mary'}
```

The following two expressions will give the names of those programmers who are in the `python_programmers` set but not in `java_programmers` set or `c_programmers` set.

```
>>> python_programmers.difference(java_programmers,
c_programmers)
{'Rose', 'Sam', 'Zara', 'Mary'}
```

```
>>> python_programmers - java_programmers-
c_programmers
{'Mary', 'Rose', 'Zara', 'Sam'}
```

Now, let us use the `symmetric_difference` method on the two sets `python_programmers` and `java_programmers`.

```
>>>
python_programmers.symmetric_difference(java_programmers)
{'Sandy', 'Rose', 'Nick', 'Zara', 'Ruby', 'Mary',
'Ross', 'Sam', 'Ted'}
>>> python_programmers ^ java_programmers
{'Sandy', 'Rose', 'Nick', 'Zara', 'Ruby', 'Mary',
'Ross', 'Sam', 'Ted'}
```

This symmetric difference gives us a set of those programmers who can program either in Java or in Python but not both. So, this set contains all names in set `python_programmers` and in set `java_programmers` minus the names that are common to both sets.

All these methods and operations were non-mutating. They do not make any in-place changes in the set, which calls them; they always return a new set. We can see that the original sets have not been changed.

```
>>> python_programmers
{'Max', 'Rose', 'Sam', 'Peter', 'Nick', 'Zara',
'Alan', 'Mary'}
>>> java_programmers
{'Ross', 'Sandy', 'Max', 'Ted', 'Ruby', 'Alan',
'Peter'}
>>> c_programmers
{'Julie', 'Abbie', 'Nick', 'Jack', 'Ted', 'Jill',
'Peter'}
```

The four nonmutating methods that we have seen have mutating equivalents also. Here are the equivalent mutating methods and their equivalent operators:



|                                                 |                           |
|-------------------------------------------------|---------------------------|
| <code>s1.update(s2)</code>                      | <code>s1  = s2</code>     |
| <code>s1.intersection_update(s2)</code>         | <code>s1 &amp;= s2</code> |
| <code>s1.difference_update(s2)</code>           | <code>s1 -= s2</code>     |
| <code>s1.symmetric_difference_update(s2)</code> | <code>s1 ^= s2</code>     |

**Table 5.8:** *Mutating methods and operators for sets*

These mutating methods perform the same operation as their non-mutating counterparts, but they perform the operation in-place, which means that they change the set which calls them instead of returning a new set. All these methods return `None`. These four mutating methods are also accessible using the augmented assignment syntax.

```
>>>
python_programmers.intersection_update(java_programmers)
>>> python_programmers
{'Alan', 'Max', 'Peter'}
```

We can see that the set `python_programmers` has been changed, and it now contains the intersection of the two sets. The same effect can be achieved by using the augmented assignment syntax.

```
>>> java_programmers &= c_programmers
>>> java_programmers
{'Peter', 'Ted'}
```

Now the set `java_programmers` has changed, and it contains the intersection of the two sets `java_programmers` and `python_programmers`. Similarly, the mutating equivalents of other methods also make in-place changes.

If you want to perform these operations on other types like lists, string, or tuple, you can do so by converting them to set.

```
>>> s1 = 'Welcome'
>>> s2 = 'Come here'
>>> set(s1) - set(s2)
{'l', 'W', 'c'}
```

```

>>> s3 = 'What is in a name'
>>> s4 = 'There are letters in a name'
>>> set(s3.split()) - set(s4.split())
{'is', 'What'}
>>> x = [1, 2, 3, 4, 5]
>>> y = [3, 4, 5, 6, 7]
>>> set(x) | set(y)
{1, 2, 3, 4, 5, 6, 7}

```

The set operations for finding union, intersection, and difference can be used on view objects also that are returned by dictionary methods `keys()` and `items()`. For example, if you have two dictionaries `d1` and `d2`, the expression `d1.items() & d2.items()` will give the key-value pairs that are common to both dictionaries. The expression `d1.keys() - d2.keys()` will give you the keys that are in `d1` but not in `d2`.

```

>>> d1 = {'a': 15, 'b': 22, 'c': 35, 'd': 24}
>>> d2 = {'a': 15, 'b': 20, 'x': 29, 'd': 24}
>>> d1.items() & d2.items()
{('d', 24), ('a', 15)}
>>> d1.keys() - d2.keys()
{'c'}

```

You do not need to convert the output of these methods to set and then perform these operations. This facility is not available for the `values()` method of the dictionary.

## 5.21 Frozenset

A frozenset is the immutable version of a set. Once a frozenset is created, it cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys. You can think of a frozenset as a read-only set. frozensets support the same operations as sets, except the operations that change the contents. So, methods like `add`, `remove`, `pop`, and `update` are not applicable for frozensets. You can create a frozenset by

sending an iterable to the `frozenset` function. In the following examples, we have created frozensets from a set, list, and string.

```
>>> weekdays = frozenset({'Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday'})
>>> weekend = frozenset(['Saturday', 'Sunday'])
>>> vowels = frozenset('aeiou')
>>> type weekdays
<class 'frozenset'>
>>> weekdays
frozenset({'Thursday', 'Monday', 'Tuesday',
'Wednesday', 'Friday'})
>>> weekend
frozenset({'Saturday', 'Sunday'})
>>> vowels
frozenset({'a', 'i', 'o', 'e', 'u'})
```

When you need an immutable version of a set, you can use a `frozenset`.

## Exercise

1. Which of these cannot be used as a key in a dictionary?  
(A) String  
(B) Integer  
(C) List
2. Only immutable types can be used as values in a dictionary.  
(A) True (B) False
3. Which one of these will make changes in the dictionary object referenced by name `d`?  
(A) `d.clear()` (C) Both  
(B) `d = {}` (D) None of these

4. A tuple can be used as key of a dictionary if it contains references to

- (A) Only mutable objects
- (B) Only immutable objects
- (C) Both mutable and immutable objects

5. What is wrong with this dictionary?

```
{5: 'a', 2: 'j', 9: 'y', 6: 'y', 5: 's'}
```

- (A) `int` type cannot be used as a key
- (B) There is a duplicate key

6. `d = {'apple': 100, 'banana': 75, 'mango': 80}`

What is the value of `len(d)`?

- (A) 3 (B) 6

7. `d = {'apple': 100, 'banana': 75, 'mango': 80}`

What will be the value of expression

```
d.get('grapes', -1)
```

- (A) None
- (B) -1
- (C) Only single argument allowed in `get()`
- (D) `KeyError` is raised

8. `d = {'apple': 100, 'banana': 75, 'mango': 80}`

What happens when you misspell a key while changing its value.

```
d['aple'] = 95
```

- (A) `KeyError`
- (B) new key 'aple' is added to the dictionary

9. As in strings and lists, the expression `d[:]` represents copy of a dictionary `d`.
- (A) True (B) False
10. Which one of these will give all key value pairs of a dictionary?
- (A) `d.elements()`
- (B) `d.items()`
- (C) `d.pairs()`
11. How will you check whether a value `v` is present in a dictionary `d`?
- (A) `v in d`
- (B) `v in d.values()`
- (C) Both
12. `d = {123: 'Dev', 342: 'Raj', '567': 'John', 898: 'Sam'}`
- What will the following expression return?
- `(123, 'Raj') in d.items()`
- (A) True
- (B) False
- (C) Error is raised
13. In a dictionary, the method `pop()` cannot be used without an argument.
- (A) True (B) False
14. Which key-value pair does the method `popitem()` remove?
- (A) First pair
- (B) Last pair
- (C) Random pair
15. If you want to delete a key-value pair from a dictionary and print the deleted value, what will you use?

- (A) `del` statement
- (B) `pop()` method
- (C) anyone

16. `d = dict(zip('xyz', [4, 5, 6]))`

Dictionary `d` is -

- (A) `{ 'xyz': [4, 5, 6] }`
- (B) `{ 'x': 4, 'y': 5, 'z': 6 }`

17. Which one of these cannot be used to create a dictionary using `dict()`?

- (A) `[['a', 11], ['b', 6], ['c', 7]]`
- (B) `[['a', 'x', 4], ['b', 'y', 5], ['c', 'z', 6]]`
- (C) `['AB', 'CD', 'EF']`

18. `d = { 'a': 1, 'b': 2, 'c': 3 }`

What will be the dictionary `d` after `d.update({})`?

- (A) `d` becomes empty
- (B) `d` is not changed

19. What is length of this dictionary?

`d = dict.fromkeys('HELLO', None)`

- (A) 1
- (B) 4
- (C) 5

20. What does `{ }` create?

- (A) empty dictionary
- (B) empty set
- (C) empty frozenset

21. If you want to create a dictionary from an iterable, such that all the values in the dictionary are same, which method will you use?

- (A) `items()`

(B) `setdefault()`

(C) `fromkeys()`

22. Is it possible to create a set of sets?

(A) Yes (B) No

23. \_\_\_\_\_ are very commonly used to test for membership of an item.

(A) Dictionaries

(B) Sets

(C) Tuples

24. Is it possible to create a set of frozensets?

(A) Yes (B) No

25. Which method is used to remove an element randomly from a set?

(A) `pop`

(B) `popitem`

(C) `remove`

26. What is the length of the following set `s`?

```
s = set('cookbook')
```

(A) 4

(B) 6

(C) 8

27. `s = set(1, 2, 3, 1, 3)`

What will be the value of `s`?

(A) `{1, 2, 3}`

(B) `{1, 2, 3, 1, 3}`

(C) this assignment statement raises `TypeError`

28. Which method will remove an element from a set without giving any error if the element is not present?

(A) `remove` (B) `delete`

(C) pop (D) discard

29. Which data structure will you use when you want to store things and order is important? Contents might change.

(A) list (B) tuple

(C) set (D) dictionary

30. If you want to store unique values and do not care about the order in which they are stored, you can use a \_\_\_\_\_.

(A) list

(B) tuple

(C) set

31. When you have some ordered data that you know will not change, you can store it in a \_\_\_\_\_

(A) list (B) tuple

(C) set (D) dictionary

32. Use \_\_\_\_\_ when you want to attach some information to values and want to access that value by the information not by a numeric index.

(A) list (B) tuple

(C) set (D) dictionary

33. Which of these does not allow duplicate values?

(A) tuples (B) frozensets

34. Which of these is not a sequence?

(A) list

(B) tuple

(C) set

35. When you have a table-like data, which data structure would you use?

(A) list (B) tuple

(C) set (D) dictionary



36. Which one of these cannot be used as a key in a dictionary?
- (A) string
  - (B) list
  - (C) tuple
37. \_\_\_\_\_ should be used for static sequences of elements.
- (A) list
  - (B) tuple
  - (C) set
38. \_\_\_\_\_ are generally used when the data is labelled.
- (A) Dictionaries (B) Lists
39. Dictionaries and sets can retrieve a value in constant time regardless of the number of entries.
- (A) True (B) False
40. String is a mutable sequence of characters.
- (A) True (B) False
41. `V = 'aeiou'`  
`L = ['a', 'e', 'i', 'o', 'u']`  
`S = {'a', 'e', 'i', 'o', 'u'}`
- Which of the following expression is most efficient?
- (A) `ch in V`
  - (B) `ch in L`
  - (C) `ch in S`
42. Set is \_\_\_\_\_ unordered collection of unique \_\_\_\_\_ objects.
- (A) immutable, immutable
  - (B) mutable, mutable
  - (C) immutable, mutable
  - (D) mutable, immutable

What will be the output of the code given in questions 43 to 55?

43. `d = {(3, 4): 100, (5, 3): 20, (4, 5): 32}`

`print(d[5, 3])`

44. `d = dict(zip('good', range(4)))`

`print(d)`

45. `d = {'x': 10, 'y': 20, 'x': 33, 'z': 40}`

`print(d['x'])`

46. `s = {1, 2, 3, 4}`

`print(s[1])`

47. `s1 = {3, 2, 4}`

`s2 = {3, 2, 4}`

`print(s1 < s2)`

48. `s1 = {3, 2, 4}`

`s2 = {3, 2, 4}`

`print(s1 <= s2)`

49. `d = {'a': 1, 'b': 2, 'c': 2}`

`s = set(d)`

`print(s)`

50. `x = {'hello'}`

`y = set('hello')`

`print(x, y)`

51. `d = {'a': [1, 2, 3], 'b': 10, 'c': 12}`

`d2 = d`

`d['a'][1] = 55`

`d['b'] = 99`

`print(d2)`

```
52. d = {'a': [1, 2, 3], 'b': 10, 'c': 12}
 d2 = d.copy()
 d['a'][1] = 55
 d['b'] = 99
 print(d2)
```

```
53. a = 5
 D = {'k1': a, 'k2': 60, 'k3': 70}
 a = 10
 print(D['k1'])
```

```
54. x = frozenset(['a', 'b', 'c'])
 y = {'d', 'e'}
 x |= y
 print(x)
```

```
55. x = y = z = 0
 x = 2
 print(x, y, z, end=' ')
 d1 = d2 = d3 = {}
 d1['a'] = 2
 print(d1, d2, d3)
```

56. On the interactive prompt, create an empty dictionary named currency and then add these key-value pairs to it.

```
'India': 'Rupee'
'UK': 'Pound'
'Japan': 'Yen'
'Austria': 'Euro'
'Bangladesh': 'Taka'
```

57. From the `currency` dictionary created in the previous question, delete the entry related to key `'UK'`
58. Delete the entry related to key `'Japan'` and store the return value in another variable named `c`.
59. Add a new entry in the dictionary with the key `'Switzerland'` and the value `'Swiss Franc'`.
60. Change the value for key `'India'` from `'Rupee'` to `'Indian Rupee'`
61. Delete a random key-value pair from the dictionary.
62. Use appropriate methods to get lists of all keys, all values, and all key-value pairs of the `currency` dictionary.

63. Given the following dictionary:

```
fruits_prices = {'apple': 100, 'banana': 75,
 'mango': 80}
```

Use the appropriate method to access the values associated with keys `'apple'` and `'grapes'`. If the key is not present in the dictionary, then it should be added with value 0.

64. Create a dictionary named `login` from the following list named `names`.

```
names = ['John', 'Sam', 'Marie', 'Anne']
```

The elements of this list should become the keys of the dictionary, and values associated with all keys should be `None`.

65. Given these 2 lists:

```
designation = ['programmer', 'manager',
 'accountant']
```

```
salary = [4000, 5000, 3000]
```

Create the following dictionary from the above two lists.

```
{'programmer':4000, 'manager':5000,
 'accountant':3000}
```

66. Given these 3 lists:

```
python_books = ['Learn Python', 'Programming
in Python', 'Python for beginners']
```

```
cplusplus_books = ['C++ in depth', 'C++
Programming']
```

```
java_books = ['Java Programming', 'Learn
Java']
```

Write a dictionary named `books` with the strings `'python'`, `'c++'` and `'java'` as keys and these lists as values. Thus, when you write `books['java']` you get the list of java books and similarly for other keys.

67. Given these 2 dictionaries:

```
book_prices = {'Learn ABC': 150, 'Learn 123':
200, 'Rhymes': 300, 'Cursive Writing': 250}
```

```
new_stock = {'Stories': 350, 'Poems': 290,
'Spellings': 200}
```

Add all the key-value pairs of `new_stock` to `book_prices`.

68. Create this dictionary by using `range()` function and `fromkeys()` method.

```
{1000: None, 2000: None, 3000: None, 4000:
None, 5000: None, 6000: None, 7000: None,
8000: None, 9000: None}
```

69. In the following nested dictionary, how will you access the last name of the student?

```
student = {'name': {'first': 'John',
 'last': 'Mark'},
 'marks': 98,
 'age': 20
}
```

70. From this dictionary `d`, create a list that contains all the keys in sorted order.

```
d = {2: 300, 8: 900, 7: 800, 1: 100}
```

71. In the following dictionary, key is an integer which represents the student id, and value is list type which contains marks of the student in three subjects.

```
marks = {2234: [99, 23, 56], 2135: [67, 56, 68], 2199: [78, 89, 66]}
```

Write an expression to get total marks of student with student id 2135.

72. In the previous chapter, we saw how to use a list of lists to represent a matrix. We used two indices to access an element of the matrix( for example `matrix[1][4]`). If a matrix is sparse, then we can save space by using a dictionary to implement it. A matrix is sparse, if it has many zero values in it. For example, this is a sparse matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix}$$

*Figure 5.4: Sparse matrix*

Create a dictionary named `matrix` which stores only non-zero values of this matrix. Use a tuple of row and column numbers as the key.

73. In the implementation of the matrix that we did in the last question, if we try to access any element of matrix that is zero, we will get an error. For example, if we write `matrix[1,2]` or `matrix[2,0]`, we will get an error. This is because there is no key in the dictionary corresponding to zero elements of the matrix. Tuples  $(1,2)$  or  $(2,0)$  are not present as keys in the dictionary. How will you solve this problem?
74. Input two strings `s1` and `s2`, and then create a list that contains all the common characters of the two input strings.

75. From the following two strings, find all words common in both strings. Extract words from the string by splitting on spaces.

```
string1 = 'Life has no remote, get up and
change it yourself'
```

```
string2 = 'Life has no ctrl+Z'
```

76. How will you count the number of unique items in a list?
77. Create a new list by filtering out all the duplicates from the following list by using the `set` function.

```
L = [12, 44, 46, 32, 12, 43, 55, 86, 43]
```

Will the order of the original list be preserved if you use this approach to filter out duplicates?

78. Enter a string and create 2 sets named `v` and `c`, where `v` is a set of vowels present in the string and `c` is a set of consonants present in the string.
79. How can you perform order-neutral equality tests in lists and strings using sets? The following two lists `L1` and `L2` have the same elements, only the order is different, so when you perform an order neutral equality test on these two lists, they are considered equal.

```
L1 = [1, 2, 3, 4]
```

```
L2 = [2, 3, 1, 4]
```

This test just checks whether both of them contain the same elements.

80. How will you find out all the elements of list `L1` that are not in `L2`.

```
L1 = [1, 2, 3, 7]
```

```
L2 = [2, 3, 4, 5]
```

81. How will you find all the common characters in three strings `s1`, `s2`, and `s3`

Use the following two sets for questions 82 to 87

```
toppers = {'id11', 'id23', 'id34', 'id45',
'id77', 'id12', 'id89', 'id56', 'id55',
'id19'}
```

```
champions = {'id19', 'id23', 'id78', 'id99',
'id79', 'id13', 'id56', 'id45', 'id80'}
```

The set `toppers` is a set of roll numbers of academic toppers of the school, and `champions` is a set of roll numbers of sports champions of the school.

82. From the set of toppers, remove the student with roll number `'id11'`.
83. From the set of champions, add two students with roll numbers `'id46'` and `'id20'`.
84. Find a set of all the toppers who are not champions.
85. Find a set of all the champions who are not toppers.
86. Find a set of all students who are champions as well as toppers.
87. Find a set of all students who are either champions or toppers.



# Conditional Execution

---

## 6

The control flow of a program is the order in which the code written in the program executes. Normally, the program executes from top to bottom with one statement executed at a time. This is called sequential control. All the programs we have written have been executed this way: top to bottom and one statement at a time. This normal flow of control is changed by control structures, which can be either selection control structures or iterative control structures.

In Python, selection is supported by the `if` statement, and iteration is supported by the `while` statement and `for` statement. The `if` statement is a conditional statement, meaning we can use it to process our code conditionally. The two iterative structures, `while` statement and `for` statement are called loops as they are used to repeatedly execute a section of code.

In this chapter, we will learn about the `if` statement, and in the next two chapters, we will learn about loops.

## 6.1 if statement

While solving a problem in real life, we often need to make decisions and act accordingly. Similar situations arise in programming also; we will want our program to make decisions and perform different operations based on those decisions. As in most other languages, in Python also, decision making or conditional execution is done with the help of an `if` statement. By using an `if` statement, you can make your program behave differently in different

situations. It gives your program the ability to make decisions and perform actions based on those decisions.

When you need to execute some statements only if a certain condition holds, you can use an `if` statement. Here is the syntax and flowchart of an `if` statement:

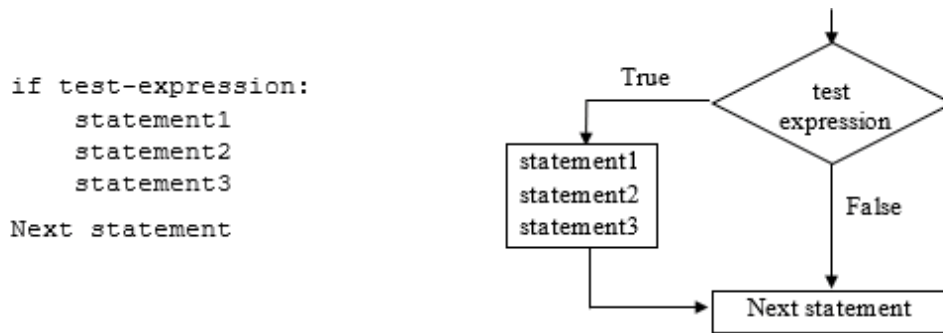


*Figure 6.1: if statement*

We have the `if` keyword followed by a test expression, and then we have a colon. The test expression is a Boolean expression, and therefore, it can be either `True` or `False`; it is often called the `if` condition. After the colon, we have the statement block, which will be executed when the test expression is `True`. Each statement in this block should be indented by the same length from the `if` line. We have seen earlier that Python uses indentation to identify a block.

If the test expression evaluates to `True`, the statements inside the block will be executed, and then the next statement after the `if` statement will be executed. If the test expression evaluates to `False`, the statements inside the block will be skipped, and the next statement will be executed. The flowchart clarifies why `if` statement is also called a branching statement. Let us see an example:

```
n1 = int(input('Enter a number : '))
n2 = int(input('Enter a number : '))
print(n1 + n2, end = ' ')
print(n1 - n2, end = ' ')
print(n1 * n2, end = ' ')
```

```
print(n1 / n2, end = ' ')\nprint(n1 // n2, end = ' ')\nprint(n1 % n2, end = ' ')\nprint(n1 ** n2, end = ' ')
```

### **Sample Run-**

Enter a number : 14

Enter a number : 4

18 10 56 3.5 3 2 38416

This code executes sequentially; two numbers are entered, and then all the statements are executed in order, one by one. We want the three statements that print `n1/n2`, `n1//n2`, and `n1%n2` to be executed only when the value of `n2` is not equal to zero because if the value of `n2` is zero, we will get a division by zero error. We want the three statements to be executed conditionally, so we will write them inside an `if` statement.

```
n1 = int(input('Enter a number : '))\nn2 = int(input('Enter a number : '))\nprint(n1 + n2, end=' ')\nprint(n1 - n2, end=' ')\nprint(n1 * n2, end=' ')\nif n2 != 0:\n print(n1 / n2, end=' ')\n print(n1 // n2, end=' ')\n print(n1 % n2, end=' ')\nprint(n1 ** n2, end=' ')
```

### **Sample Run 1-**

Enter a number : 10

Enter a number : 5

```
15 5 50 2.0 2 0 100000
```

### ***Sample Run 2-***

```
Enter a number : 10
```

```
Enter a number : 0
```

```
10 10 0 1
```

In the `if` statement written in this program, `n2 != 0` is the test expression, and the three indented statements form the `if` block. When the program is executed and 5 is entered for the variable `n2`, the condition `n2 != 0` is `True`, so the three statements inside `if` block are executed. When 0 is entered for `n2`, the condition `n2 != 0` becomes `False`, so the three statements inside the `if` block are not executed. The execution of the `if` block depends on the `if` condition. The rest of the statements outside the `if` statement will always execute.

This is the first time we have seen a block. The syntax for defining blocks is common for all the control structures and even for functions. A block is also called a suite in Python, and it is a group of statements grouped together through indentation. To specify the boundaries of a block, Python uses indentation instead of curly braces or some keywords like `begin` or `end` that are used in other languages.

In most languages, indentation is used just to enhance readability; it is not compulsory and does not affect the logic of the program. Python uses indentation for grouping together statements, so Python actually forces the programmer to write uniform and readable code.

You can have any number of statements inside a block; there is no limit, but there should be at least one statement. The colon marks the start of the statement block, and the first unindented statement marks the end of the block. The block finishes when the indentation decreases. The exact amount of indent may vary, but the indentation should be consistent. The recommended indent is 4 spaces. It is not a good idea to use tabs or mix tabs and spaces while indenting. Mixing tabs with spaces can result in errors, even though it might look correct on the screen.

So, we have seen how the `if` statement supports conditional execution; the statements inside the `if` block will be executed only if the condition is `True`.

Otherwise, they will be skipped. Now let us see some small programs.

The following program uses an `if` statement to test whether a number `n1` is divisible by another number `n2`.

```
n1 = int(input('Enter a number : '))
n2 = int(input('Enter a number : '))
if n1 % n2 == 0:
 print('n1 is divisible by n2')
```

`n1` will be divisible by `n2` if by dividing `n1` by `n2`, the remainder comes out to be zero. When the number `n1` is divisible by `n2`, the condition `n1 % n2 == 0` will be `True`, and the `print` call will execute. When `n1` is not divisible by `n2`, the condition `n1 % n2 == 0` will be `False`, and the `print` call will not execute.

While typing the `if` statement, you will notice that when you put a colon after the condition and then press Enter, the cursor goes to the next line leaving some space. This is because IDLE knows that a colon means a new block is going to start, so it automatically indents the next line. Most of the IDEs will do this automatic indenting.

Instead of `n2`, if we write `2` in the condition, we are checking divisibility by `2`, and if a number is divisible by `2`, it is an even number.

```
n1 = int(input('Enter a number : '))
if n1 % 2 == 0:
 print('n1 is even')
```

You can combine multiple conditions using the three logical operators `and`, `or`, and `not`. The following `if` condition uses the `and` operator to check if both `n1` and `n2` are even.

```
if n1 % 2 == 0 and n2 % 2 == 0:
 print('Both n1 and n2 are even')
```

The test expression will be `True` when both the expressions in it are `True`. So, the `print` call will be executed only when both `n1` and `n2` are even. Similarly, we can use `or` and `not` operators in our conditions.

If we want to perform some action when any one of the two conditions is True, we can combine the conditions using the `or` operator.

```
age = int(input('Enter age : '))
if age < 5 or age > 80:
 print('Entry prohibited')
```

To check whether a value is present in a list, tuple, string, or dictionary, we can use the `in` and `not in` operators.

```
athletes = ['Ram', 'Sam', 'Shyam', 'Abhi', 'Adi']
student = input('Enter student name : ')
if student in athletes:
 print('You are awarded a scholarship')
failed_students = ['Pam', 'Sam', 'Ron', 'Ted']
student = input('Enter student name: ')
if student not in failed_students:
 print('You are promoted')
```

If you are checking the equality of a variable multiple times, you can replace the `or` operators with the `in` operator and a set. Here is an example:

```
if error_code == 400 or error_code == 404 or
error_code == 301:
 print('Bad error')
if error_code in {400, 404, 301}:
 print('Bad error')
```

The second version is more concise than the first one. We could have used a list here, but using a set is better as searching is more efficient in it.

In the next program, we will check whether a string is a palindrome. A palindrome is a word or a phrase that reads the same forwards or backward, for example, 'madam,' 'refer,' and 'level' - these all are palindromes.

```
s = input('Enter a string : ')
```

```
if s == s[::-1]:
 print(f'{s} is a palindrome')
```

A string will be a palindrome if the reverse of the string is the same as the string. In Python, we can easily find out the reverse of a string by writing the expression `s[::-1]`.

If you want to execute compound statements like `if` statement and `for` statement on the interactive prompt, you need to enter a blank line after entering the code. This means that you have to press Enter twice to execute the compound statement.

```
>>> s = 'madam'
>>> if s == s[::-1]:
... print(f'{s} is a palindrome')
...
...
madam is a palindrome
```

As we have seen in [Chapter 2](#) when we enter a multiline statement on the interactive prompt, the prompt changes from `>>>` to three dots(`...`), which is the line continuation prompt.

## 6.2 else clause in if statement

```
if test-expression:
 statement1
 statement2
 statement3
else:
 statementA
 statementB
 statementC
Next statement
```

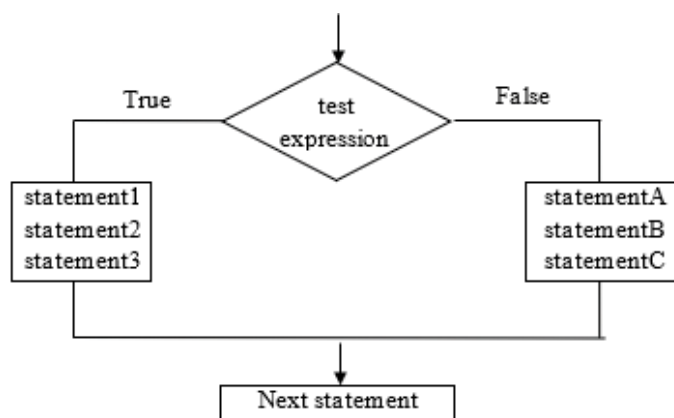


**Figure 6.2:** *if statement with else clause*

In the `if` statement, you can also add an `else` clause in which you can write the statements that you want to be executed when the test expression is `False`.

The `else` keyword is followed by a colon and should be aligned with the keyword `if`. All the statements in the `else` block should be indented by the same amount.

If the test expression is `True`, then the `if` block is executed; otherwise, the `else` block is executed. We have seen these two `if` statements in the previous section.

```
if n % 2 == 0:
 print('n is even')
if s == s[::-1]:
 print(s, 'is a palindrome')
```

Let us write the `else` clause for both of them.

```
n = int(input('Enter a number : '))
if n % 2 == 0:
 print('n is even')
else:
 print('n is odd')
```

### **Sample Run 1-**

```
Enter a number : 3
n is odd
```

### **Sample Run 2-**

```
Enter a number : 8
n is even
s = input('Enter a string : ')
if s == s[::-1]:
```



```
 print(f'{s} is a palindrome')
else:
 print(f'{s} is not a palindrome')
```

### ***Sample Run 1-***

```
Enter a string : refer
refer is a palindrome
```

### ***Sample Run 2-***

```
Enter a string : learn
learn is not a palindrome
```

## **6.3 Nested if statements**

`if` statements can be nested, which means that you can have an `if` statement inside another `if` statement. We have seen that this is the syntax of an `if` statement with an `else` clause.

```
if test-expression:
 statement1
 statement2
 statement3
else:
 statementA
 statementB
 statementC
```

Next statement

Inside the `if` block or the `else` block, we can have any type of Python statement; it can be an `if` statement also.

```

if test-expression:
 if test-expression2:
 blockA
 else:
 blockB
else:
 statementA
 statementB
 statementC

```

Next statement

Here, we have another `if` statement inside the `if` block. In the `else` block also, we could write the `if` statement. Let us see an example program.

```

s = input('Enter a string : ')
if s == s[::-1]:
 print(f'{s} is a palindrome')
else:
 print(f'{s} is not a palindrome')

```

We have seen this program before. Now, suppose we do not want to print only the message that `s` is a palindrome; we also want to check whether it is big palindrome or a small palindrome. If the length is less than 4, we will call it a small palindrome, otherwise, we will call it a big palindrome. Now, in the `if` block, instead of the statement that includes a `print` call, we will write another `if` statement.

```

s = input('Enter a string : ')
if s == s[::-1]:
 if len(s) < 5:
 print(f'{s} is a small palindrome')

```

```
 else:
 print(f'{s} is a big palindrome')
else:
 print(f'{s} is not a palindrome')
```

We have two `if` statements with `else` clauses. The first `else` goes with the inner `if`, and the second `else` goes with the outer `if`; the indentation makes it all clear. Nested statements have different levels of indentation. Here are some sample runs of this program:

### ***Sample Run 1-***

```
Enter a string : malayalam
malayalam is a big palindrome
```

### ***Sample Run 2-***

```
Enter a string : maths
maths is not a palindrome
```

### ***Sample Run 3-***

```
Enter a string : noon
noon is a small palindrome
```

Let us see one more example. We have this piece of code where we enter the marks of a student and decide whether the student has got an A grade.

```
marks = int(input('Enter marks : '))
if marks >= 70:
 print('Well done, you have got A grade')
else:
 print('Try to get A grade next time')
```

Now, we will add an `if` statement in both the `if` block and the `else` block.

```
marks = int(input('Enter marks : '))
```

```
if marks >= 70:
 print('Well done, you have got A grade')
 if marks >= 90:
 print('You are awarded a scholarship')
else:
 print('Try to get A grade next time')
 if marks < 40:
 print('You really need to work hard')
```

If marks are greater than or equal to 70, the student gets an A grade, and if he gets an A grade and his marks are greater than or equal to 90, he gets a scholarship. If a student does not get an A grade and his marks are less than 40, another print call will be executed. Here are some sample runs of this program:

#### ***Sample Run 1-***

```
Enter marks : 95
Well done, you have got A grade
You are awarded a scholarship
```

#### ***Sample Run 2-***

```
Enter marks : 80
Well done, you have got A grade
```

#### ***Sample Run 3-***

```
Enter marks : 35
Try to get A grade next time
You really need to work hard
```

#### ***Sample Run 4-***

```
Enter marks : 45
Try to get A grade next time
```

## 6.4 Multiway selection by using elif clause

Let us write a program in which we have to assign different grades to students depending on their marks. These are the criteria for assigning grades.

|                |                                     |
|----------------|-------------------------------------|
| Assign grade A | if marks $\geq$ 70                  |
| Assign grade B | if marks $\geq$ 60 and marks $<$ 70 |
| Assign grade C | if marks $\geq$ 50 and marks $<$ 60 |
| Assign grade D | if marks $\geq$ 40 and marks $<$ 50 |
| Assign grade E | if marks $<$ 40                     |

***Table 6.1***

Here is the program: first, we enter the marks, then write simple `if` conditions to assign these grades, and then print the grade.

```
marks = int(input('Enter marks - '))
if marks >= 70:
 grade = 'A'
if marks >= 60 and marks < 70:
 grade = 'B'
if marks >= 50 and marks < 60:
 grade = 'C'
if marks >= 40 and marks < 50:
 grade = 'D'
if marks < 40:
 grade = 'E'
print(f'Student gets {grade} grade')
```

This program works, but it is inefficient, as it makes the interpreter do unnecessary work. Let us discuss how.

Suppose the student scores 89 marks. The first condition `marks >= 70` evaluates to True, resulting in the grade being set to A. Since the conditions for the grades are mutually exclusive, there is no need to check the remaining conditions, as only one grade can be assigned. However, the interpreter will execute all the `if` statements one by one, even though the subsequent conditions are guaranteed to be False. The grade will remain A and will be printed at the end. We can avoid unnecessary checks done by the interpreter by conditionally executing the rest of the `if` statements. This can be done by using the nested `if` statements.

By using nested `if` statements, we can conditionally execute the subsequent checks based on the result of the first condition. If the first condition is True, we skip the other checks and directly assign the grade. If it is False, we proceed to the next condition until we find the appropriate grade. By doing this, we minimize the number of checks needed, thus making the program more efficient.

```
marks = int(input('Enter marks - '))

if marks >= 70:
 grade = 'A'
else:
 if marks >= 60:
 grade = 'B'
 else:
 if marks >= 50:
 grade = 'C'
 else:
 if marks >= 40:
 grade = 'D'
 else:
 grade = 'E'

print(f'Student gets {grade} grade')
```

If `marks >= 70`, `grade` is set to `A`. If this condition is `False`, it means that `marks` will be less than 70, and so in the `else` part, we will assign grades `B`, `C`, `D` or `E`. In the `else` part, we have written the `if` statement with condition `marks >= 60`. We need not check for `marks < 70` here because we will come here only when the condition `marks >= 70` fails, so `marks` will be less than 70. After this, we have an `else` clause for this `if` statement. In the `else` part, we will assign grades `C`, `D`, or `E`.

If `marks >= 50`, we assign the grade `C`. Again, we need not check the condition `marks < 60` because we will come here only if `marks` are less than 60. Next, we assign the grade `D` if `marks >= 40`. In the `else` part of this `if` statement, the grade will be `E` because control will come here when the condition `marks >= 40` fails, i.e., when `marks` are less than 40.

Now, let us see how this code is more efficient than the previous one. If a student gets 89 marks, then the condition `marks >= 70` will be `True`, and the statement `grade = 'A'` is executed. The whole `else` part is skipped and then the grade is printed. In the previous version, all the `if` statements were tested in this case.

Now, let us see what happens if the student gets 56 marks. The condition `marks >= 70` is `False`, so the `else` block will be executed, and in the `else` block, we have the `if` statement with the condition `marks >= 60`. This condition is also `False`, so the `else` block will be executed. In the `else` clause, we have the `if` statement with the condition `marks >= 50`. This condition is `True`, so the statement `grade = 'C'` is executed and the `else` part is skipped.

This structure is like an `else-if` chain or `else-if` ladder; it is used when we have multiple mutually exclusive conditions. There is excessive indentation involved here, which makes it difficult to read. Each time we add a nested `if`, we need to increase the indentation. Python has a solution for this in the form of an alternative syntax. You can replace `else` and the following `if` by the `elif` keyword.

```
marks = int(input('Enter marks - '))
if marks >= 70:
 grade = 'A'
```

```
elif marks >= 60:
 grade = 'B'
elif marks >= 50:
 grade = 'C'
elif marks >= 40:
 grade = 'D'
else:
 grade = 'E'

print(f'Student gets {grade} grade')
```

Each `elif` keyword should align with the `if` keyword and the final `else` keyword. The keyword `elif` is just a shortcut for `else if`. This code is similar to the previous one but is definitely more readable due to less indentation. In the previous code, we had many `if` statements, but here, we have only one `if` statement with multiple `elif` clauses and an `else` clause. That is why, in the previous code, all the blocks are indented differently, while here, all the blocks are at the same level of indentation.

The `elif` clause helps in multiway selection and reduces the amount of indentation that is to be done when we use the nested `if else` statements. The working of this construct is simple: each condition is checked in order; if the first condition is `True`, the statement block under it is executed, and other conditions are not checked. If the first condition is `False`, the second is checked; if the second is `False`, the third is checked, and so on. If any of them is `True`, the block under it executes, and the control comes out of the whole `if` statement. The final `else` block will be executed when none of the conditions is `True`, so it acts as the default case. Here is the syntax and flowchart of an `if` statement with `elif` and `else` clauses.



```

if expression1:
 statementBlockA
elif expression2:
 statementBlockB
elif expression3:
 statementBlockC
else:
 statementBlockD
Next statement

```

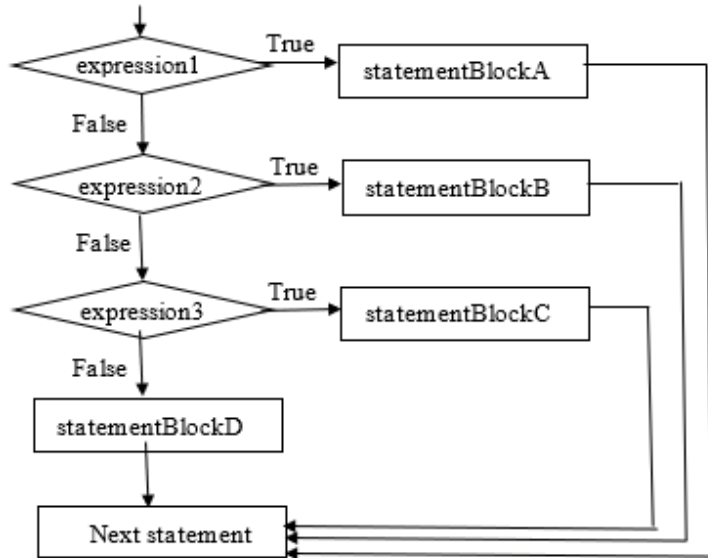


**Figure 6.3:** *if statement with elif clauses*

This `if...elif...else` statement implements multiway branching. From all the blocks, exactly one block will be executed.

If `expression1` is `True`, then `statementblockA` is executed, the `if` statement ends, and then the `Next statement` is executed. If `expression1` is `False`, then `expression2` is checked.

If `expression2` is `True`, then `statementblockB` is executed, the `if` statement ends, and then the `Next statement` is executed. If `expression2` is `False`, `expression3` is checked.

If `expression3` is `True`, then `statementblockC` is executed, the `if` statement ends, and then the `Next statement` is executed. If `expression3` is `False`, then `statementblockD` is executed, the `if` statement ends, then the `Next statement` executes.

So, when any one of the test expressions evaluates to `True`, the corresponding block is executed, the rest of the `elif` clauses are automatically skipped, and the whole `if` statement ends, and the execution resumes after the `if` statement. If none of the conditions evaluates to `True`, the block in the `else` clause will be executed.

You can have many `elif` statements, but there can be only one `else` clause, and it is optional. The `else` clause actually acts as the default or “catch-all” condition. When all the conditions are `False`, the block under `else` will be

executed. Although the `else` clause is optional, it is a good idea to write a final `else` in the `elif` ladder to ensure all the cases are covered.

While writing `if` statements with `elif` clauses, try to write those conditions first that are more likely to be true. The conditions less likely to be `True` should be towards the end.

Let us discuss one more example that uses `elif` clause. We have to enter a number and display if it is less than 100, more than 100, or equal to 100. These three are mutually exclusive conditions, which means that only one of them can be `True` at a time, and so we can use an `if` statement with `elif` clauses.

```
n = int(input('Enter a number : '))
if n < 100:
 print('Number is less than 100')
elif n > 100:
 print('Number is more than 100')
else:
 print('Number is equal to 100')
```

In our next program, we enter a single character, and the program prints what type of character it is. We have used the `else` clause to handle all the possibilities left.

```
ch = input('Enter a single character : ')
if len(ch) != 1:
 print('You did not enter a single character')
elif ch.isupper():
 print('Uppercase letter')
elif ch.islower():
 print('Lowercase letter')
elif ch.isnumeric():
```

```
 print('Number')
elif ch.isspace():
 print('Space')
else:
 print('Special character')
print('Bye')
```

We can also use `elif` clauses to create simple menu-based programs.

```
x = int(input('Enter a number : '))
y = int(input('Enter another number : '))
print('1. Add the two numbers')
print('2. Subtract first from second')
print('3. Subtract second from first')
print('4. Multiply the two numbers')
print('5. Divide first by second')
print('6. Divide second by first')
choice = input('Enter your choice : ')
if choice == '1':
 print(x + y)
elif choice == '2':
 print(y - x)
elif choice == '3':
 print(x - y)
elif choice == '4':
 print(x * y)
elif choice == '5':
```

```

 print(x / y)
elif choice == '6':
 print(y / x)
else:
 print('Wrong choice')

```

We enter two numbers and then display a menu, and then we ask the user to enter a choice. Depending on the choice entered by the user, we perform a particular operation using the `if` statement with `elif` clauses.

The `else` clause acts as the catch-all case, so if any number other than 1 to 6 is entered as the choice, the message ‘Wrong choice’ will be displayed.

We have to run this program again and again to execute different cases. We will discuss how to do this repeatedly in one run when we study loops.

## 6.5 Truthiness

We have seen that Python has a Boolean data type (`bool`), with only two values, True and False. Here are some expressions that evaluate to either True or False.

```

3 < 5 a >= b a is b not x x in listA

```

We know that we can use these expressions in a boolean context. For example, in the test expression of an `if` statement. In Python, we can use a non-boolean value also in a boolean context. For example, we could write `if` statements of this type.

```

if listA:
 print('Do something')
if dictA:
 print('Do something')
if x:
 print('Do something')

```

We are using non-boolean values in boolean context. Boolean context means a boolean value is needed from the expression. The `if` statement needs to know whether the test expression is True or False. So, there have to be rules for deciding what values are considered True and False. This brings us to the concept of truthiness. In Python, every value is either a truthy value or a falsy value. Truthy values are values that evaluate to True when used in a boolean context, and falsy values are values that evaluate to False when used in a Boolean context.

These values are considered falsy values in Python.

```
False None 0 0.0 0.0+0.0j '' []
() { } set ()
```

Boolean value False, None, 0 of any numeric type (integer, float, or complex) are considered falsy. Empty containers are considered false, so an empty string, empty list, empty tuple, empty dictionary, and empty set are all falsy values. Everything else is truthy; any non-zero number or non-empty container is evaluated to True. So, individual values or objects in Python have an inherent truthiness; they can be either truthy or falsy. User-defined objects can customize their truth value by providing a `__bool__()` method. We will discuss that later on.

In the `if` statement `if listA:`, if the list is empty, the condition will be considered False, and if it is not empty, it will be considered True. The same applies to the dictionary in the `if` statement `if dictA:`.

In the statement `if x:`, if the value of `x` is zero, the condition will be False; if it is anything non-zero, it will be considered True.

When a non-boolean value is used in a boolean context, Python evaluates the truthiness of that expression which means that it evaluates the value to either True or False. Thus, truthiness is the boolean meaning of a value.

You can explicitly check the truthiness of a value by using the `bool` built-in function. Pass the value to the `bool` function to see whether it evaluates to True or False.

```
>>> bool(0)

False

>>> bool(90)
```

```
True
```

```
>>> bool('')
```

```
False
```

```
>>> bool('ab')
```

```
True
```

```
>>> bool([])
```

```
False
```

```
>>> bool([1,2,3])
```

```
True
```

```
>>> bool('False')
```

```
True
```

The last one is True because 'False' is a non-empty string, not the Boolean value False. If we remove the quotes, it will be False.

```
>>> bool(False)
```

```
False
```

Similarly, `bool('0')` will be True as '0' is a non-empty string.

Whenever you have to perform an action, when some container is non-empty, or a number is non-zero, or a Boolean variable is True, you can just write `if x:` type of condition that contains only the variable. There is no need to write the full conditions.

```

if x != '':
 print('Do something')

if x != {}:
 print('Do something')

if x != []:
 print('Do something')

if x != 0:
 print('Do something')

if x == True:
 print('Do something')

if x is not None:
 print('Do something')

```

```

if x:
 print('Do something')

```

**Figure 6.4:** Concise way of writing if condition

This is a concise and more Pythonic way and is generally used by programmers. Similarly, when you have to do something when a container is empty or a number is zero, or a Boolean variable is False, or a variable is None, you can write the condition as `if not x:`

```

if x == '':
 print('Do something')

if x == {}:
 print('Do something')

if x == []:
 print('Do something')

if x == 0:
 print('Do something')

if x == False:
 print('Do something')

if x is None:
 print('Do something')

```

```

if not x:
 print('Do something')

```

**Figure 6.5:** Concise way of writing if condition

Let us discuss some examples:

```
name = input('Enter a name : ')
if name:
 print('Hello', name)
else:
 print('You did not enter anything')
```

Here, we are entering a string and assigning it to the variable name. If name is a non-empty string, the if condition will be True, and print('Hello', name) will execute, and if name is an empty string, the if condition will be False and print('You did not enter anything') will execute. Here is another example:

```
if listA:
 print('Not empty')
else:
 print('Empty')
```

Here we have a list, and we want to check if it is empty or not. If the list is not empty, the condition will be True and print('Not empty') will execute, and if the list is empty, print('Empty') will execute.

If we assign None to listA, then also print('Empty') will be executed, as None is considered falsy. To be more specific, we can write the conditions explicitly.

```
if listA is None:
 print('None')
elif listA == []:
 print('Empty')
else:
 print('Non Empty')
```



There are two built-in functions named `any` and `all` that can be used to check the truthiness of values inside an iterable like a list or tuple.

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| <code>all(x)</code> | Returns True if all elements in the iterable <code>x</code> are Truthy |
| <code>any(x)</code> | Returns True if any item in the iterable <code>x</code> is Truthy      |

***Table 6.2: Built-in functions `any` and `all`***

```
>>> help(all)
```

```
all(iterable, /)
```

```
 Return True if bool(x) is True for all values x
 in the iterable.
```

```
 If the iterable is empty, return True.
```

```
>>> help(any)
```

```
any(iterable, /)
```

```
 Return True if bool(x) is True for any x in the
 iterable.
```

```
 If the iterable is empty, return False.
```

```
>>> L = [1, 2, 0, 3]
```

```
>>> all(L)
```

```
False
```

```
>>> any(L)
```

```
True
```

```
>>> L = [0, 0, 0]
```

```
>>> any(L)
```

```
False
```

```
>>> L = [1, 2, 3]
```

```
>>> all(L)
```

```
True
```

## 6.6 Short circuit behavior of operators and or

If the value of an expression containing `and` or `or` can be determined by the first operand only, the second operand is not evaluated. Here is the truth table of `and` operator.

| operand1 and operand2 |     |       |    |       |
|-----------------------|-----|-------|----|-------|
| True                  | and | True  | => | True  |
| True                  | and | False | => | False |
| False                 | and | True  | => | False |
| False                 | and | False | => | False |

*Figure 6.6: Truth table of and operator*

We can see that if the first operand is False, the result is False regardless of the value of the second operand. If the first operand is True, the value of the second operand does not really matter since the result will be False anyway. And this is why the interpreter will not evaluate the second operand if the first one is False. When the first operand is True, the result can be either True or False depending on the second operand, so when the first operand evaluates to True, the interpreter has to evaluate the second operand.

A similar explanation goes for the `or` operator. Here is the truth table of `or` operator.

| operand1 or operand2 |    |       |    |       |
|----------------------|----|-------|----|-------|
| True                 | or | True  | => | True  |
| True                 | or | False | => | True  |
| False                | or | True  | => | True  |
| False                | or | False | => | False |

*Figure 6.7: Truth table of or operator*

We can see that if the first operand is True, the result is True regardless of the value of the second operand. If the first operand evaluates to True, the interpreter will not evaluate the second operand and will consider the whole expression as True. If the first operand is False, the result can be True or

False depending on the second operand, so when the first operand evaluates to False, the interpreter has to evaluate the second operand.

Therefore, in the case of `and` operator, if the first operand is False, the second operand is not evaluated, and in the case of `or` operator, if the first operand is True, the second operand is not evaluated.

This is called the short circuit evaluation of these operators. This feature not only makes the interpreter do less work but sometimes it can also be used to prevent certain types of errors. Here are some examples:

```
if x != 0 and 1/x > n:
 print('Do something')
if i < len(data) and data[i] == item:
 print('Do something')
if x >= 0 and x**0.5 > 4:
 print('Do something')
if 'city' in d and d['city'] == 'Paris':
 print('Do something')
```

In these cases, we want the second condition to be checked only if the first condition is True. If the first condition is False, we do not want the second condition to be checked because, in that case, it will give an error. For example, in the first code snippet, we want the comparison `1/x > n` to be done only when `x` is not equal to 0 because otherwise, it will give a divide by zero error.

`x != 0 and 1/x > n` are the two operands of `and` operator; the interpreter will evaluate the second operand only if the first one is True. This means that it will evaluate `1/x > n` only if `x` is non-zero. If `x != 0` is False, `1/x > n` will not be evaluated, so there are no chances of getting any divide by zero error. If the interpreter were to evaluate both operands, we would get a divide by zero error when `x` is zero. Similarly, in the second example, we have avoided taking the square root of negative numbers. The operand `x**0.5 > 4` will be evaluated only when `x` is a positive number.

This short circuit evaluation can also be useful in sequences and dictionaries to avoid `IndexError` or `KeyError`. In a sequence, before checking data at a certain index, we can ensure the index is valid. Similarly, in a dictionary, we can check for a valid key before accessing the value associated with that key. This way, we can avoid `IndexError` in sequences and `KeyError` in dictionaries, as we have done in the last two examples. Thus, the left operand can act as a guard for the second operand.

Equivalently, we could have written these constructs using two `if` statement. For example, we can write the first example like this:

```
if x != 0:
 if 1/x > n:
 print('Do something')
```

This one works in the same way, but the one with the `and` operator is more readable and is a common trick used by programmers.

## 6.7 Values returned by `and` and `or` operators

Unlike some other languages, in Python, the logical operators `and` and `or` do not return Boolean values `True` or `False`; they actually return the last evaluated operand. We generally use these operators in `if` and `while` conditions, so we do not get to know what they return exactly because, in those cases, only their truth value is used. Let us see what they actually return.

```
>>> 0 and 4
0
>>> 4 and 8
8
>>> 0 or 4
4
>>> 4 or 8
```

For `and` operator, the second operand is not evaluated if the first one is `False`.

In the expression `0 and 4`, the interpreter evaluates the first operand; it is `False`, so there is no need to evaluate the second operand. `0` is the last evaluated operand and so it is returned.

In the expression `4 and 8`, the first operand is `True`, so the second operand has to be evaluated, and so here, `8` is the last evaluated operand, and it is returned.

For `or` operator, the second operand is not evaluated if the first one is `True`.

In the expression `0 or 4`, the first operand is `False`, so the second operand has to be evaluated. `4` is the last evaluated operand, so it is returned.

In the expression `4 or 8`, the first operand is `True`; there is no need to evaluate the second one. Thus, `4` is the last evaluated operand, so it is returned.

The expression `operand1 and operand2` first evaluates `operand1`; if it is `False`, its value is returned; otherwise, `operand2` is evaluated and its value is returned.

The expression `operand1 or operand2` first evaluates `operand1`; if it is `True`, its value is returned; otherwise, `operand2` is evaluated, and its value is returned.

These operators actually return operands, but most of the time, they are used in a Boolean context, so only their truth value is used. The fact that they return the last evaluated argument can be used by programmers in certain situations.

Suppose we have a string `s`, and if it is empty, it has to be replaced by a default value, `'NA'`. We can write this:

```
s = s or 'NA'
```

If the string `s` is empty, the first operand `s` will be `False`, so the second operand will be evaluated, and it becomes the value of the expression. If the string `s` is not empty, the second operand will not be evaluated, and the

value of the expression `s or 'NA'` will be `s` only. Here is another example:

```
average = count!=0 and total/count
```

Here, we are finding the average and guarding our division by using the `and` operator. If `count` is not equal to zero, the first operand will be `True`, so the second operand will be evaluated, and its value will be returned and assigned to `average`.

If `count` is equal to 0, the first operand will be `False`, so the second operand will not be evaluated, thus avoiding divide by zero error. The value of the first operand will be assigned to `average`. So, `average` will be assigned `False`. In Python, `False` is numeric value 0, and `True` is numeric value 1. So, when we use this `average` in mathematical context, value 0 is used.

The operator `not` always returns Boolean value `True` or `False`; `True` if its argument is falsy, `False` if its argument is Truthy.

## 6.8 if else operator

We know that unary operators operate on one operand, and binary operators act on two operands. The operator that we are going to see now is a ternary operator, as it acts on three operands. Here is what the if-else operator looks like with its three operands.

```
expression1 if test-expression else expression2
```

The 3 expressions are the 3 operands. The keywords `if` and `else` form the operator. Let us see how this operator works.

The `test-expression` or the condition is evaluated, and if it is `True`, the left expression is evaluated and its value is the value of the whole expression, and if the condition is `False`, then the right expression is evaluated and its value is the value of the whole expression. So, this operator checks the condition and then returns the value of either of the two expressions, depending on the truth value of the condition. The condition will always be evaluated, while only one of the two expressions will be evaluated. Here is an example:

```
x = 6
```

```
y = x+5 if x%2==0 else x+10
```

First, the condition `x%2==0` is checked, the value of `x` is 6, `6%2` is 0, and the condition `x%2==0` is True, so the first expression is evaluated, and the value `6+5` is assigned to `y`.

If the value of `x` is 7, the condition `x%2==0` will be False, so the second expression will be evaluated, and the value `7+10` will be assigned to `y`.

We can read the statement as - `y` will be equal to `x+5` if `x` is even, else `y` will be equal to `x+10`. The statement could be written using an `if` statement.

```
if x % 2 == 0:
```

```
 y = x + 5
```

```
else:
```

```
 y = x + 10
```

We can see that the ternary operator is just a shorthand operator that reduces a 4-line `if else` code to a simple one-line code, which is quite readable. Let us see some more examples that will make things clearer.

```
remarks = 'Pass' if marks >= 40 else 'Fail'
```

If `marks >= 40`, the string `'Pass'` is assigned to `remarks`; otherwise, `'Fail'` is assigned.

```
discount = 5 if items < 10 else 15
```

If a customer buys less than 10 items, the discount is 5 percent; otherwise, the discount is 15 percent.

```
greater = x if x > y else y
```

Here, the variable `greater` is assigned the value of `x` if `x` is greater than `y`; otherwise, it is assigned the value of `y`.

```
average = total/count if count else None
```

If `count` is non zero, `total/count` value is assigned to `average`; otherwise, `None` is assigned to `average`.

```
print('Sir' if gender == 'male' else 'Madam')
```

Here, we have used the ternary operator inside a `print` function call.

```
voter_id = 'NA' if age < 18 else input('Enter voter id')
```

If `age` is less than 18, 'NA' is assigned to `voter_id`; otherwise, the value returned by `input` is assigned.

```
z = 10 + (x if x > y else y)
```

Here, we add 10 to the greater of the two values `x` and `y`. If we do not put the parentheses, Python will interpret it differently, taking `10 + x` as the first expression.

```
b = 100 * (a if a>=0 else -a)
```

Here, we are multiplying 100 with absolute value of `a`.

For these simple cases, a full 4-line `if-else` code would be an overkill, while the ternary operator is concise and more readable. There is no efficiency difference between an `if-else` statement and `if-else` operator code, but the code with this operator is shorter.

Let us see an example in which one conditional expression is placed inside another.

```
remarks = 'Excellent' if marks>=90 else ('Pass' if marks>=40 else 'Fail')
```

If `marks >=90`, `remarks` will be assigned 'Excellent'; otherwise, if `marks >= 40`, `remarks` will be assigned 'Pass'; otherwise, `remarks` will be assigned 'Fail'.

The equivalent code using the `if` statement would be:

```
if marks >= 90:
 remarks = 'Excellent'
elif marks >= 40:
 remarks = 'Pass'
else:
 remarks = 'Fail'
```



## Exercise

What will be the output of questions 1 to 7?

```
1. n = 2
 if n = 2:
 print('X')
 else:
 print('Y')
2. units = 95
 if units < 100:
 bill = units * 1
 else:
 bill = uniiits * 1.5
 print(bill)
3. s = None
 if s is 'None':
 print('this')
 else:
 print('that')
4. x = 9.7
 if x:
 print('Hello')
 else:
 print('Hi')
5. listA = [1, 2, 3, 4]
 if not listA:
```

```
 print('Good Morning')
else:
 print('Good Evening')
```

6. `m = 10`

```
n = 50 if m < 0 else 20
print(n)
```

7. `y = 402`

```
x = 2 if y % 2 == 0 else 1
print(x)
```

8. When will the following code print C as the output?

```
if expression1:
 print('A')
elif expression2:
 print('B')
elif expression3:
 print('C')
else:
 print('D')
```

(A) when expression1, expression2 and expression3 all are True

(B) when expression1 and expression2 are False and expression3 is True

9. `a = 10`

`b = 20`

Which of the following expressions will be True?

(i) `a > 0 and b % 2 == 0` (ii) `a % 2 == 0 and b < 0`

(A) Only (i) (C) Both (i) and (ii)

(B) Only (ii)

10. `x = 3`

Which of the following expressions is False?

(A) `x < 0` (C) `not x % 2 != 0`

(B) `not x` (D) All are False

11. `if n > 0:`

`if n < 10:`

Which of these is equivalent to the above nested if conditions?

(A) `if 0 < n < 10:`

(B) `if 0 < n and n < 10:`

(C) Both

12. Write a program that enters a string and prints whether it is a palindrome. Ignore case and spaces, so that all strings like ‘Nurses run’ ‘Was it a rat I saw’ are considered palindromes.

13. Write a program that inputs the length of three sides of a triangle, and prints the perimeter of the triangle. It should also print whether the triangle is equilateral, isosceles or scalene. If there is no triangle possible with the given sides, then instead of printing the above things it should print ‘No triangle possible with these sides’.

In an equilateral triangle, all sides are equal. In an isosceles triangle, any two sides are equal. In a scalene triangle, all three sides are unequal. For a triangle to be possible with given sides, sum of any two sides should be greater than the third side.

Here are two sample runs of the program.

**Sample run 1- Sample Run 2-**

Enter first side : 2      Enter first side : 2

Enter second side : 3      Enter second side : 1

Enter third side : 4      Enter third side : 4

Perimeter of the triangle is 9 No triangle possible with these sides

Scalene Triangle

14. Write a program that prompts the user to input his/her weight in kg and height in cm, and calculates the body mass index (BMI). BMI is calculated by dividing body weight in kg by square of height in meters. For example if weight is 70 kg, height is 170 cm, then BMI is  $70/(1.7*1.7) = 24.2$  Display the BMI and appropriate message according to the BMI.

< 18.5 - Underweight 18.5 to 24.9 - Normal weight 25 to 29.9 - Overweight  $\geq 30$  - Obese

15. In the previous program, give the user an option to enter the height in inches or cm and weight in kgs or pound.

1 inch = 2.54 cm, 1 pound = 0.4535924 kg

16. Write a program to check whether a given sentence is a pangram or not. A pangram is a sentence that uses every letter of the alphabet at least once. Some examples of pangrams are - "The quick brown fox jumps over the lazy dog." "Pack my box with five dozen liquor jugs." "Waltz, nymph, for quick jigs vex Bud."

17. Write a program to find whether two phrases are anagrams. Anagram is a word or a phrase, formed by rearranging the letters of a different word or phrase. Some examples of anagrams are -

"binary" and "brainy", "silent" and "listen", "forty five" and "over fifty", "Madam Curie" and "Radium came"

18. Write a program to find whether a year is a leap year or not. A year is a leap year if it is divisible by 4, but not every year that is a multiple of 4 is a leap year. If a year is divisible by 100, then it is not a leap year unless it is divisible by 400.

Years 1980, 2040 are leap years as they are divisible by 4.

Years 2000, 2400, 1800, 1900, 2500 are divisible by 4, but since they are divisible by 100 we cannot say that all of them are leap years. Only those which are divisible by 400 will be leap years.

Years 2000 and 2400 are leap years as they are divisible by 400, while 1800, 1900, 2500 are not leap years.

19. We have seen the following program in the chapter.

```
marks = int(input('Enter marks - '))
if marks >= 70:
 grade = 'A'
elif marks >= 60:
 grade = 'B'
elif marks >= 50:
 grade = 'C'
elif marks >= 40:
 grade = 'D'
else:
 grade = 'E'
print(grade)
```

Suppose most of the students get grade E or grade D and very less students get A grade. It would be more efficient to rewrite this code the other way round. Refactor this code so that the more frequent conditions are written at the top of the if statement.

20. Rewrite the following piece of code using the ternary operator.

```
if bill_amount > 2000:
 free_home_delivery = 'Available'
else:
 free_home_delivery = 'Not Available'
```

21. Write a more efficient version of this code

```
if x < y:
 print('x is less than y')
```

```

if x > y:
 print('x is greater than y ')
if x == y:
 print('x is equal to y')

```

22. A list named `L` contains some integer values. Write a line of code to find the average of the list elements using the ternary operator.

23. We have seen that we can use the `get` method to avoid errors while accessing a non-existent key.

```

D = {'a': 23, 'd': 34, 'j': 56}
val = D['b'] # Raises Error
val = D.get('b', 0) # Returns 0

```

Instead of using `get()`, write a line of code that uses a ternary operator to return a default value when the key is not present in the dictionary.

24. Rewrite these expressions by eliminating the `not` operator so that the new expressions are more readable.

(i) `if not grade == 'A':` (iv) `if not (marks > 0 and marks <= 100):`

```

 print('Work Hard')
print('Out of range')

```

(ii) `if not age < 18:` (v) `if not (age < 18 or weight > 60):`

```

 print('You can vote')
print('Allowed to play the game')

```

(iii) `if not n % 2 == 0:`

```

 print('n is odd')

```

You can avoid using the `not` operator by using the opposite relational operator.

For (iv) and (v) you can use the DeMorgan's laws to distribute the not operator over boolean expressions.

1. NOT (a AND b) = (NOT a) OR (NOT b) 2. NOT (a OR b) = (NOT a) AND (NOT b)

25. What will be the output of the following code? Will it show TypeError?

```
print(True + 4)
print(False - 3)
```

26. Write a program using if...elif...else for printing the name of the day depending on the value of a variable.

| value           | Action          |
|-----------------|-----------------|
| 0               | Print Sunday    |
| 1               | Print Monday    |
| 2               | Print Tuesday   |
| 3               | Print Wednesday |
| 4               | Print Thursday  |
| 5               | Print Friday    |
| 6               | Print Saturday  |
| Any other value | Print Invalid   |

Write the same code using a dictionary.

Statements written in a program are executed sequentially, and each statement is executed only once. However, many tasks are repetitive in nature, so there can be situations when we need to execute a statement or a block of statements multiple times. Python provides two control statements called loops, that can be used to repeatedly execute a piece of code.

A loop or an iterative control statement is a control statement that is used for repeated execution of a block of statements. We need loops when we want to do something more than once. Instead of writing the same statements repeatedly in our code, we can use loops to automate the repetition. In Python, we have two loops: `while` loop and `for` loop.

## 7.1 while loop

In `while` loop, there is repeated execution of a block of statements while a given condition is `True`. Here is the syntax and flowchart of a `while` loop.

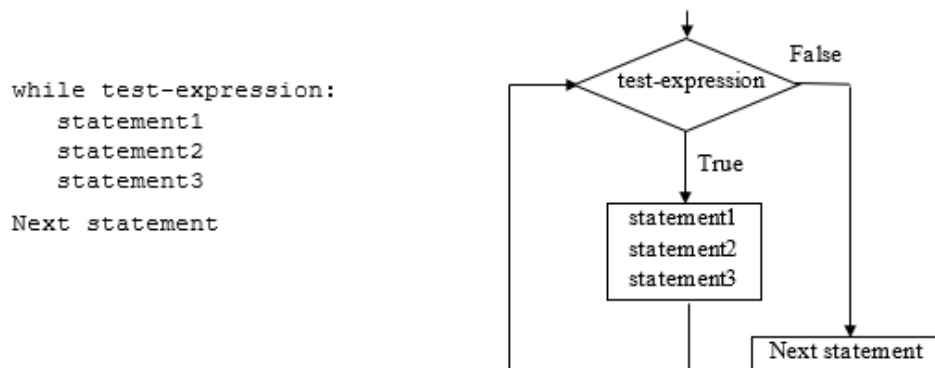


*Figure 7.1: Syntax and flowchart of while loop*



The `while` keyword is followed by a test expression, also called the loop condition, which is followed by a colon. This colon marks the start of the statement block, which is to be executed repeatedly while the test expression is true. The statement block is also called the loop body, and the indentation separates this loop body from the header line.

Let us understand how this loop works. First, the test expression is evaluated; if it is True, the statement block executes, and then the control returns to the test expression. If it is True, again the block executes and then the test expression is checked. This process continues till the test expression is True. When it becomes False, the loop terminates, the control comes out, and the next statement out of the loop is executed. So, this loop keeps running while the test expression at the top is True. One complete execution of a loop is called an iteration. Here is a simple example to show how the while loop works:

```
n = 1
while n <= 3:
 print('Hello ' * n)
 n += 1
print('Bye')
```

### ***Output-***

```
Hello
Hello Hello
Hello Hello Hello
Bye
```

We have an integer variable `n` initialized to 1. Then, we have the `while` loop, and after the `while` loop, we have a `print` call that prints Bye. The test expression or the loop condition is `n <= 3`.

When the control enters the loop, the loop condition is checked. It is True because the value of `n` is 1, so the loop body executes. `Hello` is printed one time, and then the value of `n` is incremented. So, `n` now becomes 2. The condition is checked again. It is True, so the loop body executes again; `Hello` is printed two times, and the value of `n` becomes 3. The condition is checked again. It is True, so the loop body executes again; `Hello` is printed three times, and the value of `n` becomes 4.

Once again, the condition is checked. Now, it is False since the value of `n` is 4. So, the loop terminates, and the control comes to the next statement out of the `while` loop. This statement prints `Bye`. In simple English, this loop means that “while `n` is less than or equal to 3, keep executing the block of statements”.

If the condition is False the first time through the loop, the statements inside the loop are never executed. For example, in our loop, if the initial value of `n` is 4 instead of 1, the condition will be False the first time, and so the loop body will not execute even once.

```
n = 4
while n <= 3:
 print('Hello ' * n)
 n += 1
print('Bye')
```

### **Output-**

Bye

There should be a statement inside the loop body that makes the loop condition False at some point; otherwise, the loop condition will always be True, and the loop will keep on executing infinitely.

In our example loop, the statement `n += 1` is the statement that will make the loop condition False eventually. If we delete this statement, then the value of `n` will remain 1 always, and the condition will never become False, and the loop will never end.

```
n = 1
while n <= 3:
 print('Hello ' * n)
print('Bye')
```

If you execute this loop, you will see `Hello` being printed continuously. This is an infinite loop; we can press `Ctrl-C` to break it, or we have to close the window. To avoid an infinite loop, remember to place an update statement inside the loop body and make sure that the condition becomes False eventually at some point.

You can type this example loop and play around with it to understand how it works. For example, you can change the condition to `n <= 10` or `n >= 3` and

see how the output is affected. If you change the update statement `n+=1` to `n+=2`, `n` will be incremented by 2 each time.

Here is another example of a while loop:

```
total = 0
while total <= 100:
 num = int(input('Enter a number : '))
 total += num
print(total)
```

This loop adds the numbers input by the user and stops when the total exceeds 100.

## 7.1.1 Indentation matters

In Python, indentation determines the body of the loop. Unlike other languages, there are no curly braces or keywords to mark the beginning and end of the loop body. If you make any mistakes in indenting, you might get an unexpected output. Consider the following while loop that calculates the sum of digits of an integer:

```
n = int(input('Enter a number : '))
sum_digits = 0
while n > 0:
 digit = n % 10
 sum_digits += digit
 n //= 10
print(sum_digits)
```

### ***Sample Run-***

```
Enter a number : 3214
10
```

First, let us understand how this loop works, and then we will see how indentation can affect the output. Inside the loop, we extract the digits of the number from right to left, and the extracted digits are added to the variable `sum_digits`. The statement `n % 10` extracts the rightmost digit from the number, the next statement adds the extracted digit to the variable `sum_digits`, and the statement `n //= 10` divides `n` by 10 so that the next digit comes at the

rightmost place and can be extracted in the next iteration. Here is the dry run of the loop for a value of  $n$  equal to 3214.

```
sum_digits = 0 n = 3214
n > 0 is True digit = 3214 % 10 = 4 sum_digits = 0 + 4 = 4 n
= 3214 // 10 = 321
n > 0 is True digit = 321 % 10 = 1 sum_digits = 4 + 1 = 5 n =
321 // 10 = 32
n > 0 is True digit = 32 % 10 = 2 sum_digits = 5 + 2 = 7 n = 32
// 10 = 3
n > 0 is True digit = 3 % 10 = 3 sum_digits = 7 + 3 = 10 n = 3 //
10 = 0
n > 0 is False Loop terminates
```

This is how the loop works and gives the desired output.

If you have a loop that has multiple lines in the loop body, it is possible that mistakenly the last one or two lines do not get indented. The interpreter will not complain in this case as it is satisfied with just a single line in the loop body. So, there will be no syntax error, but the lines that are not indented will not be part of the loop and hence will not be repeated. For example, in the previous loop, suppose the last line is not indented.

```
while n > 0:
 digit = n % 10
 sum_digits += digit
n //= 10
print(sum_digits)
```

This will result in an infinite loop because the update statement ( $n //= 10$ ) is now outside the loop, with only 2 lines inside the loop body. So, make sure that you indent all the lines that you intend to be inside the loop body. Whatever is not indented will be considered out of the loop and will not be repeated.

Now, suppose the statement `print(sum_digits)`, which was supposed to be outside the loop, is indented by mistake.

```
while n > 0:
 digit = n % 10
 sum_digits += digit
 print(sum_digits)
```

```
n //= 10
print(sum_digits)
```

Now the statement `print(sum_digits)` is a part of the loop body, so in each iteration of the loop, this statement will also be executed. This will result in some extra undesired output on our screen.

Improper indentation can lead to such logical errors in our code, which the interpreter cannot detect but they make the program give unexpected results.

## 7.1.2 Removing all occurrences of a value from the list using the while loop

We have seen in an earlier chapter that the `remove` method of `list` type removes only the first occurrence of the given item.

```
L = [1, 5, 2, 3, 9, 4, 3, 2, 4, 2, 1, 2]
n = 2
L.remove(n)
print(L)
```

### **Output-**

```
[1, 5, 3, 9, 4, 3, 2, 4, 2, 1, 2]
```

We can see many 2s in our original list, but only the first occurrence of 2 was removed from the list. Now, we will use a `while` loop to remove all the occurrences.

```
L = [1, 5, 2, 3, 9, 4, 3, 2, 4, 2, 1, 2]
n = 2
while n in L:
 L.remove(n)
print(L)
```

### **Output-**

```
[1, 5, 3, 9, 4, 3, 4, 1]
```

We have written the statement `L.remove(n)` inside a `while` loop. This statement will continue executing until the loop condition `n in L` is `True`. So, the `remove` method will be repeatedly called till there is value `n` in the list `L`.

When `n in L` returns `False`, the loop will end. The value of `n` is 2, so this code will remove all the 2s from the list `L`.

## 7.1.3 while loop for input error checking

We can use the `while` loop to validate input, which means that we can ensure that the user enters valid input. Here is a small piece of code where the user is expected to enter a student id in the range of 1000-9999.

```
student_id = input('Enter student id (1000-9999) : ')
print(student_id)
```

If the user enters something that is not an integer or is not in the valid range (1000-9999), the program will not complain. We have instructed the user to enter the correct id, but we are not checking the input.

We can use an `if` statement here.

```
student_id = int(input('Enter student id (1000-9999) : '))
if student_id >= 1000 and student_id <= 9999:
 print(student_id)
```

If the entered input is in the correct range, the condition will be `True`, and the id will be printed. This `if` statement checks the input, but we want to give the user another chance to enter the input in the correct form. We want to keep asking him to enter the id till he enters the id in the correct form. For that, we can use a `while` loop.

```
student_id = int(input('Enter student id (1000-9999) : '))
while student_id < 1000 or student_id > 9999:
 student_id = int(input('Enter student id (1000-9999) : '))
print(student_id)
```

The first `input` statement executes and then the control goes to the `while` loop. If the entered id is not in the valid range, then the loop condition is `True`, and the loop body executes and keeps executing until the user enters a valid id. When the correct id is entered, the loop will terminate and the program will continue. If the

id is entered in the correct form the first time itself, the loop condition will be False, so the loop body will not execute even once. So, we can use the `while` loop to ensure the user enters the correct input. In the next chapter, we will discuss a better way of writing this loop.

## 7.1.4 Storing user input in a list or dictionary

We can use the `while` loop to get data from the user and store it in a list or dictionary. In the following example, we have a dictionary with a few items. By using a `while` loop, we are letting the user enter some more items in this dictionary.

```
fruit_prices = {'apple': 210, 'banana': 100, 'grapes': 90}

done = False

while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True

print(fruit_prices)
```

We have taken a Boolean variable named `done` and initialized it to False. We have made it True inside the loop when the user is done entering all the items. This loop executes as long as the variable `done` is False because we have used the `not` operator in the loop condition. When `done` is False, the loop condition is True, and when `done` is True, the loop condition is False. So, when `done` will become True, the loop will terminate.

Inside the loop, we are asking the user to enter the fruit name and then the price, and in the next statement, we are entering the pair into the dictionary. After that, we are asking the user if he wants to enter more pairs. If users types 'n', which stands for no, variable `done` is set to True, and the loop terminates. Otherwise, the loop keeps executing, and the user can enter several pairs of fruits and prices, which will be added to the dictionary.

## 7.2 for loop

The `while` loop of Python is similar to the `while` loop of most other programming languages. However, the syntax of `for` loop differs from the standard three-expression `for` loop in languages like C++ or Java. The `for` loop in Python is more like a `for each` loop available in some other languages.

Like the `while` loop, the `for` loop is also used to repeatedly execute a block of code, but unlike a `while` loop, it is not based on a condition. It is a collection-controlled loop, and it iterates once for each element in the collection. Here is the syntax of a `for` loop:

```
for item in iterable:
 statement1
 statement2
 statement3
```

We have the keyword `for`, then a variable name, another keyword `in`, and then an iterable name. This iterable can be any iterable structure like a string, list, tuple, set, dictionary, or even a file. The elements in this iterable are assigned to the variable named `item` one by one, and the statement block is executed once for each item. Here is an example of `for` loop in action.

```
data = [3, 5, 9, 8]
for number in data:
 print(number)
```

### **Output-**

```
3
5
9
8
```

This loop prints each element of the list on a separate line. Let us discuss how this loop is working. When the loop starts, the first element in the list is assigned to the iterating variable named `number`, and the statement block is executed. On the next iteration, the second element of the list is assigned to the variable `number`, and the statement block is executed. This process continues until the



entire list is exhausted. So, the loop terminates when this loop body has been executed for each element of the list. In simple English, this loop means “for every number in data, execute this statement.”

You can think of the loop working in this way:

```
First iteration : number = 3 print(number)
Second iteration : number = 5 print(number)
Third iteration : number = 9 print(number)
Fourth iteration : number = 8 print(number)
```

We could do the same work using a `while` loop.

```
data = [3, 5, 9, 8]
i = 0
while i < len(data):
 print(data[i])
 i += 1
```

Let us discuss how this loop works. `i` is initially zero, and the loop condition is `i < len(data)`. When `i` will become equal to the length of the list, the loop condition will become False, and the loop will terminate. For the given list, the length is 4, so this loop will execute for `i=0`, `i=1`, `i=2`, `i=3`, and when `i` will become 4, the loop condition will become False, and the loop will end. This is how we access the elements at indices 0, 1, 2, and 3.

The `for` loop syntax is much simpler and cleaner as there is no need to manage an index, calculate the length of the list, write a Boolean condition, and update expression; all this is done for you automatically by the `for` loop. The whole process is automated; we do not have to tell the loop when to terminate; it automatically terminates when it has iterated for all the elements of the iterable. It is a smart loop that knows everything about the iterable that is provided to it.

Although you can write your code using a `while` loop, use the `for` loop wherever you can, as it is cleaner and is considered more Pythonic. Moreover, `for` loop is important in iterables that do not support direct indexing, like sets and dictionaries.

Let us discuss some more examples of `for` loops. The `for` loop example we have discussed prints the numbers on a separate line. If we want to print them on a single line, we can make this small change.

```
data = [3, 5, 9, 8]
for number in data:
 print(number, end=' ')
```

### **Output-**

```
3 5 9 8
```

The numbers are printed with spaces in between them. Now, instead of printing these numbers, let us print the squares of these numbers.

```
data = [3, 5, 9, 8]
for number in data:
 print(number * number, end=' ')
```

### **Output-**

```
9 25 81 64
```

Suppose we want to display the squares of only even numbers. We can put an `if` condition for that.

```
data = [3, 5, 9, 8]
for number in data:
 if number % 2 == 0:
 print(number * number, end=' ')
```

### **Output-**

```
64
```

Next, we will write a `for` loop to count how many even numbers are in a list.

```
numbers = [2, 4, 5, 34, 7, 21, 67]
even_count = 0
for number in numbers:
 if number % 2 == 0:
 even_count += 1
print(even_count)
```

### **Output-**

We took a variable `even_count` and initialized it to zero. If the number is even, we increment the variable `even_count`. At last, we print the variable `even_count`. This is how we get the count of even numbers in the list.

The loop variable can be given any name that is a valid Python identifier, but it is good to give names that denote a single item from the iterable. For example, suppose you are iterating on a list named `students`. It is good to take the loop variable name as `student`, as each item in the list will represent a student. So, it is good to use plural names for the lists, etc, and singular ones for the loop variable. This naming convention will make your code more intuitive and readable and look more English-like.

## 7.2.1 Iterating over a string with for loop

In the following program, we have used a string at the place of iterable in a `for` loop.

```
message = 'Hello World'
for ch in message:
 print(ch, end=' ')
```

### **Output-**

```
H e l l o W o r l d
```

A string is a sequence of characters, so this `for` loop actually iterates over all characters in the string. Each time through this loop, a character from the string is assigned to the variable `ch`, and the loop terminates when there are no more characters left. When we run this loop, all the characters of the string are displayed with space in between. If we want to display only the vowels, we can put an `if` condition.

```
message = 'Hello World'
for ch in message:
 if ch in {'a', 'e', 'i', 'o', 'u'}:
 print(ch, end=' ')
```

### **Output-**

```
e o o
```

Only the vowels from the string are displayed. Now let us try to encrypt the message. We will encrypt our message by simply replacing each character with the subsequent character in the Unicode. For example, Hello World ! after encryption will become Ifmmp!Xpsme!"

We will use the built-in functions `ord` and `chr`, so first, let us discuss these functions. The function `ord` returns the Unicode code point for a one-character string, and the function `chr` returns a Unicode string of one character from the number provided.

```
>>> ord('a')
```

```
97
```

```
>>> ord('b')
```

```
98
```

```
>>> chr(97)
```

```
'a'
```

```
>>> chr(98)
```

```
'b'
```

The function `ord` converts a character to a number, and the corresponding function `chr` converts a number to a character. 'ord' in the `ord` function stands for ordinal. `ord('a')` is 97, 97+1 is 98, and if we put 98 inside `chr` function, we get 'b'. Therefore, if we write `chr(ord('a')+1)` we get 'b'. Similarly, if we write `chr(ord('f')+1)` we get 'g'. So, by writing the expression `chr(ord(ch)+1)` we can get the character that comes after the character `ch`.

Here is the loop that encrypts a message. The string `emessage` will be used to store the encrypted message; initially, it will be empty.

```
message = 'Hello World !'
```

```
emessage = ''
```

```
for ch in message:
```

```
 emessage += chr(ord(ch) + 1)
```

```
print(emessage)
```

### **Output-**

```
Ifmmp!Xpsme!"
```

We are iterating over the string `message` in the `for` loop, and inside the loop, we are building the string `emessage` by adding the subsequent character of each character in the string `message` and at the end, we print `emessage`. The following loop will decrypt the encrypted message:

```
emessage = 'Ifmmp!Xpsme!''
dmessage = ''
for ch in emessage:
 dmessage += chr(ord(ch) - 1)
print(dmessage)
```

### **Output-**

```
Hello World !
```

The string `dmessage` denotes the decrypted message. We are iterating over the encrypted message, and inside the loop, we are building the decrypted message. In the expression `chr(ord(ch)-1)`, we have written `-1` to get the previous character. When we execute this loop, we will get the original message back.

So, we have seen how to use `for` loops with lists and strings. In the coming sections, we will see how to use the `for` loop with dictionaries, sets, and `range` function.

## **7.2.2 Unpacking in for loop header**

We know that we can have a sequence of sequences like list of lists, or tuple of tuples or list of tuples. Suppose we have this list of 2-item tuples:

```
L = [('John', 20), ('Sam', 15), ('Dev', 21), ('Ryan', 10)]
```

To iterate over this list, we can write the following loop:

```
for t in L:
 print(t)
```

In each iteration, the name `t` is assigned a tuple from the list `L`, which is printed inside the loop.

First iteration : `t = ('John', 20)`

Second iteration : `t = ('Sam', 15)`

Third iteration : `t = ('Dev', 21)`

Fourth iteration : `t = ('Ryan', 10)`

Inside the loop, we are printing the tuple as a whole. If we want to access each item of this tuple separately, we can unpack the tuple inside the code block.

```
L = [('John', 20), ('Sam', 15), ('Dev', 21), ('Ryan', 10)]
```

```
for t in L:
 name, age = t
 if age > 18:
 print('Mr', name)
 else:
 print('Master', name)
```

### ***Output-***

Mr John

Master Sam

Mr Dev

Master Ryan

We have unpacked the tuple `t` and stored the values in the variables `name` and `age`, and then we are using these variables inside the loop. So here, in each iteration, the tuple assignment is done first, and then tuple unpacking is done inside the loop.

|                   |                               |                            |
|-------------------|-------------------------------|----------------------------|
| First iteration : | <code>t = ('John', 20)</code> | <code>name, age = t</code> |
|-------------------|-------------------------------|----------------------------|

|                    |                              |                            |
|--------------------|------------------------------|----------------------------|
| Second iteration : | <code>t = ('Sam', 15)</code> | <code>name, age = t</code> |
|--------------------|------------------------------|----------------------------|

|                   |                              |                            |
|-------------------|------------------------------|----------------------------|
| Third iteration : | <code>t = ('Dev', 21)</code> | <code>name, age = t</code> |
|-------------------|------------------------------|----------------------------|

|                    |                               |                            |
|--------------------|-------------------------------|----------------------------|
| Fourth iteration : | <code>t = ('Ryan', 10)</code> | <code>name, age = t</code> |
|--------------------|-------------------------------|----------------------------|

This unpacking can be done in the `for`-loop header itself.

```
for name, age in L:
 if age > 18:
 print('Mr', name)
```

```
else:
 print('Master', name)
```

First iteration :    name, age = ('John', 20)

Second iteration :    name, age = ('Sam', 15)

Third iteration :    name, age = ('Dev', 21)

Fourth iteration :    name, age = ('Ryan', 10)

In each iteration, a tuple is unpacked, and its values are assigned to `name` and `age`. So now we are unpacking the tuples that we get from the list `L` directly in the header only. There is no need for the extra step inside the loop body.

We have seen that we can unpack for any iterable type, so we can do the same thing if we have a list of lists. Let us change `L` to a list of lists.

```
L = [['John', 20], ['Sam', 15], ['Dev', 21], ['Ryan', 10]]
```

```
for name, age in L:
 if age > 18:
 print('Mr', name)
 else:
 print('Master', name)
```

The unpacking still works. Now the inner lists are being unpacked in each iteration. So, this is how we can unpack in the for loop header. When we have multiple identifiers after the `for` keyword, it means unpacking is being done.

Now, suppose we have a list of 3 item tuples.

```
L = [(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
```

To unpack these inner tuples, we will need three identifiers in the loop header.

```
for i, isquare, icube in L:
 print(i, isquare, icube)
```

### **Output-**

1 1 1

2 4 8

```
3 9 27
4 16 64
```

If we want to use only the first and the last value, we can ignore the second one using an underscore.

```
for i, _, icube in L:
 print(i, icube)
```

### **Output-**

```
1 1
2 8
3 27
4 64
```

We discussed the role of underscore in [Chapter 4](#).

We have seen how to iterate over a sequence of sequences and access the items inside the inner sequence through unpacking. We will see this unpacking when we use `enumerate` function, `zip` function, and dictionary `items` method in the `for` loop.

## **7.2.3 Iterating over dictionaries and sets**

To iterate over the keys of a dictionary, we can use one of the following loops:

```
for key in D:
 print(key)
for key in D.keys():
 print(key)
```

To iterate over the values of the dictionary, we can use the dictionary `values` method.

```
for value in D.values():
 print(value)
```

To iterate over both the keys and values of the dictionary, we can use the `items` method.



```
for item in D.items():
 print(item)
```

The `items` method returns the keys and values packed inside tuples. In each iteration of this for loop, the tuple returned by this method will be assigned to the variable `item`. We can unpack the tuple in the header to get the key and value.

```
for key, value in D.items():
 print(f'key is {key}, value is {value}')
```

Here is an example dictionary. Let us write different loops to iterate over this dictionary.

```
>>> D = {'apple': 210, 'banana': 100, 'grapes': 90,
'mango': 250, 'cherry': 225}
>>> for fruit in D:
... print(fruit, end=' ')
apple banana grapes mango cherry
>>> for fruit in D.keys():
... print(fruit, end=' ')
apple banana grapes mango cherry
>>> for price in D.values():
... print(price, end=' ')
210 100 90 250 225
>>> for pair in D.items():
... print(pair, end=' ')
(('apple', 210) ('banana', 100) ('grapes', 90) ('mango',
250) ('cherry', 225))
>>> for fruit, price in D.items():
... print(f'Price of {fruit} is {price}')
```

Price of apple is 210  
Price of banana is 100  
Price of grapes is 90

Price of mango is 250

Price of cherry is 225

The following loop will print only the costly fruits (the ones whose price is more than 200).

```
>>> for fruit, price in D.items():
... if price > 200:
... print(f'Price of {fruit} is {price}')
```

Price of apple is 210

Price of mango is 250

Price of cherry is 225

In the next for loop, we have decreased the price of costly fruits by 10%.

```
>>> for fruit, price in D.items():
... if price > 200:
... D[fruit] -= 0.1 * price
>>> D
{'apple': 189.0, 'banana': 100, 'grapes': 90, 'mango':
225.0, 'cherry': 202.5}
```

for loops can also be used to iterate over the items of a set. Here is an example:

```
>>> s = {4, 6, 2, 8, 9}
>>> for x in s:
... print(x, end=' ')
2 4 6 8 9
```

This loop prints all the elements of the set. Sets are unordered, so the order of iteration is undefined.

## 7.2.4 Iterating through a series of integers

We can use a `while` loop to iterate through numerical values. For example, the following `while` loop iterates through integers from 1 to 10.

```
i = 1
```

```
while i < 11:
 print(i)
 i += 1
```

Although we can do this using a `while` loop, the more Pythonic way of iterating through a series of numbers is to use a `for` loop.

The `for` loop of Python is different from the standard `for` loops in languages like C, which allows us to iterate through a numerical range easily. The `for` loop in these languages are specifically counter-based loops, while in Python, `for` loop is a collection-based loop. It is used to iterate through elements of an iterable. So, if you want to iterate over a range of numbers, you could use a list of numbers in the loop.

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
 print(i)
```

This loop iterates through numbers from 1 to 10, but hardcoding the list in this way is not a good idea; what if we have to iterate over a large range of values, like from 1 to 100? The common Pythonic idiom is to use the `range` function that we have seen before in [Chapter 4](#). This function returns an iterable object which can be converted to a list, as we have seen, or it can be used in a `for` loop.

```
for i in range(11):
 print(i, end=' ')
```

### **Output-**

```
0 1 2 3 4 5 6 7 8 9 10
```

The `range` function call used here generates integers from 0 to 10. The integer is assigned to the loop variable in each iteration, and the loop body executes once for each integer generated by the `range` function.

The `range` function does not store all the values in the memory like a list. It gives an object that provides values on the fly as they are needed. So even a call like `range(9999999)` does not consume a lot of memory because it does not store all the numbers from 0 to 9999998. Instead, it provides the next number when asked.

As we have seen before, we can send different arguments to the `range` function to get different series of integers.

|                       |                                    |
|-----------------------|------------------------------------|
| <code>range(n)</code> | generates integers from 0 to $n-1$ |
|-----------------------|------------------------------------|

|                             |                                                                                            |
|-----------------------------|--------------------------------------------------------------------------------------------|
| <code>range(m, n)</code>    | generates integers from <code>m</code> to <code>n-1</code>                                 |
| <code>range(m, n, k)</code> | generates integers from <code>m</code> to <code>n-1</code> , with a step of <code>k</code> |

**Table 7.1:** *range function*

We can also create a decrementing loop by providing a negative value for the step.

```
for i in range(6, 0, -1):
 print(i, end=' ')
```

**Output-**

6 5 4 3 2 1

We often need to repeat a task a specified number of times. For example, we might need to print a line of dashes 4 times. This is how we can do it using a `for` loop.

```
for i in range(4):
 print(30 * '-')
```

**Output-**

```



```

If you want to print the line of dashes a different number of times, you can simply modify the argument passed to the `range()` function accordingly.

This `for` loop iterates exactly four times, but note that we did not use the loop variable anywhere inside the loop body. For these cases where we need to iterate a specified number of times but do not need to use the loop variable inside the loop body, the idiom is to write an underscore instead of the loop variable.

```
for _ in range(4):
 print(30 * '-')
```

Here we want the loop to run four times, but we are not concerned about the specific values returned by `range`, so we have used underscore to explicitly state that we are ignoring the value.

We can use the `range` function in `for` loop when we have to iterate over a series of integers or when we want to perform a task a given number of times.

## 7.3 Nesting of Loops

The body of a loop can contain any valid Python statement; `for` and `while` statements are also valid statements, so we can have a loop inside another loop, which means that loops can be nested. The following dummy code shows some examples of nested loops:

```
while test-expression: while test-expression:

 for item in iterable: while test-
expression:

for item in iterable: for item in iterable:

 for item in iterable: while test-
expression:


```

In the first example, we have a `for` loop inside a `while` loop; in the second one, we have a `while` loop inside a `while` loop; in the third one, we have a `for` loop inside a `for` loop; and in the fourth one, we have a `while` loop inside a `for` loop. Here are two more examples:

```
for item in iterable: for item in iterable:

 while test-expression: while test-
expression:
```

```

..... while test-expression2:
.....
for item in iterable:
.....
.....

```

In the first example, we have a `while` loop and a `for` loop inside a `for` loop; and in the second example, we have three levels of nesting. Inside the `for` loop, we have a `while` loop, and inside that `while` loop, we have another `while` loop. Let us see some programs of nested loops.

```

i = 1
while i <= 3:
 print('Outer while loop iteration', i)
 for j in range(1, 5):
 print('\tInner for loop iteration', j)
 i += 1

```

We have a `while` loop, and inside the `while` loop, we have three statements. The first statement is a `print` call. Then there is a `for` statement, and then the statement `i += 1`. So here, `while` loop is the outer loop, and `for` loop is the inner loop. In each iteration of the `while` loop, the `for` loop will be fully executed. This is the output that we get:

```

Outer while loop iteration 1
 Inner for loop iteration 1
 Inner for loop iteration 2
 Inner for loop iteration 3
 Inner for loop iteration 4
Outer while loop iteration 2
 Inner for loop iteration 1
 Inner for loop iteration 2
 Inner for loop iteration 3
 Inner for loop iteration 4

```

Outer while loop iteration 3

Inner for loop iteration 1

Inner for loop iteration 2

Inner for loop iteration 3

Inner for loop iteration 4

Let us understand how these loops execute. Initially, `i` is 1, so the loop condition `i <= 3` is True, the `print` call executes, then the `for` loop executes, and it iterates 4 times. So, the `print` call inside the `for` loop is executed four times. After this, `i` is incremented. It is now 2, which is less than 3, so the loop condition is still True. The `print` call executes, then `for` loop is executed so the `print` call inside it is executed 4 times.

`i` is incremented again and is now 3. The loop condition is still True, so the `print` call executes. Then, the `for` loop executes and `i` is incremented. Now `i` is 4, the loop condition is False, so the `while` loop terminates. Let us see some more examples.

The following `for` loop prints the times table of number `n`.

```
n = 3
for i in range(1, 11):
 print(f'{n} X {i:2} = {n * i:3}')
print()
```

When this code will be executed, the times table of 3 will be printed. The numbers 2 and 3 placed after the colons in the parentheses represent the field width in which the given value is displayed.

Now, suppose we want to print tables of 5, 7 and 9 also. We can just copy, paste, and repeat this task.

```
n = 5
for i in range(1, 11):
 print(f'{n} X {i:2} = {n * i:3}')
print()
n = 7
for i in range(1, 11):
```

```

 print(f'{n} X {i:2} = {n * i:3}')
print()
n = 9
for i in range(1, 11):
 print(f'{n} X {i:2} = {n * i:3}')
print()

```

The tables of 5, 7, and 9 will also be printed. Suppose we want tables from 2 to 10. We will have to repeat this task 9 times. We know that when we have to repeat a task, we need to use a loop. So, we can enclose the whole code inside a `for` loop that provides different values of `n` in different iterations.

```

for n in [3, 5, 7, 9]:
 for i in range(1, 11):
 print(f'{n} X {i:2} = {n * i:3}')
 print()

```

In the first iteration of the outer loop, `n` will be 3, so the table of 3 will be printed. In the second iteration, `n` will be 5, so the table of 5 will be printed, and so on. The outer loop has 4 iterations, and for each iteration of the outer loop, the inner loop iterates 10 times.

If we want tables from 2 to 10, we can write a call to `range` function instead of the list.

```

for n in range(2, 11):
 for i in range(1, 11):
 print(f'{n} X {i:2} = {n * i:3}')
 print()

```

Now we will get tables from 2 to 10.

Let us discuss another example. We have a dictionary with fruit names as keys and prices as values. We want to display a data chart for this dictionary.

```

D = {'apple': 50, 'banana': 25, 'guava': 40, 'grapes':
34, 'orange': 30}
for fruit in D:

```



```

print(f'{fruit:8}', end=' ')
for i in range(D[fruit]):
 print('=', end='')
print()

```

### **Output-**

```

apple
=====
banana =====
guava =====
grapes =====
orange =====

```

We iterate over the keys of the dictionary and print the fruit name in a field width of 8. For each item, we have printed equal signs whose number is equal to the price of fruits. We have to repeat a task a specified number of times, so we have used the `range` function in the `for` loop. After printing a fruit, we need the next fruit on a newline, so there is a `print()` call at the end.

The inner `for` loop is used to repeatedly print the string '=' a number of times. In Python, we have another way of performing this repetitive task with strings. We can use the repetition operator instead.

```

for fruit in D:
 print(f'{fruit:8}', end=' ')
 print('=' * D[fruit], end='')
 print()

```

## **7.3.1 Using nested loops to generate combinations**

You can use nested loops to go through every possible combination of two, three, or more lists. For example, in the following code, we have written nested loops to print combinations of elements of two lists.

```

L1 = ['X', 'Y', 'Z']

```

```
L2 = [1, 2, 3, 4, 5]
for ch in L1:
 for num in L2:
 print(f'({ch},{num}) ', end=' ')
 print()
```

### **Output-**

```
(X,1) (X,2) (X,3) (X,4) (X,5)
(Y,1) (Y,2) (Y,3) (Y,4) (Y,5)
(Z,1) (Z,2) (Z,3) (Z,4) (Z,5)
```

The next program accepts three digits and creates a list that contains all the three-digit numbers that are combinations of those three digits.

```
a = int(input("Enter first digit : "))
b = int(input("Enter second digit : "))
c = int(input("Enter third digit : "))
digits = [a, b, c]
numbers = []
for i in digits:
 for j in digits:
 for k in digits:
 numbers.append(i * 100 + j * 10 + k)
print(numbers)
```

### **Sample Run-**

```
Enter first digit : 5
Enter second digit : 6
Enter third digit : 8
[555, 556, 558, 565, 566, 568, 585, 586, 588, 655, 656,
658, 665, 666, 668, 685, 686, 688, 855, 856, 858, 865,
866, 868, 885, 886, 888]
```

We stored the entered digits in a list named `digits` and then created another list named `numbers` to store all the combinations. The outer `for` loop will iterate three times; the inner `for` loop will iterate nine times; and the innermost `for` loop will iterate 27 times. In each iteration of the innermost loop, a number is appended to the list. So, the list `numbers` will have total of 27 numbers in it. The number that is appended will have `k` as the units digit, `j` as the tens digit, and `i` as the hundreds digit. This is how we get all three-digit numbers possible by combining these three digits.

If you want only unique digits in the combination, you can put an `if` statement before appending the number.

```
if i!=j and j!=k and k!=i:
```

## 7.3.2 Iterating over nested data structures

Nested loops can be used for iterating over nested data structures like nested lists or nested dictionaries. We had seen in [Chapter 4](#) that a matrix can be represented using a list of lists. Here is a nested list that represents a matrix with three rows and four columns.

```
matrix = [[1, 4, 8, 3],
 [2, 5, 6, 3],
 [1, 9, 5, 8]
]
```

To print the elements of this matrix in row and column form, we can use a nested `for` loop.

```
for i in range(3):
 for j in range(4):
 print(matrix[i][j], end=' ')
 print()
```

### **Output-**

```
1 4 8 3
2 5 6 3
1 9 5 8
```

In the outer loop, we are iterating over the outer list, which has three elements, so we have sent 3 as the argument to the `range` function. Each iteration of the outer for loop will print a row of the matrix. The inner for loop iterates over the inner lists, which have 4 elements each. We need each row on a separate line so we have written `print()` call after the inner for loop.

In our next example, we have a list of strings, and we have written a `for` loop to print only the alphabetical characters of each string in the list.

```
L = ['abc12*', 'xyz45!', '12pqr%', '(lmn)']
for string in L:
 for ch in string:
 if ch.isalpha():
 print(ch, end='')
 print()
```

### **Output-**

```
abc
xyz
pqr
lmn
```

In the outer `for` loop, we are iterating over the list `L`. In the inner `for` loop, we are iterating over each string of the list. Inside the inner loop, we are printing only those characters of the string which are alphabetical.

Let us see an example of nested dictionaries. In [Chapter 5](#), we saw this dictionary of dictionaries.

```
students = {105416: {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
 'age': 21,
 'marks': {'Maths': 89, 'Physics':
89, 'Chemistry': 91},
 'is_sporty': True},
```

```

144547: {'name': 'Dev',
 'gender': 'M',
 'city': 'London',
 'age': 23,
 'marks': {'Maths': 88, 'Physics':
77, 'Chemistry': 98},
 'is_sporty': False},
132399: {'name': 'Mary',
 'gender': 'F',
 'city': 'Paris',
 'age': 22,
 'marks': {'Maths': 99, 'Physics':
87, 'Chemistry': 88},
 'is_sporty': True}
}

```

Suppose we have to calculate the total marks of each student and add a new key named `total` for each inner dictionary. The value for that key will be the total marks. We can write the following nested loops to achieve this.

```

for student in students.values():
 total = 0
 for marks in student['marks'].values():
 total += marks
 student['total'] = total

```

In the outer `for` loop, we are iterating over the values of the `students` dictionary. Inside the loop, we take a variable `total` and initialize it to zero. Then, we write another `for` loop to iterate over the `marks` list and calculate the total marks. Then, we add a new key to each student dictionary. On printing the dictionary, we can see that a new key has been added for each student.

```

import pprint
pprint.pp(students)

```

## **Output-**

```
{105416: {'name': 'John',
 'gender': 'M',
 'city': 'Paris',
 'age': 21,
 'marks': {'Maths': 89, 'Physics': 78,
 'Chemistry': 91},
 'is_sporty': True,
 'total': 258},
 144547: {'name': 'Dev',
 'gender': 'M',
 'city': 'London',
 'age': 23,
 'marks': {'Maths': 88, 'Physics': 77,
 'Chemistry': 98},
 'is_sporty': False,
 'total': 263},
 132399: {'name': 'Mary',
 'gender': 'F',
 'city': 'Paris',
 'age': 22,
 'marks': {'Maths': 99, 'Physics': 87,
 'Chemistry': 88},
 'is_sporty': True,
 'total': 274}}
```

We can print the data inside the dictionary in tabular form using f strings and nested for loops.

```
print(f"{'id':8}{ 'Name':8}{ 'Age':>5}{ 'Maths':>10}{ 'Physics':>10}{ 'Chemistry':>10}{ 'Total':>7}")
```

```

for id_num, student in students.items():
 print(f'{id_num: <8}', end='')
 print(f'{student["name"]:8}', end='')
 print(f'{student["age"]:5}', end='')
 for marks in student["marks"].values():
 print(f'{marks:10}', end='')
 print(f'{student["total"]:7}')

```

### **Output-**

| id     | Name | Age | Maths | Physics | Chemistry |
|--------|------|-----|-------|---------|-----------|
| 105416 | John | 21  | 89    | 78      | 91        |
| 258    |      |     |       |         |           |
| 144547 | Dev  | 23  | 88    | 77      | 98        |
| 263    |      |     |       |         |           |
| 132399 | Mary | 22  | 99    | 87      | 88        |
| 274    |      |     |       |         |           |

## **7.4 Premature termination of loops using the break statement**

Normally, a `while` loop terminates when the loop condition becomes `False`, and a `for` loop terminates when the whole iterable has been iterated over. However, in some situations, we might need to come out of the loop even before the loop condition becomes `False` in a `while` loop or before the iterable is exhausted in a `for` loop. In these cases, we can use the `break` statement to terminate the loop immediately.

The `break` statement is written inside a loop to prematurely terminate it when some particular condition is met. Practically, this `break` statement appears inside an `if` statement, so it executes conditionally. Here is the flowchart of a `while` loop that contains a `break` statement.

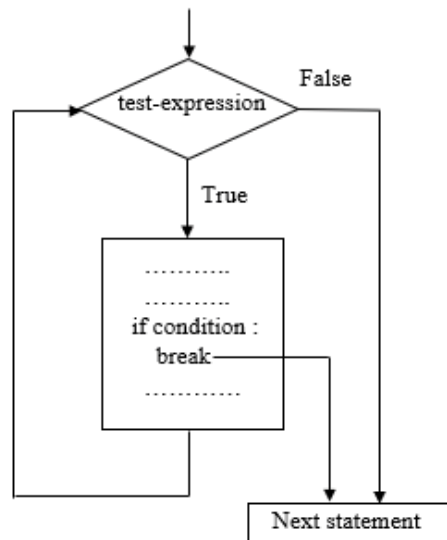


**Figure 7.2:** Flowchart of while loop with a break statement

The loop will keep executing while the test expression (loop condition) is True. It terminates when it becomes False. In any iteration of the loop, if the `if`-condition becomes True, the `break` statement is executed. The loop terminates, and the control goes directly to the next statement out of the loop. It works similarly inside a `for` loop. If the `break` statement is written inside a nested loop structure, it causes an exit from the innermost loop.

So, the `break` statement is used to break out of a loop, even if the loop condition has not become False or the iterable has not been completely iterated over. Let us discuss some examples of the `break` statement.

We have a list of cities, and we want to print the city names from this list till we reach 'Berlin'. Once we print Berlin, we want to stop printing.

```

trip = ['Milan', 'Venice', 'Munich', 'Vienna',
 'Budapest', 'Prague', 'Berlin', 'Amsterdam', 'Paris',
 'Nice']

for city in trip:
 print(city, end=' ')
 if city == 'Berlin':
 break

```

### **Output-**

Milan Venice Munich Vienna Budapest Prague Berlin



In each iteration, the condition is checked, and when it becomes True, the `break` statement executes, and the loop is stopped. Now, only the names till Berlin are printed.

In the next example, we have to find whether there is a negative number in a list of numbers.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99, 72, 94, 12]
for number in numbers:
 if number < 0:
 print('Found a negative number in the list')
 break
```

### ***Output-***

```
Found a negative number in the list
```

We iterate over the `numbers` list using a `for` loop and terminate the loop using a `break` as soon as we find a negative number. If the list has no negative numbers, nothing is printed in the output. We want to print a message in that case also when no negative is present in the list. For that, we can take a variable named `found` and initialize it to `False`. When we find a negative number in the list, we will change it to `True`.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99, 72, 94, 12]
found = False
for number in numbers:
 if number < 0:
 found = True
 print('Found a negative number in the list')
 break
```

If no negative number is found in the list, the variable `found` will be `False` after the loop terminates. So, we can put an `if` statement after the loop to print the information that there is no negative number in the list.

The condition `if found == False` can be written using the `not` operator also.

```
if not found:
```

```
print('No negative number in the list')
```

When `found` will be `False`, `not found` will be `True`. These types of Boolean variables are called flags.

Now, let us write a program to find whether a number is prime. A prime number, as you know, is a whole number greater than one which has only two factors, 1 and itself. This means that a prime number cannot be evenly divided by any number other than 1 and itself. To find whether a number `n` is prime or not, we will divide it by numbers 2,3,4, and so on till `n-1`, and if any of these numbers divides `n` fully, that number is a factor of `n`. This means that `n` is not prime.

We need to check divisibility by numbers from 2 to `n-1`, so we will write a `for` loop with a `range` function that gives us these numbers.

```
n = int(input('Enter a number : '))
for i in range(2, n):
 if n % i == 0:
 break
```

The loop variable `i` takes values from 2 to `n-1`. As soon as we get a number that divides the number `n`, we break out of the loop because we have found a factor, and we can say that `n` is not prime, so there is no need to check till the end. We will introduce a flag in this code.

```
n = int(input('Enter a number : '))
is_prime = True
for i in range(2, n):
 if n % i == 0:
 is_prime = False
 break
if is_prime == True:
 print(f'{n} is prime')
else:
 print(f'{n} is not prime')
```

We have taken a Boolean variable `is_prime` and initialized it to `True`. When a factor is found, we make it `False`. Outside the loop, we have checked the variable

`is_prime` and printed the appropriate message.

We can write the condition `if is_prime == True:` as `if is_prime:` also. It means the same thing.

If we want, we can write the 'not prime' message with the `break` statement, as we had done in the previous program when we found a negative number.

In the loop, we are checking divisibility by numbers till  $n-1$ . Actually, there is no need to check till  $n-1$ ; if we check till  $n//2$ , it is also sufficient. This way, we can make the loop more efficient by reducing the number of iterations.

```
for i in range(2, n//2 + 1):
```

We have discussed how to write nested loops, so now let us enclose the prime number checking code inside another loop to print all prime numbers from 2 to 100.

```
for n in range(2, 100):
 isprime = True
 for i in range(2, n//2 + 1):
 if n % i == 0:
 isprime = False
 break
 if isprime:
 print(n, end=' ')
```

Here instead of inputting `n`, we are getting `n` from the outer loop. We also notice that when a `break` is inside a nested loop structure, it terminates only the closest enclosing loop.

Now, let us see an example of a `break` statement inside a `while` loop. In the following loop, we are adding the numbers entered by the user, and the loop will terminate when the total of numbers exceeds 100.

```
total = 0
while total <= 100:
 num = int(input('Enter a number : '))
 total += num
print(total)
```

Suppose we want this process to stop prematurely if the user enters a negative number. For that, we can use a `break` statement.

```
total = 0
while total <= 100:
 num = int(input('Enter a number : '))
 if num < 0:
 break
 total += num
print(total)
```

This loop will stop naturally when the total exceeds 100, and it will stop prematurely when a negative number is entered.

## 7.5 continue statement

The `break` statement terminates the loop, but there may be situations when we need to terminate only the current iteration, not the whole loop. In these cases, we can use the `continue` statement to jump directly to the next iteration without finishing the current iteration.

Like the `break` statement, the `continue` statement is allowed only inside a loop body and with an `if` condition. When a `continue` statement is encountered in a loop body, it does not execute the remaining statements of the current iteration and immediately takes control to the top of the loop. In a `while` loop, the control goes to the test expression, and in the `for` loop, the next item from the collection is processed. So, the `continue` statement terminates the current iteration and continues with the next iteration of the loop. The following figure shows the flowchart of a `while` loop with a `continue` statement.

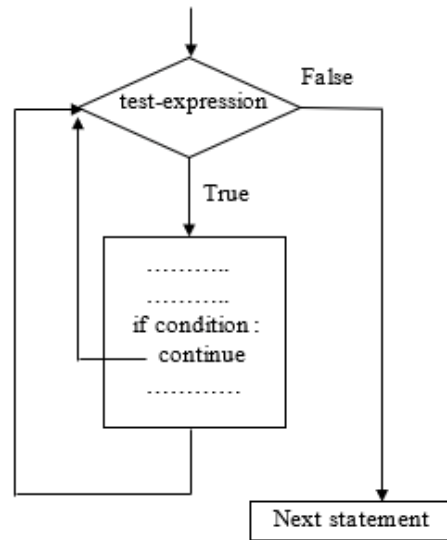


**Figure 7.3:** Flowchart of while loop with a continue statement

The `while` loop executes as usual. If, in any iteration, the `if`-condition is `True`, then the `continue` statement executes, and the control is transferred to the top of the loop. The rest of the statements of the loop are not executed for that iteration. So, when the `continue` statement executes, the rest of the loop body is skipped, and the loop continues with the next iteration.

If the `continue` statement is present inside a nested loop structure, it takes the control to the top of the closest enclosing loop. Let us take some small examples to understand this statement.

We can use the `continue` statement inside a `for` loop when we do not want to process some elements of the iterable that are being iterated over.

```
for i in range(100):
 if i % 10 == 0:
 continue
 print(i)
```

This loop prints the numbers from 0 to 99, except those divisible by 10. When `i` is divisible by 10, the `continue` statement executes, `print(i)` is skipped, and control goes to the top of the loop and iterates for the next number generated by `range`.

Now let us see an example of a `continue` statement inside a `while` loop. We saw this program that stores the user input in a dictionary.

```

fruit_prices = {'apple': 210, 'banana': 100, 'grapes':
90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)

```

Suppose we do not want to enter those fruit and price pairs in the dictionary for which the price is greater than 200. In an iteration, if the price is greater than 200, we can use `continue` to skip the rest of the statements of the loop.

```

fruit_prices = {'apple': 210, 'banana': 100, 'grapes':
90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 if price > 200:
 print('Price more than 200 not allowed')
 continue
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)

```

Now, when the price is greater than 200, the rest of the statements in the current iteration are bypassed, and control will go to the top of the loop, and the next iteration will start.

We could have written this one without the `continue` statement like this:

```
fruit_prices = {'apple': 210, 'banana': 100, 'grapes':
90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 if price > 200:
 print('Price more than 200 not allowed')
 else:
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)
```

This will work in the same way as the previous one. Now, suppose we had a lot of things to be done when the price is okay ( $\leq 200$ ).

```
fruit_prices = {'apple': 210, 'banana': 100, 'grapes':
90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 if price > 200:
 print('Price more than 200 not allowed')
 else:
 print('Do something')
 print('Do something')
 fruit = fruit.lower()
```

```

 if price < 30:
 price += 10
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)

```

There is a lot of code in the part when the price is not more than 200. Let us write the same program using the `continue` statement.

```

fruit_prices = {'apple': 210, 'banana': 100, 'grapes':
90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 if price > 200:
 print('Price more than 200 not allowed')
 continue
 print('Do something')
 print('Do something')
 fruit = fruit.lower()
 if price < 30:
 price += 10
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)

```

We can compare the two ways of writing this code. Both of them give the same results; one uses `continue`, and the other one does not. We can see that using `continue` in the code makes the code more readable, as we can avoid statement



nesting. If the price is more than 200, just skip the rest of the loop body; there is no need to indent the code.

Let us discuss one more example to explore the usefulness of the `continue` statement.

```
student_marks = {'Sam': [46, 37, 38],
 'Pam': [99, 97, 95],
 'Ria': [45, 63, 55],
 'Joe': [34, 36, 34],
 'Jim': [99, 97, 96],
 'Ted': [33, 24, 51],
 'Tim': [78, 98, 79]
 }

for name, marks in student_marks.items():
 total = sum(marks)
 percentage = total / 3
 if percentage < 60:
 grade = 'C'
 elif percentage < 90:
 grade = 'B'
 else:
 grade = 'A'
 if percentage > 95:
 print(f'{name} awarded a scholarship')
 print(f'{name} gets {grade} grade', end=' ')
 print(f'with {percentage:.1f} marks\n')
```

In this program, we have a dictionary with student names as keys and a list of marks as the values. In the `for` loop, we are calculating the total of each student and then the percentage assuming 100 as the maximum mark for each subject.

Then, based on the percentage, we calculate the grade. If the student gets an A grade with more than 95 percent marks, he or she gets a scholarship.

Suppose we want to calculate the percentage, grade, etc., only if the student's total is more than or equal to 120. To do this, we can place a continue statement.

```
for name, marks in student_marks.items():
 total = sum(marks)
 if total < 120:
 print(f'{name} failed the exam\n')
 continue
 percentage = total / 3
 if percentage < 60:
 grade = 'C'
 elif percentage < 90:
 grade = 'B'
 else:
 grade = 'A'
 if percentage > 95:
 print(f'{name} awarded a scholarship')
 print(f'{name} gets {grade} grade', end=' ')
 print(f'with {percentage:.1f} marks\n')
```

If the total is less than 120, the rest of the statements will be skipped, and the next iteration will start. Let us try to write the same thing without the continue statement.

```
for name, marks in student_marks.items():
 total = sum(marks)
 if total < 120:
 print(f'{name} failed the exam\n')
 else:
 percentage = total / 3
```

```

if percentage < 60:
 grade = 'C'
elif percentage < 90:
 grade = 'B'
else:
 grade = 'A'
 if percentage > 95:
 print(f'{name} awarded a scholarship')
print(f'{name} gets {grade} grade', end=' ')
print(f'with {percentage:.1f} marks\n')

```

We get the same result, but this code is less understandable than the one with `continue`.

Although you can achieve similar results using `if` and `else` statements, the `continue` statement provides a more concise and readable way to handle certain situations. Using `continue`, you can avoid writing nested `else` clauses, and thus it prevents the need to increase the indentation level of your code. This leads to cleaner and more readable code, especially in complex loops with multiple conditions.

## 7.6 else block in Loops

A loop can terminate in two ways, either naturally or prematurely. A `while` loop terminates naturally when the test condition becomes `False` and prematurely when `break` is encountered. A `for` loop terminates naturally when the loop has iterated over all items of the iterable and prematurely when the `break` is encountered.

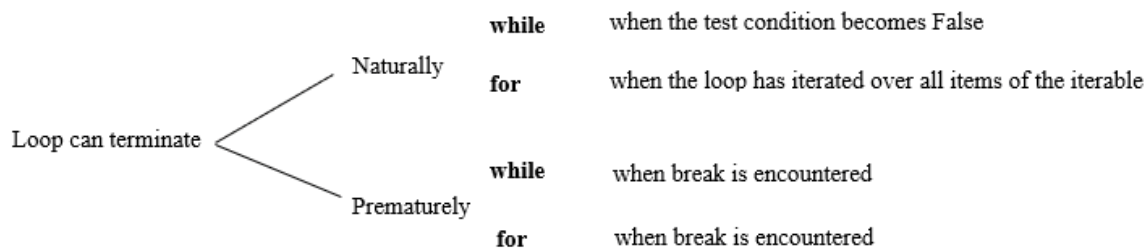


**Figure 7.4:** Termination of a loop

We need to understand this difference because the `else` block of a loop executes only when the loop is terminated naturally. Both `while` and `for` loops can have an `else` clause. Here is the syntax of writing an `else` clause:

```
while test-expression: for item in iterable:
 statement1 statement1
 statement2 statement2
else: else:
 statementA statementA
 statementB statementB
Next statement Next statement
```

The statements in the `else` block will be executed only once when the loop terminates naturally without encountering a `break` in the first block. If the loop ends due to a `break` statement, the `else` block is skipped; statements inside it will not be executed.

If the `else` statement is used in a `for` loop, the `else` block is executed when the loop has exhausted iterating over the iterable. If the `else` clause is used in a `while` loop, the `else` block is executed when the loop condition becomes `False`. So, if you come out of the loop normally without breaking anywhere in between, the `else` block will be executed. The following figure shows the flow chart of a `while` loop with an `else` block.

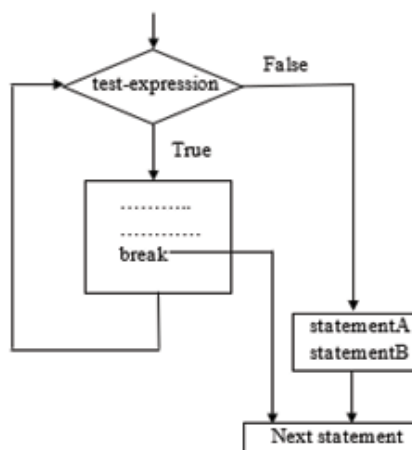


**Figure 7.5:** Flowchart of while loop with else block

We can see that if the loop terminates due to `break`, the `else` block is not executed. But if the loop terminates naturally, the block is executed.

The `else` block is also executed if the loop body is not run even once because, in that case also, the loop exits naturally and not due to `break`. The `for` loop will not execute even once if the iterable is empty, and the `while` loop will not execute even once if the condition is `False` the first time through the loop.

The `else` block is mostly used to replace the search status flags. Let us see some examples that use `else` block.

Here is the program that we have seen earlier in the section on the `break` statement.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99]
found = False
for number in numbers:
 if number < 0:
 print('Found a negative number')
 found = True
 break
if not found:
 print('No negative number in the list')
```

If we use an `else` block in this loop, there will be no need for the flag.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99]
for number in numbers:
 if number < 0:
 print('Found a negative number')
 break
else:
 print('No negative number in the list')
```

The `else` block will be executed only when the `for` loop terminates naturally, i.e., when the full list has been iterated over. If any negative number is found, the `break` statement will execute, the loop will terminate prematurely, and so the

`else` block will not be executed. Similarly, in the following program that we have seen before, we can get rid of the `isprime` flag if we use the `else` block.

```
for n in range(2, 100):
 is_prime = True
 for i in range(2, n // 2 + 1):
 if n % i == 0:
 break
 else:
 is_prime = True
 print(n, end=' ')
 if is_prime:
 print(n, end=' ')
```

The code with the `else` block is more concise and elegant. The `else` block is particularly useful when performing searches, as it provides a natural way to handle the case of search failure without the need for additional flags or variables.

```
for item in iterable:
 if desired item found:
 break
else:
 desired item not present
```

The use of an `else` block might not seem very intuitive because of its name. The name can be misleading, as it may suggest that the `else` block will execute if the loop body does not execute normally. However, the opposite is true, which can confuse readers. Therefore, the `else` block is not commonly used in practice as it can be hard to understand for those unfamiliar with its behavior.

## 7.7 pass statement

When a `pass` statement is executed, nothing happens. It is just a null operation or a do-nothing statement. It is used as a placeholder when the syntax requires a statement, but you do not want to execute anything. Here is an example:

```
if x >= 0:
 pass
else:
 x += 2
```

The `pass` statement does nothing, so if the value of `x` is greater than or equal to 0, nothing will be done. Here we have used it to fill the syntactic requirements of the `if` statement. If we leave the place empty, we will get a syntax error. You might say that we can invert the condition and write the code like this:

```
if x < 0:
 x += 2
```

We can do this, but sometimes it is better to explicitly state the ignoring of a certain condition. In the previous code, we are explicitly stating that if `x >= 0`, nothing has to be done.

Moreover, after some time, you can decide to take an action when `x >= 0` and place the real code instead of the `pass` statement. So, you can use the `pass` statement as the placeholder if you have not decided what code has to be written. We will use this statement in functions and classes that we will discuss later.

```
def func(arg):
 pass # function that does nothing, code will be
added later on
```

Here, we have defined a function, but the code for it will be decided later on. So, you can use the `pass` statement in the initial stages of writing a program. This `pass` statement can also be used for ignoring exceptions that are caught by `try` statements. We will see that later on in another chapter.

Thus, `pass` is the empty placeholder statement of Python. It is used when the syntax wants you to execute something, but you do not have anything to execute, so you satisfy the syntax by executing nothing.

It is mostly used to represent an empty body of a compound statement while initially writing a program. In other languages like C or Java, you can represent an empty block by a pair of empty braces, but in Python, blocks are determined by indentation and not by braces. If you need an empty block, you cannot simply leave the place empty; you need to write the `pass` statement. In Python 3, you can even use three consecutive dots known as ellipses (...) for the same purpose.

```
if x >= 0:
 ...
else:
 x += 2
```

## 7.8 for loop vs. while loop

We have seen that Python provides two loops to perform repetitive tasks - for loop and while loop. Both of them serve different purposes, but it is possible to perform some tasks using either of them. It would be better if you clearly understand the difference between the two and know when to use which one.

The `while` loop is a condition-controlled loop because the number of iterations of the loop is determined by the condition. The `for` loop is a collection-controlled loop since it iterates over a collection of things. It can be used as a counter-controlled loop also using the `range` function.

The `while` loop is an indefinite loop because before the loop executes, we cannot always tell how many times it will execute. It runs indefinitely until some condition is met. The `for` loop is a definite loop because before the loop executes, we know exactly how many times it will execute. The number of times it iterates depends on the size of the collection.

Use a `while` loop when you do not know in advance how many times to repeat the task, but you know when to stop repeating. Use a `for` loop when you have to iterate over a collection of things, i.e., when you want to perform an action on every item in a collection. You can also use a `for` loop when you have to repeat a task a fixed number of times.

The `while` statement can be used to write both definite and indefinite loops, but `for` loop is specifically made for definite loops. So, whenever you know ahead of time how many times to iterate, use a `for` loop. For example, printing of numbers from 1 to 10 should be done using a `for` loop, although we can do it using a `while` loop also.

You have to be careful while writing your `while` loops, as you can write a loop that never ends. In a `while` loop, we have to write the update step, which generally updates the variables used in the loop condition. If we forget to write the update step or write it in such a way that the loop condition never becomes False, our program will be stuck in an infinite loop. So, it is important to write your



update step correctly in order to avoid an infinite loop. In the `for` loop, we have an in-built update step, and the loop knows when to stop, so the infinite loop problem will generally not arise in a `for` loop.

In the next chapter, we will discuss how to take advantage of an intentionally created infinite loop. So, in case you need to deliberately create an infinite loop, a `while` loop has to be used.

## Exercise

How many times will the following loops iterate?

1. `while True:`

```
 print('I love Python')
```

2. `x = 10`

```
 while x < 1:
```

```
 x -= 2
```

```
 print(x)
```

3. `x = 10`

```
 while x != 1:
```

```
 x -= 2
```

```
 print(x)
```

4. How many iterations will the following loop have if the value of `i` is (i) 5 (ii) `<5` (iii) `>5` ?

```
i = int(input('Enter the value of i : '))
```

```
while i != 5:
```

```
 print(i)
```

```
 i += 1
```

What will be the output of the code given in questions 5 to 19?

5. `x = 5`

```
while x:
```

```
 x -= 1
```

```

 print(x, end=' ')
6. for item in [1, 2, 3]:
 print(item * 4, end=',')
 print(item)
7. s = 'Hello World'
 count = 0
 for ch in s:
 if ch.isupper():
 count += 1
 print(count)
8. L = [8, 2, -3, 4, -5, 6]
 s = 0
 for i in L:
 if i > 0:
 s += i
 print(s)
9. s = 'abc'
 for ch in s:
 print(s, end='')
10. L = [[1,2,3], [4,5,6], [7,8,9]]
 for n, _, _ in L:
 print(n, end=' ')
11. for item in [1, 2, 3, 4]:
 print(item if item%2 == 0 else 0,
 end=' ')
12. for n in range(5, 15, 3):
 print(2 * n, end=' ')
13. text = 'Be happy. Be bright. Be you.'

```

```

s = ''
for ch in text:
 if ch == '.':
 break
 s += ch
print(s)
14. for n in range(10, 20):
 isprime = True
 for i in range(2, n):
 if n % i == 0:
 break
 isprime = False
 if isprime:
 print(n, end=' ')
15. for i, _ in [('x', 2), ('y', 5)]:
 for j in (6, 7):
 print(i, j, end=' ')
16. for i in range(1, 10):
 if i % 3 == 0:
 continue
 print(i, end=' ')
17. n = 2345
sum = 0
while n > 0:
 rem = n % 10
 sum += rem
 n //= 10

```

```

 print(sum)
18. listA = [1, 2, 3, 4]
 listB = []
 while listA:
 listB.append(3 * listA.pop())
 print(listA, listB)
19. D = {'Mark': 25, 'Tom': 65, 'John': 37, 'Rob': 45}
 for name, age in D.items():
 if age > 65:
 print(name)
 break
 else:
 print('No senior citizens')
20. Will these two loops give the same output?
 L = [2, 4, 5, 3, 7, 9, 6]
 for n in L:
 print(n, end=' ')
 i = 0
 while i < len(L):
 print(L[i], end=' ')
 i += 1
21. Will these two loops give the same output?
 D = {'a': 1, 'b': 2}
 for x in D:
 print(x)
 for x in D.keys():
 print(x)
22. The break statement can be written only inside a loop.

```

(A) True (B) False

23. Write code to find the sum of first n natural numbers using a:

(i) `while` loop

(ii) `for` loop

(iii) without any loop

24. Factorial of a non-negative integer, n is the product of all positive integers less than or equal to n. For example, the factorial of 6 is equal to  $6*5*4*3*2*1 = 720$ . Write a program to find out the factorial of a given number using a while loop.

25. Write a program that counts the number of vowels, consonants, and digits in a string.

26. Write a `for` loop to find the product of all the numbers in a list.

27. Write a program to count the frequency of all characters in a string. Store the result in a dictionary in which keys are characters of the string, and the corresponding values are the number of occurrences of the characters. For example, for the string 'Hello world !!! ', the resulting dictionary should be:

```
{ 'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 5, 'w': 1, 'r': 1, 'd': 1, '!': 3 }
```

28. Write a program to count the frequency of all words in a string. Split the string into words using whitespace as the separator. Store the result in a dictionary in which keys are words of the string, and the corresponding values are the number of occurrences of the words. For example, for the string, 'Humpty Dumpty sat on a wall Humpty Dumpty had a great fall ' the resulting dictionary should be:

```
{ 'Humpty': 2, 'Dumpty': 2, 'sat': 1, 'on': 1, 'a': 2, 'wall': 1, 'had': 1, 'great': 1, 'fall': 1 }
```

Modify your program to work even if the string contains numbers and punctuation characters.

29. Given a text string, create a dictionary in which keys are five vowels and values are the frequencies of those vowels in the string.

30. Write a program to multiply two numbers using the Russian Peasant Method. In this method, any two numbers can be multiplied using only multiplication by 2, division by 2, and addition.

To multiply two numbers, divide the first number by 2 (integer division) and multiply the second number by 2 repeatedly till the first number reduces to 1. Suppose we have to multiply 38 and 16.

|           |            |
|-----------|------------|
| 38        | 16         |
| <b>19</b> | <b>32</b>  |
| <b>9</b>  | <b>64</b>  |
| 4         | 128        |
| 2         | 256        |
| <b>1</b>  | <b>512</b> |

We stopped when the first number was reduced to 1. To get the product, we will add those values on the right-hand side, for which the corresponding left-side value is odd. On adding 32, 64, and 512, we get 608, which is the product of 38 and 16.

31. Write a loop to censor certain words in a text by replacing them with asterisks. The words to be replaced are given in a list. Here is an example:

```
s = '''A group of fearless rebels emerged,
unafraid to be labelled as crazy or mad.
```

```
Others called them mad troublemakers, but their
insane ideas held the power to change the world.
These visionaries proved that it is often the
seemingly insane ones who hold the key to
progress.'''
```

```
L = ['crazy', 'mad', 'rebels', 'lunatic',
'troublemakers', 'insane']
```

The string `s` after replacement should look like this:

```
A group of fearless ***** emerged, unafraid to be
labelled as ***** or ***. Others called them ***
*****, but their ***** ideas held the
power to change the world. These visionaries
proved that it is often the seemingly ***** ones
who hold the key to progress.
```

32. Change the code written in the previous question such that for all words that are to be censored, only the first letter is displayed. For the rest of the

word, asterisks are displayed. Taking the example string `s` and list `L` of the previous question, the string `s` should look like this after replacement.

A group of fearless r\*\*\*\*\* emerged, unafraid to be labelled as c\*\*\*\*\* or m\*\*. Others called them m\*\*t\*\*\*\*\*, but their i\*\*\*\*\* ideas held the power to change the world. These visionaries proved that it is often the seemingly i\*\*\*\*\* ones who hold the key to progress.

33. The following code gives an error if the user enters any non-numeric value.

```
age = int(input('Enter your age : '))
print(age)
```

Rewrite the above two lines of code so that the user is forced to enter a numeric value for age, which should be between 10 and 100.

34. From the following dictionary named `students`, create two sets named `toppers` and `champions`. In the `toppers` set, add names of those students who have got more than 90 marks, and in the `champions` set, add names of those students who have more than 4 sports medals.

```
students = {'id11': {'name': 'Amit', 'marks': 97,
 'sports_medals': 0},
 'id12': {'name': 'Dev', 'marks': 92,
 'sports_medals': 6},
 'id13': {'name': 'Ted', 'marks': 81,
 'sports_medals': 2},
 'id14': {'name': 'Rob', 'marks': 96,
 'sports_medals': 1},
 'id15': {'name': 'Sam', 'marks': 56,
 'sports_medals': 1},
 'id16': {'name': 'Pam', 'marks': 66,
 'sports_medals': 7},
 'id17': {'name': 'Ram', 'marks': 98,
 'sports_medals': 9},
 'id18': {'name': 'Tim', 'marks': 66,
 'sports_medals': 5},
```

```
}
```

35. The following program is written for creating two lists named `evens` and `odds` from the list `numbers`. This code does not give the correct output. Can you find out what the problem is?

```
numbers = [10, 2, 3, 41, 5, 7, 8, 9, 62]
evens = odds = []
for number in numbers:
 if number % 2 == 0:
 evens.append(number)
 else:
 odds.append(number)
print(evens)
print(odds)
```

36. From the following set, make another set of all the names that start with an underscore.

```
names = {'_num', 'var', 'product', '_add', '_sub',
'square'}
```

37. The following dictionary has fruit names as keys and prices as values.

```
D = {'apple': 100, 'grapes': 55, 'banana': 200,
'guava': 60}
```

Write a for loop that iterates over this dictionary and increases the price of fruit by 10 if its price is less than 100. Otherwise, it decreases the price by 10. This should be the resulting dictionary.

```
{'apple': 90, 'grapes': 65, 'banana': 190,
'guava': 70}
```

38. What is wrong with the following code?

```
L = [['John', [88, 89, 78]], ['Sam', [89, 76,
99]], ['Dev', [85, 67, 89]]]
for name, m1, m2, m3 in L:
 total = m1 + m2 + m3
```



```
print(name, total)
```

39. Write a program to create this dictionary in which the keys are numbers and values are their squares.

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

40. Write a program to enter more student records inside the following dictionary.

```
students = {105416: {'name': 'John',
 'age': 21,
 'marks': {'Maths': 89,
 'Physics': 78,
 'Chemistry': 91}
 },
 144547: {'name': 'Dev',
 'age': 23,
 'marks': {'Maths': 88,
 'Physics': 77,
 'Chemistry': 98}
 },
 132399: {'name': 'Mary',
 'age': 22,
 'marks': {'Maths': 99,
 'Physics': 87,
 'Chemistry': 88},
 }
}
```

41. Write a program that finds the shortest and the longest string from a list of strings. Use a for loop to iterate over the list of strings.
42. Write a program that inserts all common items of the following 2 lists into a third list L3.

```
L1 = ['China', 'Brazil', 'India', 'Iran', 'Iraq',
 'Russia']
```

```
L2 = ['Italy', 'Japan', 'China', 'Russia',
 'Nepal', 'France']
```

```
43. D = {'pen': 10, 'pencil': 5, 'eraser': 8,
 'marker': 15, 'ruler': 19}
```

Draw the following chart for the dictionary D.

pen -----

pencil ----

eraser -----

marker -----

ruler -----

```
44. D = {'pen': 10, 'pencil': 5, 'eraser': 8,
 'sharpener': None, 'marker': 15, 'ruler': None}
```

In this dictionary, keys are names of fruits, and values are their respective prices. For fruits that are out of stock, the price is marked as None. Iterate over this dictionary and print only those fruits with their prices that are in stock. Use `ljust()` and `rjust()` methods of `str` type to align your output.

45. Create a randomized list of size 10 that contains random numbers in the range 1 to 50. Use `randint` function from the `random` module.

46. Fibonacci series is a series of numbers in which each number is the sum of previous two numbers.

0 1 1 2 3 5 8 13 21 34 55 89 144 233

(i) Print first `n` Fibonacci numbers using a for loop.

(ii) Print all Fibonacci numbers less than a number `n`, using a while loop

47. Create a list of all the methods of `str` type that start with '`is`' (use `dir` function).

48. Write a program to print these pyramids without using nested loops.

|      |      |      |      |      |
|------|------|------|------|------|
| *    | 1    | A    | *    | *    |
| **   | 22   | BB   | **   | **   |
| ***  | 333  | CCC  | ***  | ***  |
| **** | 4444 | DDDD | **** | **** |
|      |      | EEEE |      |      |

*Figure 7.6*

49. Write a program to print these pyramids using nested for loops.

|      |      |             |             |      |      |      |
|------|------|-------------|-------------|------|------|------|
| 1    | 1    | 10          | 10          | A    | A    | A    |
| 12   | 22   | 10 11       | 11 12       | AB   | BB   | BC   |
| 123  | 333  | 10 11 12    | 13 14 15    | ABC  | CCC  | DEF  |
| 1234 | 4444 | 10 11 12 13 | 16 17 18 19 | ABCD | DDDD | GHIJ |

*Figure 7.7*

50. Write a program that creates a list of all prime numbers from 100 to 300.

51. Write a program that simulates dice rolling. Use `randint` from `random` module.

52. Write a program that adds numbers entered by the user. Stop entering when user enters 0. Do not add numbers that are negative or greater than 500.

53. What is wrong with this program written to find a value in a list?

```
L = [1, 2, 4, 5, 6, 8, 9]
target = 3
found = False
for n in L:
 if n == target:
 found = True
 print(f'{target} found')
 break
 else:
 print(f'{target} not found')
```

54. Here is a text string and a list of prohibited words.

```
text = 'It is often the seemingly insane ones who
hold the key to progress'
prohibited_words = ['mad', 'insane', 'crazy']
```

Use a `for` loop with `else` block to find whether the text string contains any prohibited word. If you find a prohibited word, display 'Found a prohibited word.' If the text string does not contain any prohibited word, then display 'No prohibited word in the list'.

55. Draw a flowchart for a `while` loop that shows `break`, `continue`, and `else` blocks.

56. Write a `for` loop to print all divisors of a number.

57. Find the smallest divisor of a number greater than 1, using (i) a `while` loop (ii) `for` loop

```
58. fruits = {'apple', 'banana', 'grapes'}
 veggies = {'potato', 'onion', 'cabbage'}
 stationery = {'pencil', 'eraser', 'sharpener',
 'marker'}

 prices = {'pencil': 10, 'eraser': 5, 'sharpener':
4, 'marker': 20, 'potato': 30,
 'onion': 25, 'cabbage': 22, 'apple': 90,
'banana': 60, 'grapes': 80}
```

Write a `for` loop to increase the price of all items by 10%, except fruits (use `continue` statement). Rewrite the loop without `continue` statement.

59. We saw the following loop in the section on `break` statement. Modify this so that all the non-prime (composite) numbers are also printed.

```
for n in range(2, 100):
 isprime = True
 for i in range(2, n):
 if n % i == 0:
 isprime = False
 break
 if isprime:
 print(n)
```

The output should be of this form.

```
2 is prime
```

```
3 is prime
4 is not prime as 4 = 2 * 2
5 is prime
6 is not prime as 6 = 2 * 3
7 is prime
8 is not prime as 8 = 2 * 4
9 is not prime as 9 = 3 * 3
```

60. Write a program to find all occurrences of a substring in a string.

61. The following string represents data that contains names and ages.

```
data = 'Amit:20,Sumit:30,Namit:34,Dev:23,Ankur:32'
```

Write a program to convert this string into a dictionary of this form:

```
D = {'Amit': 20, 'Sumit': 30, 'Namit': 34, 'Dev':
23, 'Ankur': 32}
```

### **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# Looping Techniques

---

## 8

Loops are used extensively in programming, so we must write loops that are readable, concise, and efficient. In this chapter, we will discuss some common looping techniques and idioms that can help make our code more Pythonic.

## 8.1 Iterating in sorted and reversed order

Whenever we print the elements of a list using a `for` loop, the elements will be printed in the order they appear because the list elements have an inherent order, and the iteration is defined in that order only.

```
numbers = [2, 1, 4, 6, 3]
for number in numbers:
 print(number, end=' ')
```

### **Output-**

2 1 4 6 3

If we want the iteration done in a sorted order of elements, we can use the `sorted` function.

```
for number in sorted(numbers):
 print(number, end=' ')
```

### **Output-**

1 2 3 4 6

Similarly, we can loop over a sequence in reverse order using the `reversed` function.

```
for number in reversed(numbers):
 print(number, end=' ')
```

***Output-***

3 6 4 1 2

The advantage of using these functions in the `for` loop is that we can iterate over the sequence elements in a different order, and our original sequence remains unchanged.

We can use the `sorted` function on sets also to iterate over them in sorted order. We know that there is no inherent order among the elements of a set, so if we write a `for` loop to iterate over the elements of a set, the elements are iterated in no particular order.

```
primes = {31, 3, 5, 11, 2, 13, 17, 43, 19, 7, 37,
23, 29, 41}
for number in primes:
 print(number, end=' ')
```

***Output:***

2 3 37 5 7 41 11 43 13 17 19 23 29 31

If we write the `sorted` function, the elements will be iterated in sorted order.

```
for number in sorted(primes):
 print(number, end=' ')
```

***Output:***

2 3 5 7 11 13 17 19 23 29 31 37 41 43

If we want the elements in reversed sorted order (descending order), we can apply the `reversed` function on the output of `sorted` function.

```
for number in reversed(sorted(primes)):
 print(number, end=' ')
```

***Output:***

```
43 41 37 31 29 23 19 17 13 11 7 5 3 2
```

A better approach to get this result would be to set the reverse parameter of the sorted function to True.

```
for number in sorted(primes, reverse=True):
 print(number, end=' ')
```

**Output:**

```
43 41 37 31 29 23 19 17 13 11 7 5 3 2
```

Dictionaries can also be iterated over in sorted order using the sorted function.

```
prices = {'apple': 210, 'banana': 100, 'grapes':
90, 'mango': 250, 'cherry': 225, 'guava': 80}
```

```
for fruit in sorted(prices.keys()):
 print(fruit, prices[fruit], end=' | ')
```

```
print()
```

```
for fruit, price in sorted(prices.items()):
 print(fruit, price, end=' | ')
```

**Output:**

```
apple 210 | banana 100 | cherry 225 | grapes 90 |
guava 80 | mango 250 |
```

```
apple 210 | banana 100 | cherry 225 | grapes 90 |
guava 80 | mango 250 |
```

Here, the sorting is done based on keys; if we want to sort according to values, we can invert the keys and values by using the zip function.

```
for fruit, price in sorted(zip(prices.values(),
prices.keys())):
```

```
 print(fruit, price, end=' | ')
```

**Output-**

```
80 guava | 90 grapes | 100 banana | 210 apple | 225
cherry | 250 mango |
```



Now, the items are iterated over in sorted order of values. Later in the book, we will explore a better approach to sorting based on values by utilizing lambda functions.

We have seen that from Python 3.8 onwards, the dictionary views are reversible, so we can iterate over the dictionary using the `reversed` function.

## 8.2 Iterating over unique values

In the following example, we have written a loop that iterates over a list of numbers, and in each iteration, we are printing the number and its square.

```
L = [2, 3, 1, 4, 5, 7, 4, 2, 1, 3]
for i in L:
 print(f'square of {i} is {i * i}')
```

If we enclose our list in the `set` function, then the loop will iterate over only unique elements of the list.

```
L = [2, 3, 1, 4, 5, 7, 4, 2, 1, 3]
for i in set(L):
 print(f'square of {i} is {i * i}')
```

### **Output-**

```
square of 1 is 1
square of 2 is 4
square of 3 is 9
square of 4 is 16
square of 5 is 25
square of 7 is 49
```

In this process, the order of elements in the list is lost, as there is no order among the elements of a set. Let us discuss another example. We have seen the following code to print only the vowels from a phrase.

```
phrase = 'colourful umbrella'
for ch in phrase:
```

```
if ch in {'a', 'e', 'i', 'o', 'u'}:
 print(ch, end=' ')
```

### **Output-**

```
o o u u u e a
```

If you do not want to print duplicate vowels, you can iterate over the set of characters in the string.

```
phrase = 'colourful umbrella'
for ch in set(phrase):
 if ch in {'a', 'e', 'i', 'o', 'u'}:
 print(ch, end=' ')
```

### **Output-**

```
a o e u
```

We have used the `set` function, so iteration is done only on unique values of the phrase. Similarly, we can use the `set` function to iterate over unique values of any list, tuple, or dictionary.

## **8.3 Index-Based for loops**

If we want to iterate over the indices of a sequence, we need to create a sequence of numbers that represent indices. This can be done by using the `range` function. If we have a list of length 6, the indices of elements would be 0, 1, 2, 3, 4, and 5, and these indices can be generated by the call `range(6)`.

```
data = [2, 3, 1, 4, 7, 9]
for i in range(6):
 print(data[i], end=' ')
```

### **Output-**

```
2 3 1 4 7 9
```

In this loop, we are not iterating over the elements of the list. Instead, we are iterating over the indices of the elements in the list. The iterating variable `i` will not be assigned elements of the list. It will be assigned integer values

from 0 to 5 that are generated by the `range` function. Inside the loop, we have written `data[i]` to access the element at index `i` of the list. If you do not want to hardcode the length of the list, then you can use the `len` function that will give the length of the list, which can be sent to the `range` function to generate indices.

```
for i in range(len(data)):
 print(data[i], end=' ')
```

### **Output-**

2 3 1 4 7 9

This way, we can write index-based `for` loops for sequences by combining `range` and `len` functions. The following `for` loop will give the same output as the previous one.

```
for item in data:
 print(item, end=' ')
```

### **Output-**

2 3 1 4 7 9

In this loop, we are iterating over items while in the previous one we were iterating over index values. In this loop, the loop variable `item` is assigned values of the list and we print `item` in each iteration. Although both the loops will give the same output, this one is simpler than the previous one and also runs faster. However, there are special cases when you will need to write an index-based loop. In the next 2 sections we will see those cases.

## **8.4 Making in-place changes in a list while iterating**

Suppose we have a list of numbers, and we want to add 5 to each element of the list. We have written an index-based `for` loop and an item based `for` loop to achieve this.

```
data = [2, 3, 1, 4, 7, 5]
for i in range(len(data)):
 data[i] += 5
```

```
print(data)
```

### **Output-**

```
[7, 8, 6, 9, 12, 10]
data = [2, 3, 1, 4, 7, 5]
for item in data:
 item += 5
```

### **Output-**

```
[2, 3, 1, 4, 7, 5]
```

When we used the index-based `for` loop, the list is modified and, 5 is added to each element of the list. When we used the item based `for` loop, the list remains unchanged. Let us understand why this is happening.

First, let us see how the item-based `for` loop is working. In the first iteration, the variable `item` is assigned the first element of the list, so the value of `item` becomes 2, then inside the loop 5 is added to variable `item`, so then the value becomes 7. In the next iteration, `item` is assigned the second value of list, so `item` becomes 3 and then 5 is added to it, so the value becomes 8. Thus, whatever is happening, is happening to the loop variable `item`; it is being reassigned and updated. The elements of the list remain unchanged.

In the index-based `for` loop, the list elements get updated directly.

#### Iterating over Sequence index values

```
i = 0
data[i] += 5
i = 1
data[i] += 5
i = 2
data[i] += 5
i = 3
data[i] += 5
i = 4
data[i] += 5
i = 5
data[i] += 5
```

#### Iterating over Sequence items

```
item = 2
item += 5
item = 3
item += 5
item = 1
item += 5
item = 4
item += 5
item = 7
item += 5
item = 9
item += 5
```

**Figure 8.1:** Comparison of iterating over sequence index values and sequence items

Therefore, in this particular problem where we wanted to add something to each item of the list, we have to use the index-based for loop.

If the data present inside the list is mutable, then in-place changes can be made to it by calling its methods by using the `item` identifier. The reassignment of the identifier `item` has no effect on the original data. Let us understand this with the help of an example.

We have a list of lists, and we have written an item-based for loop. In each iteration, we remove the first element from the list.

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for item in L:
 item.pop(0)
print(L)
```

### ***Output-***

```
[[2, 3], [5, 6], [8, 9]]
```

Here is another loop that iterates over the same list but it reassigns `item` each time.

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for item in L:
 item = item * 3
print(L)
```

### ***Output-***

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Now, we cannot see any change in the list. It is because in this loop the variable `item` is reassigned in each iteration. In the previous loop, in-place changes were made in the inner lists so we could see the changes. It is important to understand the difference between the two things: in-place change and reassignment.

Now, let us use the augmented assignment syntax in the loop instead of the multiplication operator.

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for item in L:
```

```
 item *= 3
print(L)
```

### **Output-**

```
[[1, 2, 3, 1, 2, 3, 1, 2, 3], [4, 5, 6, 4, 5, 6, 4, 5, 6], [7, 8, 9, 7, 8, 9, 7, 8, 9]]
```

Now the list was changed because the augmented assignment makes in-place changes in the list. It will not just reassign the variable `item`, it changes the list in-place, we had seen this in [Chapter 4](#).

## **8.5 Skipping some items while iterating**

With index-based `for` loops, you can specify a different start and end value, while with the standard `for` loop, you have to iterate over all items. In the following example, we have a list of 10 elements, and we want to iterate over only the first 4 elements of the list. We can do this by using an index-based `for` loop:

```
data = [2, 1, 3, 4, 5, 6, 9, 1, 7, 8, 9]
for i in range(4):
 print(data[i], end=' ')
```

### **Output-**

```
2 1 3 4
```

So, if you want to iterate over a part of the sequence, instead of all the items in the sequence, you can use the `range` function to generate desired indices. You can use it to skip some items while iterating. For example, if you want to print every third item you can write this loop:

```
for i in range(0, len(data), 3):
 print(data[i], end=' ')
```

### **Output-**

```
2 4 9 8
```

You can get the same results by looping over slices of the list in item-based `for` loops.

```
for item in data[:4]:
 print(item, end=' ')
```

**Output-**

```
2 1 3 4
```

```
for item in data[::3]:
 print(item, end=' ')
```

**Output-**

```
2 4 9 8
```

This approach is cleaner and more readable, but slicing makes a separate copy of the list. If your list is large and space is your priority, you can use the index-based approach.

## 8.6 Using range and len combination to shuffle a sequence

The range and len combination can be used to shuffle a sequence.

```
data = [4, 5, 6, 7]
for i in range(len(data)):
 print(data[i:] + data[:i])
```

**Output-**

```
[4, 5, 6, 7]
```

```
[5, 6, 7, 4]
```

```
[6, 7, 4, 5]
```

```
[7, 4, 5, 6]
```

In this loop we have created four reordered lists by joining together different slices of the list.

## 8.7 enumerate function

We have seen index-based `for` loops that iterate over indices and regular item-based `for` loops that iterate over items. While looping over a sequence (list, string or tuple), there can be situations when you need both the index and the item. In these cases, using the `enumerate` function instead of the `range` and `len` combination is cleaner and more Pythonic.

You can generate the index and the corresponding value at the same time by using the `enumerate` function. This function takes in an iterable and returns an object which can be converted into a list of tuples by using the `list` function or can be used directly in a `for` loop.

```
>>> trip = ['Milan', 'Venice', 'Munich', 'Vienna',
 'Budapest', 'Prague']
>>> enumerate(trip)
<enumerate object at 0x00000243E0FDA100>
>>> list(enumerate(trip))
[(0, 'Milan'), (1, 'Venice'), (2, 'Munich'), (3, 'Vienna'), (4, 'Budapest'), (5, 'Prague')]
```

By enclosing the object in the `list` function, we get a list of tuples where each tuple contains a count and an item returned by the list. The count starts with 0 by default. We can make it start from any other number instead of zero; for example, we can start it from 1.

```
>>> list(enumerate(trip,1))
[(1, 'Milan'), (2, 'Venice'), (3, 'Munich'), (4, 'Vienna'), (5, 'Budapest'), (6, 'Prague')]
>>> list(enumerate(trip,100))
[(100, 'Milan'), (101, 'Venice'), (102, 'Munich'), (103, 'Vienna'), (104, 'Budapest'), (105, 'Prague')]
```

Now, let us use the `enumerate` function to display all the items of the list with their index number.

```
>>> for i, city in enumerate(trip, 1):
... print(f'Destination {i} -> {city}')
...
```



```
Destination 1 -> Milan
Destination 2 -> Venice
Destination 3 -> Munich
Destination 4 -> Vienna
Destination 5 -> Budapest
Destination 6 -> Prague
```

In each iteration, `enumerate` object gives a tuple which we are unpacking into the variables `i` and `city`. So, when you have a situation where you want to iterate over items of a sequence and also need the index number, you can use the `enumerate` function.

We have seen the following loop in [Section 7.6](#) of the previous chapter. We used a `break` statement, and the `else` block to print whether a negative number was present in the list.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99]
for number in numbers:
 if number < 0:
 print(f'Found negative number {number}')
 break
else:
 print('No negative number in the list')
```

### **Output-**

```
Found negative number -36
```

When we find a negative number, we print a message and break out of the loop. If we want to know the index where the first negative number was found, we can use the `enumerate` function.

```
numbers = [23, 78, 98, 78, 65, -36, 78, 99]
for i, number in enumerate(numbers):
 if number < 0:
 print(f'Found negative number {number} at
index {i}')
 break
```

```
else:
 print('No negative number in the list')
```

### **Output-**

Found negative number -36 at index 5

Now we get to know the first negative number as well as its index.

Here is one more example where we can use `enumerate` function. We have a list of student names and our objective is to assign each student a roll number, starting from 1000. All the roll numbers and names should be stored in a dictionary named `data` in which roll numbers are used as keys and student names are the values. The roll numbers will serve as the keys in the dictionary, while the student names will be the associated values.

```
students = ['Pam', 'Sam', 'John', 'Ryan', 'Neil', 'Dev']
data = {}
for i, student in enumerate(students, 1000):
 data[i] = student
print(data)
```

### **Output-**

```
{1000: 'Pam', 1001: 'Sam', 1002: 'John', 1003: 'Ryan', 1004: 'Neil', 1005: 'Dev'}
```

## **8.8 Iterating over multiple sequences using zip**

The `zip` function can be used to iterate over multiple sequences of the same length. We have already seen the `zip` function in the chapter on dictionaries. It takes multiple sequences and returns an object that gives us tuples from items that are at the same offsets in those sequences. We can use this `zip` function in for loop to iterate over two or more sequences at the same time.

Suppose we have 3 lists of same length, the first one contains names of people, second one contains their salaries and the third one contains their cities at the corresponding indices.

```
>>> names = ['Amit', 'John', 'Mark', 'Raj']
>>> salaries = [2000, 3000, 2500, 3200]
>>> cities = ['Delhi', 'Chennai', 'Delhi',
 'Bangalore']
>>> zip(names, cities, salaries)
<zip object at 0x0000019C90023940>
```

This zip function returns an iterable object, we need to enclose it in a list to be able to see the tuples.

```
>>> list(zip(names, cities, salaries))
[('Amit', 'Delhi', 2000), ('John', 'Chennai',
3000), ('Mark', 'Delhi', 2500), ('Raj',
'Bangalore', 3200)]
```

The following for loop iterates over the three lists names, cities and salaries by using the zip function.

```
>>> for name, city, salary in zip(names, cities,
salaries):
... print(f'{name} posted in {city} with
{salary}')
...
Amit posted in Delhi with 2000
John posted in Chennai with 3000
Mark posted in Delhi with 2500
Raj posted in Bangalore with 3200
```

We are doing tuple unpacking in the for loop header. This way we can iterate over multiple sequences simultaneously.

Now, suppose we have to increase salaries of all those posted in Delhi. Let us try to do that.

```
>>> for name, city, salary in zip(names, cities,
salaries):
... if city == 'Delhi':
... salary += 1000
```

```
...
>>> salaries
[2000, 3000, 2500, 3200]
```

The salaries were not changed. If you need to make changes in any of these lists while iterating, then you have to use the index-based loop only.

```
>>> for i in range(len(names)):
... if cities[i] == 'Delhi':
... salaries[i] += 1000
...
>>> salaries
[3000, 3000, 3500, 3200]
```

In a subsequent chapter, we will see the `itertools` module, which provides various functions such as `cycle`, `chain`, and combinations that can be useful in looping scenarios.

## 8.9 Modifying a collection while iterating in a for loop

When we try to add or remove elements from a list while iterating over it using a for loop, we get incorrect results. First, we will explore the reasons behind the unexpected results by examining a few examples, and then we will look at the solution. Here is the first example:

```
students = ['Era', 'Ted', 'Rob', 'Joe', 'Amy',
 'Sam', 'Pat', 'Joy', 'Tia']
failed_students = ['Ted', 'Amy', 'Sam']
for student in students:
 if student in failed_students:
 students.remove(student)
print(students)
```

**Output-**

```
['Era', 'Rob', 'Joe', 'Sam', 'Pat', 'Joy', 'Tia']
```

In the `for` loop, we are iterating over the `students` list, and if a name appears in the `failed_students` list, we remove it from the `students` list. In the output, we can see that all failed students have not been removed from the list. 'Sam' is in the `failed_students` list but has not been removed from the `students` list.

In our next example, we are trying to delete negative numbers from a list while iterating over it.

```
numbers = [2, 3, -7, 8, -5, -2, 9, 10]
for number in numbers:
 if number < 0:
 numbers.remove(number)
print(numbers)
```

### **Output-**

```
[2, 3, 8, -2, 9, 10]
```

In this case also, we failed to achieve the desired result as all negative numbers were not removed from the list. Let us try to understand the reason behind this behavior by printing the numbers before performing the check:

```
numbers = [2, 3, -7, 8, -5, -2, 9, 10]
for number in numbers:
 print(number, end=' ')
 if number < 0:
 numbers.remove(number)
print(numbers)
```

### **Output-**

```
2 3 -7 -5 9 10 [2, 3, 8, -2, 9, 10]
```

We can see that all the items of the list were not covered by the `for` loop, it skipped 8 and -2. To understand the reason for this, let us first see how the `for` loop works internally. The `for` loop works by keeping an internal counter to keep track of which item will be used next. This counter is incremented at the end of each iteration and when this counter is equal to the current length of the iterable, the loop terminates. So basically, the `for` loop accesses each element of the sequence by index.

Figure 8.2 will help you visualize what is happening in the example program. In the first iteration, the loop variable `number` is assigned 2, it is not negative so it is not removed. In the second iteration, `number` is assigned 3, it is not removed. In the third iteration, `number` is assigned -7, it is negative and so, it is removed from the list and 8 comes in its place. In the fourth iteration, `number` will be assigned -5, because `for` loop is done with items till index 2 and now it will fetch item at index 3. This is why, the element 8 is skipped by the loop. -5 is negative so it is removed from the list and -2 comes at its place. Now `for` loop has treated elements till index 3 so in fifth iteration, it will fetch element at index 4 which is 9. Thus, -2 was skipped by the loop. 9 is positive, so it is not removed. Then in the next iteration `number` is 10.

|                          |    |    |     |     |     |     |    |    |
|--------------------------|----|----|-----|-----|-----|-----|----|----|
| <code>number = 2</code>  | 0  | 1  | 2   | 3   | 4   | 5   | 6  | 7  |
|                          | 2, | 3, | -7, | 8,  | -5, | -2, | 9, | 10 |
| <code>number = 3</code>  | 0  | 1  | 2   | 3   | 4   | 5   | 6  | 7  |
|                          | 2, | 3, | -7, | 8,  | -5, | -2, | 9, | 10 |
| <code>number = -7</code> | 0  | 1  | 2   | 3   | 4   | 5   | 6  |    |
|                          | 2, | 3, | 8,  | -5, | -2, | 9,  | 10 |    |
| <code>number = -5</code> | 0  | 1  | 2   | 3   | 4   | 5   |    |    |
|                          | 2, | 3, | 8,  | -2, | 9,  | 10  |    |    |
| <code>number = 9</code>  | 0  | 1  | 2   | 3   | 4   | 5   |    |    |
|                          | 2, | 3, | 8,  | -2, | 9,  | 10  |    |    |
| <code>number = 10</code> | 0  | 1  | 2   | 3   | 4   | 5   |    |    |
|                          | 2, | 3, | 8,  | -2, | 9,  | 10  |    |    |

**Figure 8.2:** Working of `for` loop

When an item is removed inside the loop, the next item is skipped by the loop because that next item gets the index of the item that has been removed. Before looking at the solution of this problem, let us first see what happens when we add some items in a list while iterating:

```
cities = ['Rome', 'Berlin', 'Delhi', 'Bareilly']
for city in cities:
 if city.startswith('B'):
 cities.append(city)
print(cities)
```

We have a list of cities, and inside the `for` loop, we are iterating over the list. If a city's name begins with 'B', then we are appending that name to the cities list. So basically, we are expecting this output.

Expected output: ['Berlin', 'Rome', 'Bareilly', 'Delhi', 'Berlin', 'Bareilly']

After the loop finishes, the two cities that start with 'B' should be added to the list. But when we run the program, we are stuck in an infinite loop. The following figure will help you comprehend the reason for this infinite process:

```

city = 'Rome' 0 1 2 3
city = 'Berlin' 0 1 2 3 4
city = 'Delhi' 0 1 2 3 4
city = 'Bareilly' 0 1 2 3 4 5
city = 'Berlin' 0 1 2 3 4 5 6
city = 'Bareilly' 0 1 2 3 4 5 6 7
city = 'Berlin' 0 1 2 3 4 5 6 7 8

```

*Figure 8.3: Infinite for loop*

So, the `for` loop will not work correctly if the list is mutated inside the loop. These types of problems can be removed by iterating over a copy of the list instead of iterating over the list itself. We can use the slice notation to get a copy. In the following loops, we have changed the list to a copy of the list, and we get the expected output.

```

cities = ['Rome', 'Berlin', 'Delhi', 'Bareilly']
for city in cities[:]:
 if city.startswith('B'):
 cities.append(city)
print(cities)

```

**Output-**

```
['Rome', 'Berlin', 'Delhi', 'Bareilly', 'Berlin',
'Bareilly']
numbers = [2, 3, -7, 8, -5, -2, 9, 10]
for number in numbers[:]:
 if number < 0:
 numbers.remove(number)
print(numbers)
```

### **Output-**

```
[2, 3, 8, 9, 10]
students = ['Era', 'Ted', 'Rob', 'Joe', 'Amy',
'Sam', 'Pat', 'Joy', 'Tia']
failed_students = ['Ted', 'Amy', 'Sam']
for student in students[:]:
 if student in failed_students:
 students.remove(student)
print(students)
```

### **Output-**

```
['Era', 'Rob', 'Joe', 'Pat', 'Joy', 'Tia']
```

Another approach could be to make a new list by filtering the elements and then renaming the new list to the original list. This can be easily done using comprehensions that we will see in the next chapter. However, this approach does not change the original list in-place, it creates a new object. If there are multiple references to the original list, they will not be updated.

This problem that we saw, occurs with lists only. Strings and tuples are immutable, so there is no chance of adding or removing elements. For a dictionary or a set, if you try to add or remove elements while iterating, you will get a runtime error.

```
employees = {'Sam': 3000, 'John': 4000, 'Rob':
15000, 'Tina': 9000}
for employee, salary in employees.items():
 if salary > 10000:
```



```
 employees.pop(employee)
print(employees)
```

### ***Output-***

RuntimeError: dictionary changed size during iteration

```
pronouns = {'me', 'they', 'everybody', 'those',
 'he', 'myself', 'it'}
for word in pronouns:
 if len(word) > 4:
 pronouns.remove(word)
print(pronouns)
```

### ***Output-***

RuntimeError: Set changed size during iteration

We can iterate over a copy of the dictionary or the set to get the results that we want.

```
employees = {'Sam': 3000, 'John': 4000, 'Rob':
 15000, 'Tina': 9000}
for employee, salary in employees.copy().items():
 if salary > 10000:
 employees.pop(employee)
print(employees)
```

### ***Output-***

```
{'Sam': 3000, 'John': 4000, 'Tina': 9000}
pronouns = {'me', 'they', 'everybody', 'those',
 'he', 'myself', 'it'}
for word in pronouns.copy():
 if len(word) > 4:
 pronouns.remove(word)
print(pronouns)
```

### ***Output-***

```
{'me', 'it', 'they', 'he'}
```

## 8.10 Infinite loop with break

If the loop condition in a while loop never becomes False then the loop will keep on executing infinitely. Such a loop will never end and is called an infinite loop. We have seen in the previous chapter that these never-ending loops occur due to some programming mistake. If a program is stuck in an infinite loop, we have to interrupt the execution of the program and terminate it. Sometimes, you can write infinite loops on purpose and use them to your advantage. You can create an intentional infinite loop by writing True as the loop condition and by placing a conditional break inside the loop body to terminate the loop.

The structure of the while loop provided by the language is such that the loop condition is checked at the top of the loop. The whole loop body is executed, then the control goes to the top of the loop to decide whether the loop should continue or terminate. Sometimes, we want this decision to be made in the middle of the loop body or at the end of the loop body. Writing an infinite loop with a conditional break is a common trick to implement while loops where you need to implement loop condition in the middle or at the bottom instead of the top. Let us understand this with the help of examples.

In the following program, we ask the user to enter some names and append all those names to a list. When the user enters 'exit', this process stops.

```
names = []
name = ''
while name != 'exit':
 name = input('Enter name : ')
 names.append(name)
print(names)
```

The word 'exit', which the user typed to signify the end, is entered inside the list, but we do not want that. To avoid this, we can check the name and append it to the list only if it is not equal to the string 'exit'.

```
names = []
```

```

name = ''
while name != 'exit':
 name = input('Enter name : ')
 if name != 'exit':
 names.append(name)
print(names)

```

But now, there are two similar comparisons being done in each iteration, which is not efficient. We can avoid this by writing an infinite loop with a `break` inside it.

```

names = []
while True:
 name = input('Enter name : ')
 if name == 'exit':
 break
 names.append(name)
print(names)

```

In this `while` loop, in the place of the test expression, we have written `True`, so the test expression will never become `False`, and hence, the loop is an infinite loop. To come out of this infinite loop we have written a `break` statement. When the user enters 'exit', the `break` statement will execute, and the loop will terminate. Now we have only one comparison, and there is no need to initialize the `name` variable with an empty string before the loop. We are making the decision of continuing or ending the loop in the middle of the loop. In this situation, an infinite loop with a `break` in the middle provides a better solution than the regular `while` loop.

Note that the condition we write in the header of the `while` loop is the loop continuing condition (keep executing while this is `True`), while the condition we write before the `break` statement is the exit condition (stop when this is `True`); hence, they are opposite.

So, in situations when you want to make the loop exiting decision somewhere inside the loop body, then in that case you can write an infinite loop with a conditional `break`. Here is another example. In the following

code, we are calculating the total of numbers entered by the user and the user has to enter -1 to stop entering numbers.

```
total = 0
number = 0
while number != -1:
 number = int(input('Enter a number(-1 to quit)
: '))
 total += number
print(total)
```

This loop will give incorrect output. For example, if the user enters 2, 3 and 4 as input then the total will be 8. It is because the value -1 which was entered to end the loop was also added to the total. The string 'exit' in the previous program and -1 in this program are sentinel values. They are used to signify the end of data, but are not a part of the data. When we use a sentinel to end the input, the combination of `while True` and `break` will help us avoid any errors. We can rewrite the above code using an infinite loop like this.

```
total = 0
while True:
 number = int(input('Enter a number(-1 to quit)
: '))
 if number == -1:
 break
 total += number
print(total)
```

Now the value of `total` will be calculated correctly. The problem that we saw is also known as a loop and half problem. In the last iteration, we want only half of the loop to be executed. In a regular while loop which has loop condition in the header, the full loop body will always be executed. So, in these cases, an infinite loop with a `break` is the solution.

Let us see how we can use an infinite loop to enhance the capability of this program that we had seen in [Chapter 6](#).

```
print('1. Add the two numbers')
print('2. Subtract first from second')
print('3. Subtract second from first')
print('4. Multiply the two numbers')
choice = int(input('Enter your choice : '))
if choice == 1:
 print(x + y)
elif choice == 2:
 print(y - x)
elif choice == 3:
 print(x - y)
elif choice == 4:
 print(x * y)
else:
 print('Wrong choice')
```

When we run this program, we can perform just one operation. If we want to perform another operation, we have to run the program again. To execute the operations repeatedly, we can put the whole code inside an infinite while loop.

```
while True:
 print('1. Add two numbers')
 print('2. Subtract first number from second number')
 print('3. Subtract second number from first number')
 print('4. Multiply two numbers')
 print('5. Exit')
 choice = int(input('Enter your choice : '))
 if choice == 5:
 break
 if choice < 1 or choice > 4:
```

```

 print('Wrong choice')
 continue
x = int(input('Enter first number : '))
y = int(input('Enter second number : '))
if choice == 1:
 print(x + y)
elif choice == 2:
 print(y - x)
elif choice == 3:
 print(x - y)
elif choice == 4:
 print(x * y)

```

If the user chooses option 5, then the `break` statement will be executed and the loop will terminate. This is a very common way of writing menu driven programs. First a menu is displayed, then the user is asked to choose a particular option, and then depending on the option entered, specific action is taken by using `if-else` statement. This whole thing is enclosed inside a loop so that it executes repeatedly.

An infinite loop with exit condition at the bottom is like the `do..while` loop available in some other languages. Python does not have a `do..while` loop like C has, it only has a `while` loop and a `for` loop. The `do..while` loop available in other languages is a special loop that has the loop condition at the end of the loop body. It is guaranteed to execute at least once since there is no check at the entry of the loop. In Python, we can simulate a `do..while` loop by placing a `break` at the end of an infinite `while` loop.

```

while True:

 if test:
 break

```

This loop's body will run at least once. The test is for exiting the loop, so it will be the opposite of the expression that would have been written in the

`while` loop header. These types of loops can be used for input checking. Let us see an example for this.

We have seen the following piece of code in the previous chapter. We are asking the user to enter a student id, and the valid student ids are from 1000 to 9999. If the user enters any other number, then we ask for the id again.

```
student_id = int(input('Enter student id : '))
while student_id < 1000 or student_id > 9999:
 student_id = int(input('Enter student id : '))
```

We are doing the input validation by using the regular `while` loop and so we have to write the input function twice. If we write an infinite loop with a `break`, we will have to write it only once.

```
while True:
 student_id = int(input('Enter student id :'))
 if student_id >= 1000 and student_id <= 9999:
 break
```

In the previous loop the `while` condition was for continuing the loop and here the `if` condition is for terminating the loop, so the conditions are opposite. This loop will keep on executing till the user enters a valid id. If we want, we can print a message if an invalid id is entered.

```
while True:
 student_id = int(input('Enter student id :'))
 if student_id >= 1000 and student_id <= 9999:
 break
 else:
 print('Invalid id : id should be a number
between 1000 and 9999')
```

The `else` is actually not required here, because the control will come at the `print` only if the `break` is not executed.

```
while True:
 student_id = int(input('Enter student id :'))
 if 1000 < student_id < 9999:
```

```
 break
 print('Invalid id : id should be a number
between 1000 and 9999')
```

Here is another program that we saw in the previous chapter.

```
fruit_prices = {'apple': 210, 'banana': 100,
'grapes': 90}
done = False
while not done:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 done = True
print(fruit_prices)
```

We can rewrite it by using an infinite loop with a break. There will be no need of the variable done.

```
fruit_prices = {'apple': 210, 'banana': 100,
'grapes': 90}
while True:
 fruit = input('Enter fruit name : ')
 price = int(input('Enter price : '))
 fruit_prices[fruit] = price
 if input('Want to enter more(y/n) : ') == 'n':
 break
print(fruit_prices)
```

## 8.11 Avoiding complex logical conditions using break

When the test expression in the while loop becomes too complex to understand, we can use the break statement to simplify the code. Here is a



dummy while loop with a complex test expression.

```
while i < 10 and j > 5 and x + y < 100 and z ==
True:
```

```
.....
.....
```

To simplify the test expression, we can bring down some of the conditions.

```
while i < 10 and j > 5:
 if x + y >= 100 or z == False:
 break
```

```
.....
.....
```

Notice the conditions have reversed; logical ‘and’ has changed to logical ‘or’. You could use an intentional infinite loop and bring down all the conditions inside the loop body if you feel that the loop exit conditions are simpler to understand and write than the loop continuing condition.

## Exercise

What will be the output for questions 1 to 17?

```
1. L = [10, 20, 30]
 for data in enumerate(L, 5):
 print(data[0], data[1], end=' ')

2. L = [1, 2, 3, 4, 5, 6]
 for i in range(0, len(L)-1, 2):
 L[i], L[i+1] = L[i+1], L[i]
 print(L)

3. listA = [1, 3, 4, 8, 5, 6, 7]
 list_even = []
 for i in range(len(listA)):
```

```

 if listA[i] % 2 == 0:
 list_even.append(listA.pop(i))
list_odd = listA
print(list_even, list_odd)
4.L = [10, 20, 30, 40, 50, 60, 70]
 for count, item in enumerate(L):
 if count == 5:
 break
 print(item, end=' ')
5.for n in reversed(range(5, 15, 3)):
 print(n, end=' ')
6.for x in enumerate([2,3,4], 2):
 print(x, end=' ')
7.L = ['yes', 'no', 'this']
 for word in L:
 word = word.capitalize()
 print(L)
8.L = [10, 11, 12, 13]
 for item in L:
 item += 1
 print(L)
9.cities = ['London', 'Paris', 'Noida',
 'Perth', 'Rome']
 for city in cities:
 if city == 'Paris':
 cities.append('New York')

```

```

 if city == 'New York':
 cities.append('New Delhi')
 print(cities)
10. cities = ['Paris', 'Noida', 'Perth',
 'Rome', 'London']
 for city in cities:
 if len(city) < 5:
 cities.append(city)
 print(cities)
11. L = [1, 2, 3]
 for i in L[:]:
 L.append(i)
 print(L)
12. L = [3, 1, 2, 6, 8, 5]
 for item in reversed(sorted(L)):
 print(item, end=' ')
13. L1 = [10, 23, 34, 90]
 L2 = [2, 4, 1, 4]
 for x, y in zip(L1, L2):
 x += y
 print(L1)
14. L1 = [10, 23, 34, 90]
 L2 = [2, 4, 1, 4]
 for i in range(len(L1)):
 L1[i] += L2[i]
 print(L1)

```

- ```
15.names = ['Sam', 'Tom', 'Bob', 'Rob']
    ages = [23, 32, 25, 30]
    cities = ['Paris', 'London', 'Tokyo',
              'Paris']
    for data in zip(names, ages, cities):
        name, age, city = data
        if age > 25:
            print(name, city, end=' ')

16.L = [1, 2, 3, -5, 6, 7]
    for i in L:
        if i < 0:
            L.insert(L.index(i), 0)
    print(L)

17.students = ['Pam', 'Sam', 'John', 'Ryan',
               'Neil', 'Dev']
    for i, student in enumerate(students):
        if i % 2 == 0:
            print(student, end=' ')

18. Write a loop to iterate over the keys of this dictionary in reverse
    sorted order.

    D = {'apple': 210, 'banana': 100, 'grapes':
    90, 'mango': 250, 'cherry':225, 'guava': 80}

19. Write a for loop to capitalize each string of this list. Use
    enumerate() function.

    L = ['this', 'that', 'the', 'hello world']

20. What is the following loop trying to do? Is there a better way of doing
    this task?
```

```
data = [2, 1, 3, 5, 7]
for i in range(len(data)-1, -1, -1):
    print(data[i])
```

21. The loop in the following code iterates over 2 collections. Rewrite the loop using a more Pythonic approach.

```
names = ['Ted', 'Sam', 'Rob']
cities = ['NY', 'GT', 'UU', 'KK']
n = min(len(names), len(cities))
for i in range(n):
    print(f'{names[i]} will be posted in
{cities[i]}')
```

22. Rewrite the following loop using a more Pythonic approach.

```
for i in [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16]:
    print(i, end=' ')
```

23. Given a list of integers, write a `for` loop that multiplies each odd number of the list by 2 and divides each even number by 2. Use `if else` operator inside the loop.

24. Write a `for` loop to print the elements of the following list in sorted order without duplicates.

```
L = [2, 4, 1, 6, 7, 8, 9, 7, 1, 2, 6]
```

25. Write a program to create the following dictionary in which keys are numbers from 1 to 7, and corresponding values are their factorials. Do not use nested loops.

```
{0: 1, 1: 1, 2: 2, 3: 6, 4: 24, 5: 120, 6:
720, 7: 5040}
```

26. Write a program to remove `nth` occurrence of an item from a list.

27. Print the names of unique cities from the following dictionary. The city names should be in all capitals.

```
D = {'Sam': 'London', 'Tom': 'Belmont', 'Bob':  
    'Belmont', 'Dev': 'Bareilly', 'Tim':  
    'Belmont', 'Raj': 'London'}
```

28. In the previous chapter, we had written a program to encrypt a message by replacing each letter by subsequent letter.

Modify that program so that a character at even index is replaced by subsequent character while a character at odd index is replaced by previous character.

29. Encrypt the strings of this list by changing each letter of the string to the next letter.

```
L = ['this', 'that', 'here', 'there']
```

30. Write a program to find whether a list contains any duplicate value.

31. The following `for` loop is written to delete all occurrences of an item from a list. Will it work properly? If not, what changes need to be made in this code.

```
L = [1, 2, 3, 2, 4, 5, 2, 2, 2, 7]  
x = 2  
for item in L:  
    if item == x:  
        L.remove(item)  
  
print(L)
```

32. What will be the output of the following program?

```
L = [1, 2, -3, 4, -5, -6, 8]  
i = 0  
while i < len(L):  
    print(i, L[i], end = ' | ')  
    if L[i] < 0:  
        L.remove(L[i])
```

```
    i += 1  
print(L)
```

33. In the following `for` loop, we are iterating over the items of a list and finding the largest even number. Rewrite the code so that you get the largest even number as well as its index.

```
data = [2, 3, 1, 4, 7, 5]  
max_even = 0  
for item in data:  
    if item % 2 == 0 and item > max_even:  
        max_even = item  
if max_even != 0:  
    print(f'Largest even number is  
{max_even}')else:  
    print(f'No even number in the list')
```

34. In the following code, we are iterating over a list and removing all the negative numbers from it. To avoid any problems, we are iterating over a copy.

```
L = [1, 2, -3, 4, -5, -6, -8]  
for item in L[:]:  
    if item < 0:  
        L.remove(item)  
print(L)
```

Another approach to do this could be to create a new list that contains all non-negative numbers of the original list and then rename the original list to the new list.

```
L = [1, 2, -3, 4, -5, -6, -8]  
L1 = []
```

```

for item in L:
    if item >= 0:
        L1.append(item)

L = L1
print(L)

```

What kind of problems can this solution create?

35. Here is some data in the form of a dictionary and three lists. The dictionary contains names of students as keys and lists of marks in 4 subjects as values. The three lists named `subjects`, `max_marks` and `pass_marks` contain the subject names, maximum marks in subjects and pass marks in subjects at the corresponding indices.

```

D = { 'John':  [90,78,87,67] ,
      'Sam'  :  [95,76,78,57] ,
      'Dev'   :  [80,69,59,45]
    }

subjects = ['Physics', 'Chemistry', 'Maths',
            'Biology']

max_marks = [100, 80, 100, 75]

pass_marks = [40, 25, 40, 20]

```

Write a program that displays the following output from the above data. Use `zip` in for loop to iterate over the lists.

John

```

-----
Physics      100   40   90
Chemistry    80    25   78
Maths        100   40   87
Biology      75    20   67

```


Total = 322

Percentage = 90.70

Sam

Physics	100	40	95

Chemistry	80	25	76

Maths	100	40	78

Biology	75	20	57

Total = 306

Percentage = 86.20

Dev

Physics	100	40	80

Chemistry	80	25	69

Maths	100	40	59

Biology	75	20	45

Total = 253

Percentage = 71.27

36. The following two code snippets are trying to remove all those elements from list `names1` that are also present in list `names2`. Which one of them will work correctly?

```
names1 = ['Sam', 'Rob', 'Fed', 'Tim']    names1
= ['Sam', 'Rob', 'Fed', 'Tim']
```

```

names2 = ['John', 'Kim', 'Rob', 'Fed', 'Jim']
names2 = ['John', 'Kim', 'Rob', 'Fed', 'Jim']
for name in names1:      temp = names1
    if name in names2:   for name in temp:
        names1.remove(name)      if name in
names2:
print(names1)            names1.remove(name)
    print(names1)

```

37. Rewrite this number-guessing program using an infinite loop and break.

```

from random import randint
secret = randint(1, 100)
print('The secret number is in between 0 and
100')
n = int(input('Enter a number : '))
attempts = 1
while n != secret and attempts != 10:
    if n > secret:
        print('Bigger than the secret number')
        n = int(input('Enter a number : '))
    elif n < secret:
        print('Smaller than the secret
number')
        n = int(input('Enter a number : '))
    attempts += 1
if n == secret:
    print('You guessed it right')

```

```
else:  
    print('No more attempts')  
    print(f'Secret number is {secret}')
```

Comprehensions

9

Creating a list, dictionary, or set by processing another iterable is a common requirement in our programs. Here are a few examples to illustrate this.

- From the list `[3, 5, 6, 7, 9]`, create another list that contains the squares of all the numbers in the list. The resulting list will be `[9, 25, 36, 49, 81]`.
- From the tuple `(4, -3, 9, -2, 6)`, create a list that contains cubes of only the positive numbers. The resultant list will be `[64, 729, 216]`
- From the dictionary `{1: 'a', 3: 'c', 5: 'c', 9: 'd'}`, create a set of all the values. The resultant set will be `{'a', 'c', 'd'}`
- From the list `[3, 5, 2, 8]` create a dictionary in which keys are numbers from the list and values are the square of keys. The resultant dictionary will be `{3: 9, 5: 25, 2: 4, 8: 64}`.

In all these examples, we are filtering and transforming the data from an iterable and creating a new list, set, or dictionary. We know that this can be done using the `for` loop and `if` statement. This is a common pattern, so Python provides a shorthand syntactic construct called comprehension, which is a more convenient and compact way of performing these types of tasks. There are three types of comprehensions:

- List comprehensions
- Dictionary comprehensions

- Set comprehensions

List comprehension is an expression that creates a new list object; a dictionary comprehension expression creates a new dictionary object, and a set comprehension creates a new set object. The type of the existing iterable need not be the same as the type of the new data structure produced by the comprehension. For instance, you can create a new list from an existing dictionary or a new set from a list.

Comprehensions are just syntactic sugar for the `for` loop syntax, but they are considered more Pythonic. Code written using comprehensions is shorter, more readable, and often more efficient than the equivalent code written using a `for` loop.

9.1 List Comprehensions

A common pattern to create a new list is to iterate over an iterable using a `for` loop and append original or transformed items from the iterable to the new list. For example, suppose we want to create a list that contains cubes of all integers from 5 to 10. This is how we will do it using a `for` loop.

```
cubes = []
for n in range(5, 11):
    cubes.append(n ** 3)
```

First, we create an empty list, then in the `for` loop, we iterate over the `range` iterable and keep on appending cubes to the list. The final list that we get is `[125, 216, 343, 512, 729, 1000]`. We can do the same work in a single line using list comprehension.

```
cubes = [n ** 3 for n in range(5, 11)]
```

List comprehension lets us do this concisely; three lines of code were reduced to one. We can see that the syntax is less verbose than the `for` loop syntax. This is a more Pythonic way of making a new list, and it may run faster than the equivalent `for` loop syntax.

To understand the list comprehension, look at it from right to left. It loops through each element in the iterable `range(5, 11)` and temporarily assigns the value of each element to the variable `n`. The expression `n **`

3 is evaluated each time and is automatically appended to the output list. You can think of it as a backward `for` loop.

Here is the syntax of a list comprehension expression:

```
[expression for item in iterable]
```

This comprehension expression creates and returns a new list object. The `for` syntax provides the values, and the expression that we have on the left is the one that is appended to the list. The comprehension expression constructs a new list object in memory, which can be assigned to a variable, as we have done in our example. You can easily create a new list object from any type of iterable object like a list, tuple, string, dictionary, set, range object, file, or anything that can be iterated over in a `for` loop. Let us see some more examples to get used to this syntax:

We have a list `L`, and we want to make a list that contains cubes of numbers from this list.

```
>>> L = [3, 5, 7, 1, 8, 9, 4]
>>> cubes = [n ** 3 for n in L]
>>> cubes
[27, 125, 343, 1, 512, 729, 64]
```

In the expression `[n ** 3 for n in L]` we are iterating over the list `L`. Instead of fetching numbers from the `range` function as we did in the last example, here we are fetching numbers from the list `L`. The resultant list contains cubes of numbers in the list `L`.

The following list comprehension produces a list that contains the double of numbers in list `L`.

```
>>> L2 = [n * 2 for n in L]
>>> L2
[6, 10, 14, 2, 16, 18, 8]
```

Since we want the double of each number, we have written the expression as `n * 2`. So, we can see that the expression defines how to transform each element of the iterable, before appending the resultant value to the list.

Our next comprehension creates a list in which all the numbers in list `L`, are in string form.

```
>>> [str(n) for n in L]
['3', '5', '7', '1', '8', '9', '4']
```

If we want a list of floats, we can use the `float` function instead.

While writing the list comprehensions, it is more intuitive to first write the `for` part and then the expression on the left side. Similarly, while reading comprehensions, it makes more sense to first read the `for` part and then the expression.

Let us see a few more examples. We have the following list of strings named `cities`, and we intend to make a list that contains the initial 3 characters of each string in this list.

```
>>> cities = ['Belmont', 'New york', 'Paris',
'Buenos aires']
>>> [city[:3] for city in cities]
['Bel', 'New', 'Par', 'Bue']
```

We iterated over the list and used the slice notation to get the first 3 characters of each string. If we want to create a list that contains all these strings in title case, we can write our comprehension as shown below:

```
>>> [city.title() for city in cities]
['Belmont', 'New York', 'Paris', 'Buenos Aires']
```

Next, we want to create a list that contains two-element tuples; the first element should be a string from the `cities` list, and the second element should be the length of that string.

```
>>> [(city, len(city)) for city in cities]
[('Belmont', 7), ('New york', 8), ('Paris', 5),
('Buenos aires', 12)]
```

Here the expression is a tuple. If we want this to be a list of lists, we can specify square brackets.

```
>>> [[city, len(city)] for city in cities]
```

```
[['Belmont', 7], ['New york', 8], ['Paris', 5],  
['Buenos aires', 12]]
```

Now, we have created a list that contains lists. This way we can create nested lists by using comprehensions.

In the following example, we have a list of lists. We want to create a new list that contains the sum of the inner sublists.

```
>>> L = [[1, 2, 11, 13], [12, 34, 56, 10], [13,  
77, 89], [56, 78]]  
>>> [sum(sublist) for sublist in L]  
[27, 112, 179, 134]
```

We get a list in which each element is the sum of inner sublists of the list L. Let us use the max function instead of sum.

```
>>> [max(sublist) for sublist in L]  
[13, 56, 89, 78]
```

Now, we get a list in which each element is the largest element of each inner sublist of list L.

In our next example, we have a list named `heights` that contains the heights in inches, and from this list, we want to create a new list that contains heights in cms.

```
>>> heights = [12, 45, 78, 77, 12, 14, 54]  
>>> heights_cm = [ht * 2.54 for ht in heights]  
>>> heights_cm  
[30.48, 114.3, 198.12, 195.58, 30.48, 35.56,  
137.16]
```

Next, we have a list named `weights` that contains weights in grams.

```
>>> weights = [2900, 3450, 6678, 2348, 800, 8999,  
90]
```

From this list, we want to create a new list that contains two-element tuples, where the first element is the number of kilograms and the second element is the number of grams in the weight. The first weight in the list is 2900 gms, which is 2 kilograms and 900 gms, so the tuple corresponding to it

would be (2, 900). Similarly, the tuple corresponding to the next one would be (3,450). To get kilograms from the weight, we can divide the list element by 1000 (integer division), and to get the grams, we can use the remainder operator. Here is the comprehension:

```
>>> wts = [(wt // 1000, wt % 1000) for wt in weights]
>>> wts
[(2, 900), (3, 450), (6, 678), (2, 348), (0, 800), (8, 999), (0, 90)]
```

We get this list of tuples where each element contains the weight in kilograms and grams. Now, suppose from this list of tuples, we want to create a new list that contains weights in grams.

```
>>> [t[0] * 1000 + t[1] for t in wts]
[2900, 3450, 6678, 2348, 800, 8999, 90]
```

We iterated over the list `wts`, each element of this list is a tuple, so the identifier `t` is a tuple here. The first element of the tuple is `t[0]`, it is the weight in kilograms, so we multiply it by 1000, and the second element is the weight in grams, so we add it. We get the list where each element represents the weight in grams. Instead of doing this, we could just unpack the tuple.

```
>>> [kg * 1000 + gm for kg, gm in wts]
[2900, 3450, 6678, 2348, 800, 8999, 90]
```

If we need to iterate over multiple lists, we can use the `zip` function. In the next example, we have three lists, and we will write a comprehension that contains the sum of the corresponding elements of these lists.

```
>>> L1 = [1, 2, 3, 4, 5]
>>> L2 = [4, 6, 7, 1, 8]
>>> L3 = [7, 5, 3, 1, 2]
>>> L = [x + y + z for x, y, z in zip(L1, L2, L3)]
>>> L
[12, 13, 13, 6, 15]
```

We get a list in which each element is the sum of corresponding elements of these lists.

Comprehensions have their own local scope. Any variables assigned in comprehensions are local to that comprehension expression and are not available outside the comprehension. So, they do not clash with similar names outside the comprehension. The scope of variables will be discussed in a later chapter.

9.2 if clause in list comprehension

To filter out unwanted values, we can append an `if` clause at the end of the list comprehension.

```
[expression for item in iterable if condition]
```

The condition after the `if` keyword will be evaluated for each item in the iterable. If the condition evaluates to `True`, only then the expression will be included in the output list.

The iterable in the `for` clause provides the values, the `if` clause does the filtration, and the expression transforms the selected values, which are then appended to the new list. Let us see a few examples, now.

We created a list of cubes of numbers from another list using list comprehension. Now, we will create a new list that contains cubes of only even numbers from the input list. For this, we have to add an `if` clause in the comprehension.

```
>>> L = [3, 5, 7, 1, 8, 9, 4]
>>> cubes = [n ** 3 for n in L if n % 2 == 0]
>>> cubes
[512, 64]
```

This is the code that we would have written, had we created the same list using a `for` loop.

```
cubes = []
for n in L:
    if n % 2 == 0:
```

```
cubes.append(n ** 3)
```

By using a list comprehension, we can filter the data and also transform the data into a single statement. Filtration is done by using the `if` clause, and transformation is done through the expression. When the `if` clause is present, we generally get a list smaller than the size of the iterable from which we are creating the list.

In our next example, we have a list of numbers, and we have created a list that contains doubles of only the positive numbers.

```
>>> L = [32, -51, 63, 11, 86, -9, 66, 88, 97]
>>> [n * 2 for n in L if n > 0]
[64, 126, 22, 172, 132, 176, 194]
```

Now from the list `L`, we need to make two lists, one that contains even numbers from `L` and the other that contains odd numbers from `L`. Here are the comprehensions for the two lists:

```
>>> evens = [n for n in L if n % 2 == 0]
[32, 86, 66, 88]
>>> odds = [n for n in L if n % 2 != 0]
[-51, 63, 11, -9, 97]
```

In our next example, we have a list of words and have made another list that contains only those words from this list that are palindromes.

```
>>> words = ['apple', 'civic', 'board', 'noon',
'moon', 'lamp', 'madam']
>>> palindromes = [word for word in words if word
== word[::-1]]
>>> palindromes
['civic', 'noon', 'madam']
```

In the next example, we have a list of two element tuples, where the first element of the tuple is a person's name and the second element is the body mass index of that person. From this list, we want to create a list of names of those people whose BMI is in the range of 20 to 26.

```
>>> L = [('Ted', 23), ('Lee', 18), ('Sam', 22),
('Bob', 30), ('Dev', 27), ('Ray', 25)]
```

```
>>> [name for name, bmi in L if 20 < bmi < 26]
['Ted', 'Sam', 'Ray']
```

We know that we can see a list of all methods related to a type by using the `dir` function, for example we can write `dir(str)` to see all methods related to `str` type. List comprehensions can be written to selectively see the methods.

```
>>> [method for method in dir(str) if not
method.startswith('_')]
>>> [method for method in dir(str) if
method.startswith('is')]
```

The first comprehension creates a list of all those methods that do not start with an underscore, and the second one creates a list of all methods that start with 'is'.

9.3 Ternary operator in list comprehension

We have seen the following comprehension that adds only even numbers to the new list and discards the rest of the numbers.

```
>>> L = [1, 2, -3, 6, 18, -9, 12, -5, 19, -8, 5]
>>> [n for n in L if n % 2 == 0]
[2, 6, 18, 12, -8]
```

Suppose we do not want to filter out the odd numbers, instead we want to replace them with another value (suppose 0), then we can use the ternary operator.

```
>>> [n if n % 2 == 0 else 0 for n in L]
[0, 2, 0, 6, 18, 0, 12, 0, 0, -8, 0]
```

Now, all the even numbers are copied as such, while odd numbers are replaced by zero in the new list. Let us see how this works. We have seen the syntax of list comprehension.

```
[ expression for item in iterable if condition]
```

In the place of expression, if we write an expression with a ternary operator, we can replace an item from the iterable with another value. We know that

the ternary operator is of this form-

```
x if condition else y
```

This expression evaluates to `x` if the condition is `True`. Else, it evaluates to `y`.

Now suppose we want to create another list in which all positive numbers of the list are copied as such but negative numbers are replaced by zero.

```
>>> L = [1, 2, -3, 6, 18, -9, 12, -5, 19, -8, 5]
>>> [n if n > 0 else 0 for n in L]
[1, 2, 0, 6, 18, 0, 12, 0, 19, 0, 5]
```

Here, we used the ternary expression `n if n > 0 else 0` in the list comprehension, to get our desired list. Now, let us create another list from the list `L` such that all the even numbers are divided by 2 and odd numbers are multiplied by 2. We want to consider only positive numbers of the list.

```
>>> L = [1, 2, -3, 6, 18, -9, 12, -5, 19, -8, 5]
>>> [n // 2 if n % 2 == 0 else n * 2 for n in L if
n >= 0]
[2, 1, 3, 9, 6, 38, 10]
```

Even numbers of this list are divided by 2, odd ones are multiplied by 2 and the negatives ones have not been considered. We have used the `if` clause to filter out the negative numbers and with the help of ternary expression we are replacing odd and even numbers. The `if` clause of the comprehension is `if n >= 0` and the ternary expression is `n // 2 if n % 2 == 0 else n * 2`.

The `if` clause of the list comprehension cannot have an `else` clause. If we see an `else` in the list comprehension, it means a ternary expression has been used before the `for` keyword.

9.4 Modifying a list while iterating

In the chapter on looping techniques, we saw that if we try to remove items from a list while iterating through it, we will get incorrect results. Here is the example that we saw in [Section 8.9](#).

```
students = ['Era', 'Ted', 'Rob', 'Joe', 'Amy',
            'Sam', 'Pat', 'Joy', 'Tia']
failed_students = ['Ted', 'Amy', 'Sam']
for student in students:
    if student in failed_students:
        students.remove(student)
print(students)
```

Output-

```
['Era', 'Rob', 'Joe', 'Sam', 'Pat', 'Joy', 'Tia']
```

As we had seen, this could be avoided by iterating over a copy of the students list. Another solution could be to create a new list instead of modifying the original list. We can write a list comprehension to create a new list that does not contain the unwanted elements and make the name students refer to the new list.

```
students = [stu for stu in students if stu not in
            failed_students]
```

With this approach, there could be side effects if there are other variables referencing the original list object. In that case you have to modify the original list instead of creating a new one and you can do this by using the slice assignment.

```
students[:] = [stu for stu in students if stu not
               in failed_students]
```

Now we have replaced the contents of the original list object.

9.5 Getting keys from values in a dictionary using list comprehension

Suppose we have a dictionary where the book names are keys and values are author names.

```
d = {'Poems for kids': 'Joe',
     'Stories for kids': 'Zen',
```

```
'Health is wealth': 'James',  
'Rhymes for Babies': 'Joe',  
'Stories for teens': 'Ted',  
'Be healthy': 'James'  
}
```

If we want to know the author of a book, we can specify the name of the book in square brackets; for example, we can write `d['Stories for kids']` to get the author of the book 'Stories for kids'. We can get the value from a key like this, but getting the key from a value is not possible with this syntax. For example, in this dictionary, we do not know how to get the book name if the author's name is given.

There can be duplicate values in a dictionary, so a particular value can be associated with multiple keys. For example, in this dictionary, for an author's name, there can be many book names. By using list comprehension, we can get all the keys that are associated with a given value.

```
>>> [book_name for book_name, author in d.items()  
if author == 'James']  
['Health is wealth', 'Be healthy']
```

This comprehension creates a list of all books written by author James. This way we can use list comprehension for getting keys from values of the dictionary.

In the next example we have a dictionary in which names are used as keys and dates of birth as values.

```
employees = {'Jack': '02-03-1973',  
             'John': '09-12-1977',  
             'Mark': '09-11-1972',  
             'Mary': '08-05-1977',  
            }
```

We need to create a list of all those names who were born in 1977. Here is the list comprehension to create that list.

```
>>> [name for name, dob in employees.items() if
dob[-4:] == '1977']
['John', 'Mary']
```

Date of birth is in the form of a string where the last four characters represent the year. So, we have used slicing to extract the last 4 characters of the string.

In our next example, we have a dictionary where keys are ids of students and values are the student records in the form of dictionaries.

```
students = {'12AB': {'name': 'Joe', 'age': 13,
'grade': 'A'},
            '17CD': {'name': 'Sam', 'age': 14,
'grade': 'A+'},
            '42CR': {'name': 'Ted', 'age': 13,
'grade': 'A+'},
            '13CR': {'name': 'Bob', 'age': 13,
'grade': 'B+'},
            '19FD': {'name': 'Rob', 'age': 12,
'grade': 'A+'}}
```

From the `students` dictionary, we want to create a list of names of all those students who got A+ grade. We need name and grade only, so there is no need to iterate over the `items` method, we will iterate over the `values` method of the `students` dictionary.

```
>>> [record['name'] for record in
students.values() if record['grade'] == 'A+']
['Sam', 'Ted', 'Rob']
```

If we want both ids and names in our list, then we have to iterate over the `items` method of the dictionary.

```
>>> [(id, record['name']) for id, record in
students.items() if record['grade'] == 'A+']
[('17CD', 'Sam'), ('42CR', 'Ted'), ('19FD',
'Rob')]
```


Now we get a list of tuples, where each tuple contains the id and the name of the student who got A+.

9.6 Using list comprehensions to avoid aliasing while creating lists of lists

In the chapter on lists, we had seen that there can be an aliasing problem when we initialize a list of lists by using the repetition operator. This problem can be solved if we use list comprehension. First, let us see once again what the problem was:

```
>>> L1 = [[]] * 3
>>> L2 = [[0] * 3] * 4
>>> L1
[[], [], []]
>>> L2
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

We have created two lists `L1` and `L2` by using the repetition operator. The lists look okay but the problem surfaces as soon as we change any inner sublist. Let us append 9 to first sublist of `L1`.

```
>>> L1[0].append(9)
>>> L1
[[9], [9], [9]]
```

9 is appended to all sublists, this happened because the four elements of the list `L` are actually references to the same list. Now, let us pop an element from the second sublist of `L2`.

```
>>> L2[1].pop()
>>> L2
[[0, 0], [0, 0], [0, 0], [0, 0]]
```

We can see that an element is popped from all sublists. The list `L2` is also made up of four references to the same list. This is why change in one

sublist is reflected in the other. To avoid these aliasing problems, you can write list comprehensions for creating these types of lists.

```
>>> L3 = [[] for i in range(3)]
>>> L3
[[], [], []]
>>> L4 = [[0] * 3 for i in range(4)]
>>> L4
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

We created lists L3 and L4 similar to lists L1 and L2 but by using list comprehension. Here in each iteration, a new empty list object will be built and appended to the list. Let us try the similar append and pop operations on the sublists of these lists L3 and L4.

```
>>> L3[0].append(9)
>>> L3
[[9], [], []]
>>> L4[1].pop()
>>> L4
[[0, 0, 0], [0, 0], [0, 0, 0], [0, 0, 0]]
```

9 is appended to only the first sublist of L3 because all the sublists are separate objects now. Similarly, an element is popped only from the first sublist of L4. So, when we need to initialize a list with some nested lists, we can use list comprehension to avoid any aliasing problem.

9.7 Multiple for clauses and Nested list Comprehensions

We can have more than one for clause in the list comprehension. Each for clause can have its own optional if clause.

```
[expression for item1 in iterable1 if condition1
      for item2 in iterable2 if condition2
      .....]
```

```
for itemN in iterableN if conditionN ]
```

The `for` clauses work like nested `for` loops. Although we can have any number of `for` clauses, more than two `for` clauses will make the comprehension complex to read. Let us see some examples:

```
s1 = 'Abc'
s2 = 'XYZ'
L = []
for ch1 in s1:
    for ch2 in s2:
        L.append(ch1 + ch2)
print(L)
```

We have two strings `s1` and `s2`, and by iterating over these strings in the two nested `for` loops we have created the list `L`. We know how these nested loops work, the inner loop executes fully for each iteration of the outer loop. Now, let us do the same thing using list comprehension:

```
>>> [ch1 + ch2 for ch1 in s1 for ch2 in s2]
['AX', 'AY', 'Az', 'bX', 'bY', 'bz', 'cX', 'cY', 'cz']
```

We get the same list. Now suppose we want to add `ch1` only if it is in lowercase and `ch2` only if it is in uppercase. To achieve this, we can write `if` clause for both the `for` clauses.

```
>>> [ch1 + ch2 for ch1 in s1 if ch1.islower() for
ch2 in s2 if ch2.isupper()]
['bX', 'bY', 'cX', 'cY']
```

Here is the code if we do the same thing in the nested loop structure:

```
L3 = []
for ch1 in s1:
    if ch1.islower():
        for ch2 in s2:
            if ch2.isupper():
                L3.append(ch1 + ch2)
```

```
print(L3)
```

We had seen that we can represent a 2D matrix using a list of lists.

```
matrix = [ [1, 4, 8, 3],  
            [2, 5, 6, 3],  
            [7, 9, 5, 8],  
          ]
```

We have a list of lists that represents a matrix of size 3 by 4. Now we want a new matrix that is of the same size as this matrix, and each element of that matrix is double of the corresponding element of this matrix. Suppose we write the following comprehension to achieve this:

```
>>> [element * 2 for row in matrix for element in  
row]  
[2, 8, 16, 6, 4, 10, 12, 6, 14, 18, 10, 16]
```

We get a list of doubled elements, but we wanted a list of lists so we need to make some changes in our comprehension. Since we need lists inside our new list, the expression that we write in the comprehension should give us a list. The initial expression that we specify in the comprehension can be any valid Python expression, list comprehension is also an expression, so we can have a list comprehension at the place of expression.

```
>>> [[element * 2 for element in row] for row in  
matrix]  
[[2, 8, 16, 6], [4, 10, 12, 6], [14, 18, 10, 16]]
```

In this list comprehension, we have only one `for` clause, and in the place of expression, we again have a list comprehension. So, we have written a nested list comprehension, which means a list comprehension inside another list comprehension, and the result that it gives us is a list of lists.

9.8 Extracting a column in a matrix

The following structure stores the matrix by rows, so if we want to extract a row, we can easily extract it. For example, to get the third row, we can write `matrix[2]`

```
>>> matrix = [[1, 4, 8, 3],
               [2, 5, 6, 3],
               [7, 9, 5, 8],
               ]
```

```
>>> matrix[2]
[7, 9, 5, 8]
```

To extract a column, we need to write a list comprehension. Suppose we want to extract the second column.

```
>>> [row[1] for row in matrix]
[4, 5, 9]
```

Now we get a list of all elements in second column. If we change this to 2, we get the elements in third column.

```
>>> [row[2] for row in matrix]
[8, 6, 5]
```

9.9 Dictionary Comprehensions

A dictionary comprehension constructs and returns a dictionary. It generates key value pairs from one or more iterable. Like list comprehension, filtering and transforming is possible. Here is the syntax of a dictionary comprehension:

```
{key_expression : value_expression for item in
iterable if condition}
```

This syntax is similar to that of list comprehension, with basically two differences. First, the whole thing is enclosed in curly braces instead of square brackets. The second difference is that here we have two expressions separated by a colon instead of a single expression. This first expression denotes the dictionary key, and the second expression denotes the corresponding value. The `for` clause and the `if` clause work the same way as in list comprehension. Here also we can have multiple `for` clauses and `if` clauses. Let us see some examples now.

Suppose we have a list of integers and we want to create a dictionary in which keys are numbers from the list and values are the square of keys.

```
>>> L = [2, 6, -4, 8, 3, 9, -5, -3]
>>> {n: n ** 2 for n in L}
{2: 4, 6: 36, -4: 16, 8: 64, 3: 9, 9: 81, -5: 25, -3: 9}
```

If we want only positive numbers to be included as keys in the dictionary, then we can add an `if` clause in this dictionary comprehension.

```
>>> {n: n ** 2 for n in L if n > 0}
{2: 4, 6: 36, 8: 64, 3: 9, 9: 81}
```

Now we have only positive numbers as the keys.

In our next example we have a dictionary where key is the id of a student and value is another dictionary representing the student record.

```
students = {'12AB': {'name': 'Raj', 'class': 5,
                    'marks': 400},
            '14XD': {'name': 'Dev', 'class': 6,
                    'marks': 350},
            '12YR': {'name': 'Rob', 'class': 4,
                    'marks': 289},
            '13CP': {'name': 'Zen', 'class': 5,
                    'marks': 315},
            '23CX': {'name': 'Ted', 'class': 5,
                    'marks': 270},
            '15XG': {'name': 'Sam', 'class': 3,
                    'marks': 390},
            '19HY': {'name': 'Pam', 'class': 5,
                    'marks': 250},
            }
```

From this dictionary, we want to create another dictionary where id is the key and value is the string `'Pass'` if a student gets more than 300 marks, else value is the string `'Fail'`. Also, we want to include only those

students who are in class 5. Let us write a dictionary comprehension for it. This will be a long comprehension, so we will write it step by step.

We will get the id and record by iterating over the `items` method.

```
{ for id, record in students.items() }
```

We want students of class 5 only, so we will add an `if` clause.

```
{ for id, record in students.items() if  
record['class'] == 5 }
```

Next, we will write the key expression. Key is equal to id so we write id at the place of key expression.

```
{id: for id, record in students.items() if  
record['class'] == 5 }
```

In the place of value, we have to write string 'Pass' or 'Fail' depending on the marks of the student. So, we can use the ternary operator here.

```
>>> {id: 'Pass' if record['marks'] > 300 else  
'Fail' for id, record in students.items() if  
record['class'] == 5 }
```

The ternary expression 'Pass' if record['marks'] > 300 else 'Fail' evaluates to string 'Pass' if marks are greater than 300 otherwise it evaluates to string 'Fail'. So finally, we will get the following dictionary:

```
{ '12AB': 'Pass', '13CP': 'Pass', '23CX': 'Fail',  
'19HY': 'Fail' }
```

Here is one more example:

```
text = 'Hello World !!!'
```

We have a string, and we want to make a dictionary where keys are characters from this string and values are their number of occurrences in the string. This is the dictionary that we should get.

```
{ 'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 2, 'W': 1,  
'r': 1, 'd': 1, '!': 3 }
```

Here is the dictionary comprehension that will give us the required dictionary:

```
>>> {ch: text.count(ch) for ch in text}
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 2, 'W': 1,
'r': 1, 'd': 1, '!': 3}
```

We iterate over the characters of the string and get each character in the variable `ch`, so at the place of key expression, we write `ch`, and at the place of value expression, we write the string `count` method. This gives us the perfect output, but there is some extra repeated work being done here. The letter 'l' occurs 3 times, so the count method is called 3 times; for the first time, the key-value pair is added to the dictionary, and the next two times, the value is updated. There is no problem with this but we have unnecessary extra steps for characters that are repeated. We can remove these extra steps by iterating over a set of characters of the string.

```
>>> {ch: text.count(ch) for ch in set(text)}
{'H': 1, '!': 3, 'l': 3, 'o': 2, ' ': 2, 'e': 1,
'd': 1, 'r': 1, 'W': 1}
```

We get the same result but this one is more efficient since we do not have to repeat the counting process for characters that are repeated.

9.10 Inverting the dictionary

We can use dictionary comprehension to create an inverted dictionary - a dictionary where the keys become values and values become keys. For example, suppose we have a dictionary where country names are keys and their capitals are values.

```
>>> d = {'India': 'New Delhi', 'France': 'Paris',
'Egypt': 'Cairo', 'Japan': 'Tokyo'}
```

We need to write a dictionary in which the keys and values are swapped, which means capitals become keys and countries become values. We can write a dictionary comprehension for it.

```
>>> {value: key for key, value in d.items()}
{'New Delhi': 'India', 'Paris': 'France', 'Cairo':
'Egypt', 'Tokyo': 'Japan'}
```

an

At the place of key expression, we have written `value` and at the place of value expression we have written `key`. This way we could invert our dictionary.

We know that there are two restrictions on the keys of a dictionary; they should be unique and immutable. However, there is no such restriction on the values of the dictionary. The dictionary that we want to invert might have duplicate values, or the values can be of any mutable type. The method of inverting the dictionary will work only if all the values in the dictionary are of an immutable type and all of them are unique because in the inverted dictionary, they are going to become the keys.

If you try to invert a dictionary that has a mutable value, then you will get a `TypeError`. If there are duplicate values in the dictionary, then some data may be lost. For instance, let us take the example of books and authors dictionary, which has multiple keys for a given value:

```
>>> d = {'Poems for kids': 'Joe',
        'Stories for kids': 'Zen',
        'Health is wealth': 'James',
        'Rhymes for Babies': 'Joe',
        'Stories for teens': 'Ted',
        'Be healthy': 'James'
        }

>>> {value: key for key, value in d.items()}
{'Joe': 'Rhymes for Babies', 'Zen': 'Stories for kids', 'James': 'Be healthy', 'Ted': 'Stories for teens'}
```

For every author, there is only one book in our resultant dictionary, the information of other books written by the author is lost. Most of the values will be lost and only one value will be there in the resulting dictionary as now the values are keys, and duplicate keys are not allowed. If we do not want to lose any value, we can use list comprehension.

We have seen earlier that we can get a list of the keys for a given value from a given dictionary using this list comprehension.

```
>>> [book_name for book_name, author in d.items()
if author == 'James']
['Health is wealth', 'Be healthy']
```

Now we can use this list comprehension inside our dictionary comprehension to avoid the loss of data while inverting.

```
>>> {value: [x for x,y in d.items() if y==value]
for key,value in d.items()}
{'Joe': ['Poems for kids', 'Rhymes for Babies'],
'Zen': ['Stories for kids'], 'James': ['Health is
wealth', 'Be healthy'], 'Ted': ['Stories for
teens']}
```

Now we do not have any loss of data. We can make it efficient by iterating over a set of values instead of items in a dictionary.

```
>>> { value: [x for x,y in d.items() if y==value ]
for value in set(d.values()) }
{'Zen': ['Stories for kids'], 'Joe': ['Poems for
kids', 'Rhymes for Babies'], 'Ted': ['Stories for
teens'], 'James': ['Health is wealth', 'Be
healthy']}
```

9.11 Set Comprehensions

A set comprehension creates and returns a new set. Here is the syntax for a set comprehension.

```
{expression for item in iterable if condition}
```

The syntax for set comprehensions is similar to that of dictionary comprehensions. The only difference is that in set comprehensions, you have a single expression, while in dictionary comprehension, you have two expressions separated by a colon. Like other comprehensions, set comprehension can also have multiple for and if clauses. We know that a set contains unique values, so whenever a new set is created, duplicate values will not be placed in the set. The values will not be in any particular order since sets are unordered structures. Let us see some examples:

```
>>> text = 'Hello !!! My name is Anthony  
Gonsalves, and you are .... ??'
```

We have this string and we want to create a set of all those characters in this string that are not alphanumeric. Here is the set comprehension to do this:

```
>>> {ch for ch in text if not ch.isalnum()}  
{ ' ', '.', ',', '!', '?' }
```

If you change the curly braces to square brackets, it becomes a list comprehension and you get a list.

```
>>> [ch for ch in text if not ch.isalnum()]  
[ ' ', '!', '!', '!', ' ', ' ', ' ', ' ', ' ', ' ', ' ',  
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',  
'?', '']
```

The next set comprehension creates a set of all consonants used in the string.

```
>>> {ch for ch in text.lower() if ch.isalpha() and  
ch not in 'aeiou'}  
{ 't', 'd', 'g', 's', 'r', 'h', 'm', 'n', 'v', 'l',  
'y' }
```

We want to ignore the case so we iterated over the string text converted to lowercase.

In our next example, we have a dictionary which has email id as keys, and values are dictionaries that contain course name and city name.

```
d = {'raj@xyz.com': {'course': 'Algorithms',  
'city': 'London'},  
     'dev@abc.com': {'course': 'Painting', 'city':  
'Delhi'},  
     'sam@pqr.com': {'course': 'Design Patterns',  
'city': 'London'},  
     'jim@xyz.com': {'course': 'Networking',  
'city': 'Delhi'},
```

```

        'pam@abc.com': {'course': 'Algorithms',
'city': 'Delhi'},
        'ray@abc.com': {'course': 'Painting', 'city':
'London'},
        'anu@xyz.com': {'course': 'Algorithms',
'city': 'London'},
        'bob@pqr.com': {'course': 'Data Structures',
'city': 'Tokyo'},
        'ted@abc.com': {'course': 'Algorithms',
'city': 'London'},
        'zen@abc.com': {'course': 'Painting', 'city':
'London'}
    }

```

From this dictionary, we want set of courses taken by students who are in London.

```

>>> {record['course'] for record in d.values() if
record['city'] == 'London'}
{'Design Patterns', 'Painting', 'Algorithms'}

```

If we use a list comprehension here, we will get a list of courses that will contain all duplicate values. This is why we need to create a set here, so that we get unique values.

A student can take multiple courses, so let us change our dictionary `d` a bit. Now we have a list of course names as the value for the key 'course'.

```

d = { 'raj@xyz.com': {'course': ['Algorithms',
'Painting'], 'city': 'London'},
        'dev@abc.com': {'course': ['Painting',
'Networking'], 'city': 'Delhi'},
        'sam@pqr.com': {'course': ['Design
Patterns', 'C', 'C++'], 'city': 'London'},
        'jim@xyz.com': {'course': ['Networking'],
'city': 'Delhi'},

```

```

        'pam@abc.com': {'course': ['Algorithms',
'Java'], 'city': 'Delhi'},
        'ray@abc.com': {'course': ['Painting',
'C++'], 'city': 'London'},
        'anu@xyz.com': {'course': ['Algorithms',
'C'], 'city': 'London'},
        'bob@pqr.com': {'course': ['Data
Structures', 'Java'], 'city': 'Tokyo'},
        'ted@abc.com': {'course': ['Algorithms',
'C++'], 'city': 'London'},
        'zen@abc.com': {'course': ['Painting'],
'city': 'London'}
    }

```

Now, to get the set of all courses in London, we have to change our comprehension. We will have to use nested for clause. Let us write this comprehension step by step:

```

{ for record in d.values() }
{ for record in d.values() for course in
record['course'] }
{ for record in d.values() for course in
record['course'] if record['city']=='London' }
{ course for record in d.values() for course in
record['course'] if record['city']=='London' }

```

We get the following set by using this comprehension.

```

{'C++', 'Algorithms', 'C', 'Painting', 'Design
Patterns'}

```

9.12 When not to use comprehensions

After working through the examples and exercises, you will have a good understanding of comprehensions in Python and will be able to write even complex comprehensions. There is no limit as to how complicated a

comprehension can be, but you should avoid using long and complex comprehensions as they will be too difficult for others to understand. If you have to split your comprehension over multiple lines, then consider using normal loop and `if` statements instead. The indented syntax of loops makes your code clearer if you have many levels of nesting. Nested comprehensions and comprehensions with multiple and complex `for` and `if` clauses can confuse the reader of the code, and hence their use should be avoided.

Code readability is more important than code conciseness. Do not compromise on readability for saving a few lines of code. Try to keep your comprehensions short and simple, longer ones are incomprehensible and difficult to maintain. So, it is okay to use comprehensions for simple cases, but when things get complex, use `for` loop which will be more readable.

Exercise

What will be the output of the code given in questions 1 to 9?

1.

```
L = [[n, n * 2, n * 3] for n in range(1, 4)]  
print(L)
```
2.

```
L = [(i, j) for i in range(5) for j in  
range(6)]  
print(len(L))
```
3.

```
s = 'What is your name'  
L = [item[0] for item in s.split()]  
print(L)
```
4.

```
L = [1, 2, -4, 5, -2, 9, -7]  
L1 = [n for n in L if n > 0 else 0]  
print(L1)
```
5.

```
L = [[1, 2, 11, 13], [12, 34, 56, 10], [13,  
77, 89], [56, 78]]
```

```

list1 = [min(sublist) for sublist in L]
print(list1)
6. L = [4, 5, 3, 7, 9, 2, 8, 1]
   L1 = [n // 2 if n % 2 == 0 else n * 2 for n in
   L]
   print(L1)
7. L1 = [1, 2, 3]
   L2 = [4, 5, 6]
   L3 = [x * y for x in L1 for y in L2]
   L4 = [x * y for x, y in zip(L1, L2)]
   print(L3, L4)
8. b = [0 for i in range(20)]
   c = [0] * 20
   print(b == c)
9. L = [-2, 1, -4, 21, 20, -3, -7, 9, 0]
   L1 = [n for n in L if n < 0] + [n for n in L
   if n >= 0]
   print(L1)
10. M = [[1, 4, 8, 3],
         [2, 5, 6, 3],
         [7, 9, 5, 8],
        ]

```

Which of these list comprehensions will give us a matrix whose elements are double the elements of this matrix M?

- (A) `[x * 2 for x in row for row in M]`
- (B) `[x * 2 for row in M for x in row]`

(C) `[[x * 2 for x in row] for row in M]`

11. Which of the following comprehensions will give us this list?

`[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]`

(A) `[[n for n in range(4)] for x in range(3)]`

(B) `[[n for n in range(3)] for x in range(4)]`

(C) `[n for n in range(3) for x in range(4)]`

12. In the following code, we get the original dictionary by inverting it two times.

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d = {val: key for key, val in d.items()}
>>> d = {val: key for key, val in d.items()}
>>> d
{'a': 1, 'b': 2, 'c': 3}
```

Will we always get the original dictionary on inverting it twice?

13. List comprehension is

(A) an expression (B) a statement

14. Write a list comprehension to create a list that contains square root of only positive numbers in this list.

`L = [81, -9, 4, 16, -25, 64]`

15. Write a set comprehension to create a set of 10 random numbers that are in the range 1 to 1000.

16. Use a list comprehension to construct this list:

`['5x', '7x', '9x', '11x', '13x', '15x', '17x']`

17. In the following code, list `L` is created from lists `X` and `Y`.

```
X = [1, 2, 3, 4]
Y = [5, 6, 7, 8]
L = [X[i] * Y[i] for i in range(len(X))]
```


Create the list `L` using `zip` instead of `len` and `range` combination.

18. What is the difference between these three pieces of code?

```
(A) names = ['ted williams', 'John smith', 'tim jones']
```

```
        names = [name.title() for name
in names]
```

```
        print(names)
```

```
(B) names = ['ted williams', 'John smith', 'tim jones']
```

```
        for name in names:
```

```
            name = name.title()
```

```
        print(names)
```

```
(C) names = ['ted williams', 'John smith', 'tim jones']
```

```
        for i in range(len(names)):
```

```
            names[i] =
```

```
names[i].title()
```

```
        print(names)
```

19. The following three lists contain names, heights, and weights of people at corresponding indices. Heights are in cms, and weights are in kilograms.

```
names = ['John', 'Joe', 'Ted', 'Sam', 'Jack', 'Jill']
```

```
heights = [160, 152, 147, 167, 177, 182]
```

```
weights = [54, 60, 90, 77, 87, 67]
```

Write a list comprehension to create a list of 2 element tuples where first element is the name, and second element is the BMI of the person. Body Mass Index (BMI) is calculated by dividing body weight in kg by the square of height in meters. For example, if

weight is 70 kg and height is 170 cm, then then BMI is $70/(1.7*1.7)$
= 24.2

20. This list comprehension creates a list of cubes of odd numbers:

```
cubes = [n ** 3 for n in range(5, 21) if n % 2  
!= 0]
```

Write it without the if clause.

21. From the following list named `data`, create a new list named `integers` that contains all the integer values from the given list. Similarly, create two more lists named `floats` and `strings` that contain all float and string values from this list.

```
data = [1, 2, 3.4, 6, 'd', 8, 7, 9.8,  
'Python']
```

22. Create a set of all possible ordered pairs, wherein each pair, the first element is from the list named `size`, and the second element is from the list named `garment`.

```
size = ['S', 'M', 'L', 'XL']  
garment = ['Shirt', 'Trousers', 'Jacket']
```

23. From the given two lists, create a dictionary where the key is an element from the list `names` and corresponding value is the element at same index from the list `marks`.

```
names = ['Sam', 'Ted', 'Joe', 'Max']  
marks = [90, 98, 78, 89]
```

24. Write a list comprehension to get a list of all the factors of a given number.

25. The following comprehension creates a dictionary where the keys are numbers from 1 to 20, and corresponding values are lists of factors of the number.

```
d1 = {num: [n for n in range(1, num + 1) if  
num % n == 0] for num in range(1, 20)}
```

Change this code to a more comprehensible form using a for loop.

26. Write list comprehension to flatten this nested list.

```
x = [[10, 20, 30], [40, 50, 60], [60, 70, 80]]
```

After flattening, the list should look like this:

```
[10, 20, 30, 40, 50, 60, 60, 70, 80]
```

27. The following dictionary contains names of products as keys and prices as values.

```
prices = {'pencil': 23, 'pen': 34, 'eraser': 12, 'sharpener': 13, 'marker': 30}
```

Write a list comprehension to create a list named `costly_products` that contains names of those products whose cost is more than 20.

28. Given this dictionary where country name is the key and currency name is the value, how will you find out the name of the country whose currency is 'Yen'.

```
d = { 'India': 'Rupee', 'UK': 'Pound',  
      'France': 'Euro', 'Japan': 'Yen', 'Austria':  
      'Euro', 'Bangladesh': 'Taka', 'Italy': 'Euro' }
```

Create a list of countries that have 'Euro' as the currency.

29. From the following dictionary, create a list of names of those students whose total marks are more than 200.

```
students = {105416: {'name': 'John',  
                    'city': 'Paris',  
                    'dob': '12-01-2000',  
                    'marks': {'Maths':  
89, 'Physics': 78, 'Chemistry': 91}},  
            },  
            144547: {'name': 'Dev',  
                    'city': 'London',  
                    'dob': '13-11-1998',
```

```

        'marks': {'Maths':
58, 'Physics': 57, 'Chemistry': 68},
    },
    132399: {'name': 'Mary',
            'city': 'Paris',
            'dob': '01-05-1997',
            'marks': {'Maths':
99, 'Physics': 87, 'Chemistry': 88},
    }
}

```

30. From the dictionary of the previous question, create a list of names of those students who were born in 1998 or later.
31. Write a list comprehension that returns the sum of the following two matrices M1 and M2.

```

M1 = [[1, 4, 8, 3],
      [2, 5, 6, 3],
      [7, 9, 5, 8]
      ]
M2 = [[3, 5, 2, 3],
      [5, 2, 7, 9],
      [2, 8, 1, 8]
      ]

```

32. This following list L contains 6 references to the same list:

```
L = [[None]*3] * 6
```

How would you create this list L to avoid this aliasing problem.

33. Write a list comprehension to create a list of lists that represents transpose of the matrix represented by the following list M. Transpose of a matrix is a new matrix in which rows become columns and vice versa.

```

M = [[5, 4, 3, 6],
     [6, 3, 1, 2],
     [8, 9, 7, 4]]

```

34. Write a list comprehension that can replace this code:

```
pairs = []  
for n1 in range(4):  
    for n2 in range(4):  
        if n1 != n2:  
            pairs.append((n1, n2))
```

35. Write a list comprehension to create a list of lists that represents the matrix of size 3 X 4 with all its elements initialized to 0.

36. Write a dictionary comprehension to create a dictionary that has integers from 1 to 20 as the keys, and values are squares of the keys.

37. `L = [2, 4, 6, 7, 5]`

Write a dictionary comprehension to create the following dictionary from the list `L`.

```
{ 2: [1, 2],  
  4: [1, 2, 3, 4],  
  6: [1, 2, 3, 4, 5, 6],  
  7: [1, 2, 3, 4, 5, 6, 7],  
  5: [1, 2, 3, 4, 5]  
}
```

38. The following four lists contain the names and marks of students in three subjects:

```
names = ['Ted', 'Sam', 'Jim', 'Rob', 'Anu']  
maths = [98, 67, 54, 88, 95]  
physics = [88, 64, 78, 99, 78]  
chemistry = [78, 67, 45, 79, 87]
```

Write a dictionary comprehension to create the following dictionary from the above four lists.

```
{ 'Ted': [98, 88, 78],
  'Sam': [67, 64, 67],
  'Jim': [54, 78, 45],
  'Rob': [88, 99, 79],
  'Anu': [95, 78, 87]
}
```

39. Create this nested dictionary from the four lists given in the previous question.

```
{ 'Ted': { 'Maths': 98, 'Physics': 88,
           'Chemistry': 78},
  'Sam': { 'Maths': 67, 'Physics': 64,
           'Chemistry': 67},
  'Jim': { 'Maths': 54, 'Physics': 78,
           'Chemistry': 45},
  'Rob': { 'Maths': 88, 'Physics': 99,
           'Chemistry': 79},
  'Anu': { 'Maths': 95, 'Physics': 78,
           'Chemistry': 87}
}
```

40. From the following dictionary, create another dictionary that contains only those key value pairs where the email domain is xyz.com.

```
d = { 'raj@xyz.com':      { 'course':
  'Algorithms',          'city': 'London'},
      'dev@abc.com':      { 'course':
  'Painting',            'city': 'Delhi'},
      'sam@pqr.com':      { 'course':
  'Design Patterns',     'city': 'London'},
      'jim@xyz.com':      { 'course':
  'Networking',          'city': 'Delhi'},
```

```

        'pam@abc.com': {'course':
'Algorithms',      'city': 'Delhi'},
        'ray@abc.com': {'course':
'Painting',        'city': 'London'},
        'anu@xyz.com': {'course':
'Algorithms',      'city': 'London'},
        'bob@pqr.com': {'course':      'Data
Structures',      'city': 'Tokyo'},
        'ted@abc.com': {'course':
'Algorithms',      'city': 'London'},
        'zen@abc.com': {'course':
'Painting',        'city': 'London'}
    }

```

41. From the dictionary `d` given in the previous question, create a new dictionary that has all the key value pairs of dictionary `d`, with all occurrences of `pqr.com` changed to `pqr.org`.
42. A training session on design patterns needs to be conducted, and all the registrations have been completed. The following dictionary comprises registration IDs as keys, paired with another dictionary as their corresponding values.

```

trainees = {'12AB': {'name':      'Ash',
'experience': 12,      'language':
'C++'},
            '34CD': {'name':      'Dev',
'experience': 5,      'language':
'Python'},
            '55AB': {'name':      'Raj',
'experience': 10,     'language':
'C++'},
            '67CD': {'name':      'John',
'experience': 3,      'language':

```

```

'Java'},
    '23ED': {'name': 'Drek',
'experience': 7, 'language':
'Python'},
    '35ED': {'name': 'Amit',
'experience': 4, 'language':
'Python'}
}

```

The trainer wants to provide hand-outs of sample programs in all the languages that trainees have chosen. From this dictionary, find a set of all the languages in which the trainer needs to provide programs.

```

43. emp = {'id01': {'name': 'Dev', 'phone':
'08056771173'},
    'id02': {'name': 'Raj',
'phone': '01176791193'},
    'id03': {'name': 'Ami',
'phone': '08056774473'},
    'id04': {'name': 'Anita',
'phone': '011767976193'},
    'id05': {'name': 'Sam',
'phone': '08056771173'},
    'id06': {'name': 'Reena',
'phone': '02276791193'},
    'id07': {'name': 'Akul',
'phone': '08056774473'},
    'id08': {'name': 'Amar',
'phone': '011767976193'}}

```

This is a dictionary of all employees where key is the id of an employee and value is another dictionary that contains name and phone number of the employee. In the phone number, the first three

characters represent the code of a city. Make a list of all those employees who have city code 080.

44. Find out the number of unique cities, whose code appears in the dictionary `emp` of previous question.

45. The following dictionary maps city codes to city names.

```
cities = {'080': 'Bengaluru', '044': 'Chennai',  
          '040': 'Hyderabad', '011': 'Delhi',  
          '022': 'Mumbai'}
```

Create a set that contains the names of all cities whose code appears in the dictionary `emp` of *Question 43*.

46. From the dictionary `emp`(of Q43) and dictionary `cities`(of Q45), create a list of employees who are in 'Delhi'.

47. What is the difference between these two list comprehensions? Write the equivalent for loop code for both of them.

```
L1 = [x * y for x in [3, 6, 7] for y in [4, 5, 6]]
```

```
L2 = [[x * y for x in [3, 6, 7]] for y in [4, 5, 6]]
```

48. The following statement initializes a tic-tac-toe game board using a list of lists. Is this the correct way to initialize? If not, write the correct way.

```
board = [[' ']*3]*3
```

49. From the following list, create a dictionary in which keys are the elements of this list, and the corresponding values are lists that contain all the indices where the particular element is present in the list.

```
numbers = [11, 20, 30, 24, 67, 30, 14, 30, 67, 52, 20]
```

Here is the resultant dictionary that you need to create:

```
{11: [0], 20: [1, 10], 30: [2, 5, 7], 24: [3],  
67: [4, 8], 14: [6], 52: [9]}
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[**https://discord.bpbonline.com**](https://discord.bpbonline.com)



A function is a named block of code that performs a specific task. It groups together some statements so that they can be used multiple times in a program. We have already used built-in functions such as `print`, `type` and `id`. Python provides many more built-in functions like these, and we can also create our own functions which are called user defined functions. So basically, we have two types of functions in Python; built-in functions and user defined functions. Built-in functions are some general-purpose functions that are already written, we just use them in our program. User defined functions are written by programmers to suit specific needs of their program. In this chapter, we will see how to define and use user defined functions. Before going into the details of defining and using functions, let us see why we need to use functions.

The main advantage of using functions is code reusability; they make the code reusable and reduce code duplication. For example, suppose in your program you need to perform a task that takes 20 lines of code. You write those 20 lines of code, test it, and find that it works perfectly. After a while, you realize that you need to perform this particular task multiple times in your program. A simple solution would be to copy and paste. Whenever you need to perform that particular task, just copy and paste those 20 lines of code.

Functions provide a better solution in such scenarios. Let us see what we will do if we use functions. We will write the 20 lines of code separately and give it a name; suppose we name it `TaskA`. Now, wherever we need to perform this specific task in our program, all we need to do is tell the interpreter to perform `TaskA`, instead of specifying all 20 lines of code.

In this process, we have two main things – defining a function and calling a function. Writing the block of code and naming it, is called defining the function, and invoking the block of code by its name is called a function call. The exact syntax of defining and calling a function will be shown in a little while, here I am just trying to explain to you the benefits of using functions.

The first benefit is that functions make your program shorter as you do not have to repeat the code. You have just one copy of the code and can reuse it in many places. In the copy-paste approach you have multiple copies of the same code, and it makes the program lengthy. So, functions reduce code duplication, and they make the code reusable. This reduces development time and prevents errors. Functions allow us to implement the DRY (Don't Repeat Yourself) principle of software development.

Suppose after some time your friend suggests a more efficient approach to code your TaskA, or you find a bug in it that needs to be fixed, or perhaps your senior instructs you to perform this task differently. Whatever may be the reason, in the future, if you decide to update the code, then in the copy-paste approach, you will have to make changes at every place wherever you have pasted the code. This could potentially lead to the occurrence of undetected bugs caused by failures in the copying and pasting process.

If you had created a function, you will have to make changes only in one place, where you have defined the function, and the rest of the program will automatically be updated since it is just calling this function. So, using functions is good for future maintenance also.

Once you make a function, you can use it in multiple places in your program, and you can share it with others also to use it in their programs if required. It reduces development time that is the work done by a programmer can be used by others.

Another significant advantage of using functions is that they let you break your system into smaller and more manageable pieces of code. Consider writing a program with thousands of lines of code, statement after statement, and lots of Python statements. If someone reads this program, it will be very difficult to understand what it does. This type of program is also very difficult to debug and manage. If we are using functions, we can break the program into smaller manageable parts that are easier to code, and you can give meaningful names to each piece of code. This way, you can separate

different parts of the program, develop them, and test them separately. This not only enhances the comprehensibility of the program but also makes it easier to maintain. So, functions provide modularity to your program. They organize your code so it becomes easy to understand what each part of the code does. Also, different people working on a complex and lengthy program can work on separate parts of the program independently. As your program grows larger, it becomes important to reduce its complexity by breaking it into functions of manageable size.

The next advantage is that functions hide implementation details from their users. For example, we have been using built-in functions without knowing the details of how they are implemented. The built-in functions are predefined, and the definitions are hidden from us. For example, we know that the function named `type` will return the type of an object, but how it does, we do not know, and we do not want to know also. The users only want to get the work done; they are not concerned about how the work is getting done. So, with functions, it becomes simple to write the program and understand *what* work is done by each part of the program. The details of *how* the work is done are hidden inside appropriate functions (definitions). This concept is called abstraction. A programmer using a built-in function or a function from some other library sees the function as a black box that hides all the details but gets their work done.

Having obtained a general understanding of functions and their importance, we can proceed to the next step: defining and using functions in our programs.

10.1 Function Definition

The definition of a function creates the function while the function call runs the code inside the function. First, we will see the function definition. A function in Python is defined by a `def` statement. Here is the syntax of a function definition in Python:

```
def function_name(par1, par2, ... ):
    statement1
    statement2
    statement3
    .....
    .....
```

Figure 10.1: *Syntax of a function definition*

The function definition consists of two parts - the header line and the function body. The header line begins with the keyword `def`, followed by an identifier, which is the name of the function, and then a pair of parentheses, which may enclose some identifiers separated by commas. These identifiers are parameter names. Parameters are input to the function; they enable us to pass different values to the function. The whole header line ends with a colon. This header line is also known as the function's signature. If the function does not have parameters, it must still include the empty parentheses. Here is the syntax of a function definition for a function that does not have any parameters:

```
def function_name( ):
    statement1
    statement2
    statement3
    .....
    .....
```

Following the header line is the body of the function. The function body consists of one or more Python statements; it contains the full code of the function. All the statements inside the function body should be indented by the same amount from the header line. It is best to follow the standard of 4 spaces. The code block of the function ends with the first non-indented statement. Writing a code block with the help of indentation is the same as we have seen in `if`, `while`, and `for` statements.

The name of the function can be any valid Python identifier, they follow the same rules that we saw for naming identifiers. The name of the function should be descriptive, and conventionally, it is in all lower case with words separated by underscores. Let us see examples of some function definitions:

```
def greet():  
    print('Hello')
```

In this definition, `greet` is the function name; the function does not take any parameters, so the parentheses are empty. The function body consists of only one statement.

```
def greet1(name):  
    print('Hello', name)
```

This function `greet1` takes one parameter, which is placed inside the parentheses. Here also, the function body has only one statement. We have used the parameter named `name` inside the function body.

```
def calculate(a, b):  
    print(a + b)  
    print(a - b)  
    print(a * b)
```

This function named `calculate` takes two parameters, `a` and `b`. The function body here consists of 3 statements, all of them indented by the same amount from the header line.

```
def print_blank_lines(n):  
    for i in range(n):  
        print()
```

This function `print_blank_lines` takes one parameter and inside the function body there is a for loop.

If we write all these function definitions in a `.py` file and execute that program, we will not see any output in the output window. When the program is run, the four `def` statements are executed and they create 4 function objects. Everything in Python is an object, and so are functions. The `def` statement is an executable statement that creates a new function object and assigns that object to the function's name. So, by running the program, we will get 4 function objects, which are assigned to the names `greet`, `greet1`, `calculate` and `print_blank_lines`.

After running the program, if we check the type of the name `greet` in the Shell window, we will see that it is of type `function`.

```
>>> type(greet)
<class 'function'>
```

The name `greet` is referring to an object of type `function`. We can see the id of the function object.

```
>>> id(greet)
2331036755072
```

If we write the function name on the prompt, we get this information.

```
>>> greet
<function greet at 0x0000021EBC9E2480>
```

This is the same id that was returned by the `id` function, it is printed in hexadecimal. you can see this by using the `hex` function.

```
>>> hex(id(greet))
'0x21ebc9e2480'
```

So, we have a function object and the name `greet` is referring to it. Similarly, we have three more function objects and the names `greet1`, `calculate` and `print_blank_lines` refer to those objects.

10.2 Function call

The statements inside the function body are not executed when the `def` statement executes. To execute the statements inside the function body, we have to call the function. So, now let us see how to call a function. If the function has no parameters, then it is called by appending empty parentheses to the function name. The function `greet` that we have written has no parameters, so it called like this:

```
greet()
```

If we add this line in our file and run the program, we can see that the code inside the function `greet` is executed and we get `Hello` printed out in the output window. So, a function call instructs the interpreter to execute the body of the function; it is also called function invocation. A function is defined once and can be called any number of times in our program. If we add the same function call two more times in our file then the function is

called three times, and `Hello` is printed three times. So, each time a function is called, the code inside it is executed.

Now, let us see how we can call the function `greet1`. From the definition, we can see that it takes one parameter. This means that this function needs some input, when it is called. So, while calling the function we will send a value to this function.

```
greet1('Sam')
```

The string that we have provided here inside the parentheses in the call will be assigned to the parameter name that we have in the function definition. When we run the program, this function call will call the function `greet1` with parameter name referring to `'Sam'` and so `Hello Sam` will be printed. If you do not send any argument and leave the parentheses empty, then there will be an error.

```
greet1()    # gives error
```

We can call the function `greet1()` with any other value also.

```
greet1('Bob')
```

Now the parameter refers to the string `'Bob'`, so `Hello Bob` will be printed in the output. We can see that parameters make our function flexible. The same function code can be executed for different values. If the facility of parameters was not there, we would have to write different functions that perform the same task but vary only in the data that they work upon. With the help of parameters, you can reuse the statements with different data.

Now let us call the function `calculate`. It takes two parameters which means that when we call the function, we have to supply two values inside the parentheses.

```
calculate(8, 5)
```

The first value 8 will be assigned to the first parameter `a`, and the second value 5 will be assigned to the second parameter `b`. When this call is executed, we will get the values of `8+5`, `8-5` and `8*5` printed in the output. If we call it with some other values, we get a different output.

```
calculate(6.5, 4)
```

Now we will get 10.5, 2.5 and 26.0 as the output.

The values that we provide in the function call are called arguments. Thus, parameters are names defined in the function definition and arguments are the values that we pass in the function call. When a function call executes, first all the arguments are assigned to corresponding parameters and then, the statements inside the function body are executed. Arguments are used to send data to the function so that the function can use the data while doing its work.

The four function calls that we have written are equivalent to writing the following statements.

<code>greet1('Sam')</code>	→	<code>name = 'Sam'</code> <code>print('Hello', name)</code>
<code>greet1('Bob')</code>	→	<code>name = 'Bob'</code> <code>print('Hello', name)</code>
<code>calculate(8,5)</code>	→	<code>a = 8</code> <code>b = 5</code> <code>print(a + b)</code> <code>print(a - b)</code> <code>print(a * b)</code>
<code>calculate(6.5,4)</code>	→	<code>a = 6.5</code> <code>b = 4</code> <code>print(a + b)</code> <code>print(a - b)</code> <code>print(a * b)</code>

Figure 10.2: Function calls

First, the arguments are assigned to the corresponding parameter names and then the code inside the function body is executed. For example, in the call `calculate(8, 5)`, 8 is assigned to `a`, 5 is assigned to `b` and then the code is executed. Once we have defined a function, we can call it as many times as we want and each time, we can send different arguments.

Actually, these function calls are not new to us, we have been writing function calls since our first Python program. When we were calling built-in functions like `print` or `type`, we were calling functions written by someone else, now we are calling the functions that we have written. So, we can see that the syntax for calling both user defined functions and the built-in functions is the same. We write the function name followed by

parentheses with possibly some arguments inside it. Now let us call the fourth function that we had defined.

```
print_blank_lines(5)
```

Here the argument is 5, so when this call will be executed, the value of parameter `n` will be 5 and this call will print 5 blank lines.

A function call can be placed inside the definition of another function. Here is an example-

```
def greet1(name):  
    print_blank_lines(3)  
    print('Hello', name)  
    print_blank_lines(2)
```

The function `greet1()` is calling the function `print_blank_lines()` two times inside its body. So, a function can call another function inside its definition. Now let us call this new version of `greet1`.

```
greet1('Sam')
```

When this call is executed, first the `print_blank_lines()` function is executed with parameter 3, so it prints three blank lines, and then `Hello Sam` is printed, and then the `print_blank_lines()` function is called with parameter 2, so two blank lines are printed.

Thus, if the function that you are writing becomes too complex and lengthy, you can place parts of the code in one or more separate helper functions and then call those helper functions inside your function.

10.3 Flow of control

The code written in our file normally executes sequentially from top to bottom. We have seen how this flow of execution is altered by `if-else` statements and loops. Now let us see how this flow is affected by functions.

A function definition does not alter the flow of control, but a function call does. When a function call is encountered, the control is transferred to that function. After all the statements inside the function's body are executed,

control returns back to the place where the function was called and the program flow resumes at the point just after the function call.

The code block of a function can include calls to other functions, so while executing a function code block, the control might have to jump to another function. Python keeps track of all these calls, and knows where to return once a function code block has finished executing.

10.4 Parameters and Arguments

We have seen that when we call a function that has parameters, we need to send some values for those parameters; these values are called arguments. Parameters are the names inside the parentheses of the header of function definition and arguments are the values that we supply in the function call. The arguments provided during the function call are assigned to the corresponding parameters present in the function definition. The function body uses the argument values by referencing the corresponding parameter name.

In the examples that we have seen till now, we have sent only literal values as arguments. Arguments can be written in the form of variables or expressions also. Suppose we have this function named `add` with two parameters.

```
def add(a, b):  
    print(a + b)
```

We can call it like this with literal values 2 and 3.

```
add(2, 3)
```

In this call, parameter `a` is assigned value 2 and parameter `b` is assigned value 3. Now suppose we have two variables in our program named `num1` and `num2`.

```
num1 = 7  
num2 = 9
```

We can write a call, as shown below:

```
add(num1, num2)
```

Here we are sending variables as arguments. In this call, parameter `a` is assigned the value of variable `num1`, and parameter `b` is assigned the value of variable `num2`. You can even send expressions as arguments.

```
add(num1 * 2, num2 / 3)
```

Here parameter `a` is assigned the value of expression `num1 * 2`, and parameter `b` is assigned the value of expression `num2 / 3`.

If the number of arguments provided in the call is not equal to the number of parameters in the definition, then the compiler will complain. The following calls will result in an error.

```
add()  
add(1)  
add(1, 2, 3)
```

In the next example, we will see how to avoid code duplicity by introducing parameters in our function.

```
def display_list_decimal(L):  
    for n in L:  
        print(n, end=' ')  
    print()  
def display_list_binary(L):  
    for n in L:  
        print(f"{n:b}", end=' ')  
    print()  
def display_list_octal(L):  
    for n in L:  
        print(f"{n:o}", end=' ')  
    print()  
def display_list_hexadecimal(L):  
    for n in L:  
        print(f"{n:X}", end=' ')  
    print()
```

```
numbers = [134, 2567, 366, 521, 689]
display_list_decimal(numbers)
display_list_binary(numbers)
display_list_octal(numbers)
display_list_hexadecimal(numbers)
```

Output:

```
134 2567 366 521 689
10000110 101000000111 101101110 1000001001
1010110001
206 5007 556 1011 1261
86 A07 16E 209 2B1
```

We have written four functions for displaying integers of a list in different bases. The code of these functions is very similar; the only difference is in the character inside the curly braces of the f-string. Instead of making four separate functions, we can make just a single function `display_list` by introducing a parameter for base.

```
def display_list(L, base):
    if base == 2:
        ch = 'b'
    elif base == 8:
        ch = 'o'
    elif base == 16:
        ch = 'X'
    else:
        ch = 'd'
    for n in L:
        print(f"{n:{ch}}", end=' ')
    print()
```

```
numbers = [134, 2567, 366, 521, 689]
display_list(numbers, 10)
display_list(numbers, 2)
display_list(numbers, 8)
display_list(numbers, 16)
```

In our function `display_list`, we initialize the value of variable `ch` depending on the value of the parameter `base`. If `base` is 2, `ch` is 'b', if `base` is 8, `ch` is 'o', if `base` is 16 `ch` is 'x', otherwise `ch` is 'd'. This variable `ch` is used in the curly braces of the f-string.

So, we can see that parameters can sometimes reduce code duplication. Instead of writing two or more functions with similar code, we can write a single function.

Some texts refer to arguments as actual parameters and parameters as formal parameters. In this book, we will use term parameters for names defined in function definition and arguments for values that we send in the function call.

10.5 No type checking of arguments

In the function definition header, there are no types declared for parameters, so there is no restriction on the type of argument that can be sent. Unlike some other languages like C++ or Java, Python lets you pass any type of object as argument.

So, we can send float values, strings, or even lists to our `add` function that we have defined.

```
add(2.1, 7.6)
add('Sponge', 'Bob')
add([1, 2, 3], [4, 5, 6])
```

When these calls are executed, we will get the following output.

```
9.7
```

SpongeBob

```
[1, 2, 3, 4, 5, 6]
```

There is no type checking done by the function. The interpreter will check only if you have passed the correct number of arguments, it does not care about the type. The function executes correctly as long as the type of arguments supports the operations performed inside the function.

Inside the function we are performing addition operation ($a + b$), and this operation is supported for integers, floats, strings, and lists so all the calls worked correctly. In the case of integers and floats, the numbers were added and in the case of strings and lists, joining was done.

If we pass two dictionaries to this function, then we will get an error, or if you pass a list and an integer, then also you will get an error.

```
add([1, 2, 3], 7)    # Error
```

Thus, you will get an error at run time if you send a type of value that does not support the operations performed inside the function. You can specify in the documentation what type of arguments are expected, but there is no type checking done. The user is free to send any type of object to the function.

If you want, you can do the type checking yourself by adding tests to check for the type of arguments at the beginning of the function. You can do it with the help of built in functions `type()` or `isinstance()`. But this testing will reduce the flexibility of your function. It will constraint your function to work on specific types only.

This behavior that our `add` function is exhibiting is called polymorphism, which means one thing, many forms. We can use this function `add` for different types of objects, as long as the type supports the operations performed in the function.

10.6 Local Variables

We can define variables inside the function body. These variables are considered local to the function. Let us see this with the help of an example. We have the following function `summation` that sums up all the numbers from `a` to `b`.


```
def summation(a, b):  
    s = 0  
    for i in range(a, b + 1):  
        s += i  
    print(s)
```

In this function, the variable `s` is a local variable. It exists only while the function is running and is destroyed when the function finishes executing. If you try to access this variable outside the function, you will get an error. The variable `i` in the for loop is also a local variable. Parameters in the function definition are like any other variable created inside the function definition, except that they are assigned automatically. Thus, the parameters of a function are also local variables of the function.

The variables defined inside the function, along with the parameters, comprise all the local variables of the function. All these names come into existence when the function is called and are destroyed when the function finishes execution. The local variables are visible only inside the function, they cannot be used anywhere else in the program. If you try to access any local variable outside the function then the interpreter will complain. This also means that you can use the same variable name inside different functions, without any clash.

```
def add(a, b):  
    s = a + b  
    print(s)
```

Using the name `s` in this function `add` is acceptable. The name `s` is different for both the functions `summation` and `add`, one is visible inside the `summation` function, and the other `s` is visible inside the `add` function.

A variable that is created outside any function is a global variable. We will talk more about local and global variables later.

10.7 return statement

Till now we have been using functions that display the data or the result directly. If the function wants to return some data to the caller, then it needs to use the return statement. Let us understand this with the help of an example.

```
def simple_interest(p, r, t):  
    si = (p * r * t) / 100  
    print(f'Simple interest is {si}')
```

```
principal = 2000  
rate = 5  
time = 4  
simple_interest(principal, rate, time)
```

Here, we have a function named `simple_interest` that takes in `principal`, `rate`, and `time`, calculates the simple interest, and prints it. The parameters are `p`, `r`, and `t`, and inside the function, we have a local variable named `si` which is used to store the simple interest. The value of this variable `si` is printed inside the function.

In the function call, we have sent the variables named `principal`, `rate`, and `time` as arguments. When this program is run, it prints the interest. Now suppose we want to print the amount that is to be paid after four years. We know that it will be equal to principal plus interest.

```
amount = principal + interest
```

To get the amount, we need to add the value of interest to the principal. Now, how do we get the interest? The function that we have made is just printing the value of interest and after that it terminates and as soon as it terminates the local variable `si` that holds the value of interest is destroyed. So, we cannot write the following:

```
amount = principal + si      # Can't use name si  
outside the function
```

The solution is to return the value of `si` from the function instead of just printing it. So now we make a small change in our function definition.

```
def simple_interest(p, r, t):
```

```
    si = (p * r * t) / 100
    return si
principal = 2000
rate = 5
time = 4
interest = simple_interest(principal, rate, time)
amount = principal + interest
print(f'Simple interest is {interest}')
print(f'Amount is {amount}')
```

Instead of printing `si`, we return the value of `si` from the function. For this, we have written the `return` keyword followed by the name `si`. In our main program, we make a variable named `interest` and the value returned by this function is assigned to `interest`. After this, the `principal` and `interest` are added to get the amount.

Now let us see the syntax of the return statement.

```
return [expression]
```

`return` keyword is written, followed by an optional expression. When this statement is encountered inside a function definition, the function's execution stops immediately, and the control is returned to the caller. The value of the expression is used as the return value which is returned to the caller. Here are some examples of return statements:

```
return 345
return 'hello'
return True
return si
return x
return x + y*3
return None
return
```

The expression can be any literal value like number, string or Boolean value True or False, it can be any variable as we have in our simple interest example, or it can be any expression combining all of these, or you can even return None. In our `simple_interest` function we could simply write `return (p * r * t) / 100`, instead of storing the value of the expression in a variable and then returning that variable.

It is optional to specify the expression in the return statement. You can write a return statement without any expression. The return statement without any expression is generally used to stop the execution of the function when a condition is checked. For example, in the following function definition, we check the condition `x < 0`, and if it is True, we just return from the function, without executing the rest of the code.

```
def func(x, y):  
    if x < 0:  
        return  
    .....  
    .....
```

Here we do not want to return any value to the caller, we have used the `return` statement just to exit from the function immediately. So, the `return` statement is used to exit from a function and it can also return a value, the returned value becomes the value of the function call.

In Python, a `return` statement without any expression is the same as `return` statement with a value of `None`. When you write `return` without any expression, Python returns `None`.

A function that has no `return` statement inside it, automatically returns `None` when the function has finished executing. For example, when the following function is called and its execution is completed, `None` will be automatically returned from it.

```
def calculate(a, b):  
    print(a + b)  
    print(a - b)  
    print(a * b)
```

A `return` statement can be placed anywhere inside the body of the function, and there can be multiple `return` statements in a single function, and these are often part of conditional logic. For example, the following function returns 1 if `a` is greater than `b`, -1 if `a` is less than `b` and 0 if `a` is equal to `b`. So, we have three `return` statements in this function:

```
def compare(a, b):  
    if a > b:  
        return 1  
    elif a < b:  
        return -1  
    else:  
        return 0
```

A function call evaluates to its return value, and so a function call that returns a value can be placed at any place where that function's return value can be placed. For example, if a function returns an `int`, you can place the function call at any place where you can place an `int`. Let us understand this with the help of examples:

```
def square(x):  
    return x * x  
  
def add(x, y):  
    return x + y  
  
s = square(4)
```

The function call `square(4)` is placed on the right side of assignment, so the return value is assigned to the variable `s`. The function call can be used in any expression also, here are some examples:

```
x = square(a) * 10 + b  
if 100 < square(x) < 500:  
    print('Do something')  
if add(a, b) > add(c, d):
```

```
    print('Do something')
p = add(square(a), square(b))
r = square(add(a, b))
print(square(a))
```

We can use the function call even as an argument to another function, as we have done in the last three examples. The return values of the function calls will be used as the argument. For example, in the call `add(square(a), square(b))`, the values returned by the calls `square(a)` and `square(b)` will be used as arguments to the `add` function.

We can write a function call in the return expression of another function.

```
def square(x):
    return x * x
def func(a, b):
    x = a + b
    return square(x)
z = func(3, 4)
print(z)
```

Output-

49

It is not compulsory to collect or use the return value of the function. We can simply ignore the return value and Python will not complain. The following code illustrates the same:

```
def result(marks1, marks2, marks3):
    total = marks1 + marks2 + marks3
    percentage = total / 3
    print(total, percentage)
    return 'Pass' if total > 100 else 'Fail'
r = result(88, 96, 46) # return value used
print(r)
result(78, 45, 89) # return value ignored
```

Output-

230 76.6

Pass

212 70.6

We can make functions that return Boolean values True and False. For example, we have these two Boolean functions `is_divisible` and `is_prime`.

```
def is_divisible(a, b):  
    if a % b == 0:  
        return True  
    else:  
        return False  
  
def is_prime(n):  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    else:  
        return True
```

This function `is_divisible` returns True if a is divisible by b, otherwise it returns False, and the function `is_prime` returns True if n is a prime number otherwise it returns False. We can easily use these types of Boolean functions in if statements and while loops.

```
if is_divisible(x, 3):  
    .....  
if not is_divisible(x, y):  
    .....  
if is_prime(x):  
    .....
```

```

if not is_prime(x):
    .....

if is_divisble(x, y) is equivalent to if is_divisible(x,
y) == True

if not is_divisible(x, y) is equivalent to if
is_divisble(x, y) == False.

```

The comparisons are redundant, we can write the function calls as the condition of the if statement.

The code of the function `is_divisible` can be further simplified.

```

def is_divisible(a, b):
    return a % b == 0

```

It returns the value of the comparison `a % b == 0` which will be either True or False.

Boolean functions can be used in while loop conditions also. For example, we have the following function `is_valid` which inputs age and returns True if the age is valid otherwise it returns False.

```

def is_valid(age):
    if not age.isdigit():
        return False
    age = int(age)
    if age <= 18 or age >= 75:
        return False
    return True

```

We can use it in while loop for input checking. The following loop will terminate only when `is_valid` returns True.

```

age = input('Enter age : ')
while not is_valid(age):
    age = input('Enter age : ')
print(f'Age is {age}')

```


We have seen that Python does not check the type of arguments sent to a function, it lets you pass any object as argument, so there are no types declared for parameters. Similarly, there is no type declared for return value, you can return any type of object from a function. So, you can return integers, strings, lists, dictionaries or any other Python type. This is because Python is a dynamically typed language.

Now, let us see what exactly happens when a value is returned from a function. We have written a function definition and a function call here:

```
def func(a, b):  
    s = a + b  
    return s  
p = func(5, 6)
```

When the function call executes, the code inside the function body will execute, an object with value 5+6 will be created and `s` will refer to it. The return value is specified as `s`. When the function finishes execution, name `s` will be destroyed as it is a local variable, but the object lives on. It is assigned to `p`. So, `p` will be assigned the object that is identified by name `s` in the function body.

Now suppose the function call is used in this expression:

```
y = func(2, 7) * 10
```

An object with value 9 is created, and `s` refers to it; when this function finishes, the name `s` is destroyed, but this object lives on and is used in the expression.

10.8 Returning Multiple Values

A function returns exactly one value and that value can be of any type; it can be an int, float, list, tuple or a dictionary. When you want to return multiple values from a function, you can pack those values into a single data structure like list, set, or tuple and then return that data structure from the function. So technically, you will be returning only one entity, but you will be able to return multiple values in that one entity.

For example, if you have to return 4 integers from a function, you can just pack them in a tuple and return that tuple. This way, we can indirectly return multiple values from a function. Let us see an example:

```
def func(a, b):  
    s = a + b  
    d = a - b  
    p = a * b  
    return (s, d, p)  
t = func(4, 5)  
sum, difference, product = func(4, 5)  
print(t, sum, difference, product)
```

Output-

```
(9, -1, 20) 9 -1 20
```

In the function body, we are returning sum, difference and product of the parameters in the form of a tuple. When we call the function, we can get the return value either in a tuple or we can unpack the returned tuple and assign the values to different variables as we have done in our example.

We know that a tuple can be created with or without the parentheses, so we can omit the parentheses in the return statement.

```
def func(a, b):  
    s = a + b  
    d = a - b  
    p = a * b  
    return s, d, p
```

These comma-separated values are automatically packed into a tuple. So, returning multiple values separated by commas is equivalent to returning a tuple. It looks like this function is returning 3 values, but in fact, it is returning just one value, which is a tuple without its optional parentheses.

Instead of storing the values of expressions inside the variables, we can return the expressions directly.

```
def func(a, b):
```

```
    return a + b, a - b, a * b
```

Let us see another example. Suppose we want to make a function that takes in a string as argument and returns the number of uppercase letters, number of lowercase letters and number of digits in that string. Here is the function.

```
def func(text):
    up = low = dig = 0
    for ch in text:
        if ch.isupper():
            up += 1
        elif ch.islower():
            low += 1
        elif ch.isdigit():
            dig += 1
    return up, low, dig
uppers, lowers, digits = func('Fall down 7 times,
Stand up 8')
print(f'Uppercase : {uppers}, Lowercase : {lowers},
Digits : {digits}')
t = func('Fall down 7 times, Stand up 8')
print(t)
```

Output-

```
Uppercase : 2, Lowercase : 18, Digits : 2
(2, 18, 2)
```

Inside the function definition, we have taken three variables, all initialized to 0. Then, we iterate over the string named `text` and store the number of uppercase letters, lowercase letters and digits in the 3 variables. After this, we have returned the values of the variables. These values are packed inside a tuple, and that tuple is returned. When we called the function, we unpacked the tuple and assigned the values to three variables, and after that, we printed those variables. When we assigned the result to a single variable, that variable was a tuple.

Let us see one more function that returns multiple values:

```
def max_min_avg(L):  
    return max(L), min(L), sum(L) / len(L)  
marks = [92, 76, 98, 67, 88, 92, 89]  
maxmarks, minmarks, avgmarks = max_min_avg(marks)  
annual_rain = [11, 2, 23, 11, 9, 2, 1, 23, 13, 3,  
12, 20]  
max_rain, min_rain, avg_rain =  
max_min_avg(annual_rain)  
print(maxmarks, minmarks, avgmarks)  
print(max_rain, min_rain, avg_rain)
```

The function takes in a list and returns the maximum, minimum, and average of the elements of that list. Inside the function definition we have called the built-in functions `max`, `min`, `sum` and `len`. We have two lists named `marks` and `annual_rain`, and we have called the function two times, with these lists as argument. In both the cases we are storing the return value in three separate variables and then we are printing those variables.

This function `max_min_avg` can accept a tuple, or a set or a dictionary also as an argument, as the built-in functions `max`, `min` and `sum` can work on these types also.

Some built-in functions also return multiple values in the form of tuple. `divmod` is one such function.

```
q, r = divmod(11, 3)
```

`divmod` function returns a tuple whose first element is the quotient and the second element is the remainder. We can assign the return value to 2 variables.

10.9 Semantics of argument passing

10.9.1 Why study argument passing

We know that the arguments that we write in a function call could be in the form of a literal, expression or a variable. When we pass literals or expressions, we do not have to worry about our data being changed by the function. But when we pass variables to a function, the question arises whether any changes made to the parameters will have an effect on the corresponding variables sent as arguments.

```
def func(x, y):  
    .....  
    .....  
a = 4  
my_list = [3, 4]  
func(2, [6, 7])    # literals as arguments  
func(a, my_list)   # variables as arguments  
print(a, my_list)
```

In the first call, we have literals as arguments, so there is no problem. In the second function call, we are sending two variables from our program to the function. These variables will exist in the program even after the function call is over. It is important for us to know whether the function can have an effect on these passed variables. For example, if we increase parameter x within the function, will the variable a also be increased.

The function could have unexpected side effects if it changes the variable that you pass to it. Sometimes these side effects would be intended, in other cases, these side effects are unwanted and can cause hidden bugs in your program if you are not aware and careful. Let us see some code snippets and their output to understand how the change in parameters affects our variables sent as arguments.

<pre>def func1(x): x += 10 n = 5 func1(n) print(n) Output- 5</pre>	<pre>def func2(x): x *= 2 L = [6, 7] func2(L) print(L) Output- [6, 7, 6, 7]</pre>	<pre>def func3(x): x = [1, 2, 3] L = [6, 7, 8] func3(L) print(L) Output- [6, 7, 8]</pre>	<pre>def func4(x): x.add(9) s = {7, 8} func4(s) print(s) Output- {8, 9, 7}</pre>
<pre>def func5(x): x = x.upper() s = 'hello' func5(s) print(s) Output- hello</pre>	<pre>def func6(x): x.pop(2) d = {2: 4, 3: 9} func6(d) print(d) Output- {3: 9}</pre>	<pre>def func7(x): x.clear() L = [6, 7] func7(L) print(L) Output- []</pre>	<pre>def func8(x): x = [] L = [6, 7] func8(L) print(L) Output- [6, 7]</pre>

Figure 10.3: Argument passing

In functions func1, func3, func5 and func8, the changes in the parameter had no effect on the variable sent as argument while in the remaining functions the change in parameter was reflected in the caller's variable sent as argument.

To know whether an argument can be changed inside a called function, we have to understand the mechanism of argument passing. Different languages pass arguments to functions in different ways, Python's approach is different from most of the other languages. To understand the argument passing mechanism in Python, you need to be clear about what assignment and in-place changes mean in Python, what are objects and references, mutable and immutable types.

10.9.2 Pass by assignment

We have seen that when a function call executes, the arguments are assigned to the corresponding parameter names. This assignment happens implicitly, before the function body executes, so we can say arguments are passed by assignment. Assignment in Python means object reference (or binding a name to an object), so we can also say that arguments are passed by object reference. Therefore, arguments passing mechanism in Python is called pass by assignment or pass by object reference.

Let us see this whole mechanism in detail. First let us see what happens when we send literals as arguments.

```
def func(x, y):  
    print(x, y)  
func(3, [1, 2, 3])
```

When the function call `func(3, [1, 2, 3])` is executed, Python sees an `int` literal `3` and a `list` literal `[1, 2, 3]` so it creates an `int` object and a `list` object for these literals. Before the function body starts executing, the `int` object is assigned to parameter `x` and the `list` object is assigned to parameter `y`. This implicit assignment is equivalent to:

```
x = 3  
y = [1, 2, 3]
```

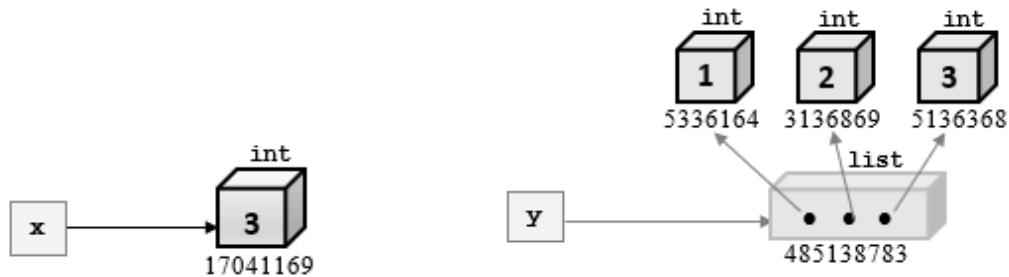


Figure 10.4: *x* refers to `int` object and *y* refers to `list` object

`x` is bound to the `int` object, and `y` is bound to the `list` object. After the function body finishes executing, the parameter names `x` and `y` are destroyed. The objects will be garbage collected since there is nothing referring to them now.

Let us see what happens when we send variables as arguments.

```
def func(x, y):  
    print(x, y)  
num = 2  
my_list = [1, 2, 3]  
func(num, my_list)
```

The assignment statements `num = 2` and `my_list = [1, 2, 3]` execute and so the variable `num` refers to an `int` object with value 2 and the variable `my_list` refers to a list object.



Figure 10.5: `num` refers to `int` object and `my_list` refers to `list` object

When the function is called, two local variables `x` and `y` are created, and implicit assignment is done which is equivalent to:

```
x = num
y = my_list
```

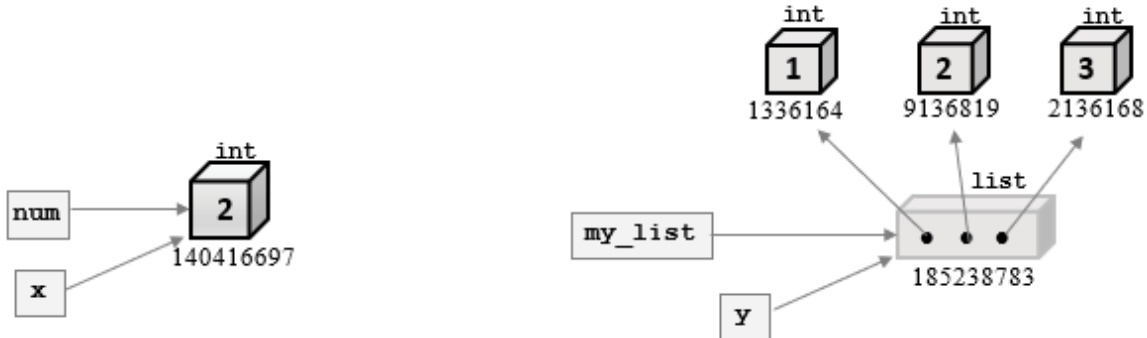


Figure 10.6: `x` and `num` refer to the same `int` object, `y` and `my_list` refer to the same `list` object

Now `x` refers to the same object to which `num` is referring and `y` refers to the same object to which `my_list` is referring. So, inside the function, the `int` object with value 2 is identified (referenced) using the name `x` and the `list` object is referenced using the name `y`. The parameter name becomes an alias for the corresponding argument variable; both refer to the same object. You can verify this by printing the ids of `x`, `y`, `num` and `my_list`. The ids of `x` and `num` will be same and ids of `y` and `my_list` will be same.

When the function call is over, the names `x` and `y` are destroyed, but the objects will still be there because they are referenced by the names `num` and `my_list`.

In Python, every variable is just a reference to an object that contains the actual data. The variable does not store the data directly, it only has information about where the object that contains the data is located in memory. You can think that a variable just contains the location of the object and it is this location that is passed to the function. The parameter name gets this location and so it also starts referring to the same object. So, we can see that the object is not passed, no copy of the object(data) is made, instead only reference to the object is passed (location of object is copied). This is why the mechanism is named pass by object reference (or call by object reference). The same object is shared by both the argument variable and the parameter and so the mechanism is also sometimes called call by sharing.

10.9.3 Assignment inside function rebounds the parameter name

When a variable is sent as argument, initially the argument and parameter share the object, i.e. they refer to the same object, but as soon as the parameter name is reassigned, this sharing ends. So, if we assign to a parameter name inside the function, it is rebound which means that it starts referring to some other object and the connection to the original object is lost. Let us understand this with the help of an example:

```
def func(x, y):  
    print(f'x : {x}, y : {y}')
```



```
    x = 0  
    y = []  
    print(f'x : {x}, y : {y}')
```



```
num = 2  
my_list = [1, 2, 3]  
func(num, my_list)
```

```
print(f'num : {num}, my_list : {my_list}')
```

Output-

```
x : 2, y : [1, 2, 3]
```

```
x : 0, y : []
```

```
num : 2, my_list : [1, 2, 3]
```

Before the function call, we have two variables `num` and `my_list`.



Figure 10.7: *num* refers to an *int* object and *my_list* refers to a *list* object

When the function is called, the parameters names `x` and `y` also refer to the same objects to which `num` and `my_list` are referring.



Figure 10.8: *x* and *num* refer to the same *int* object, *y* and *my_list* refer to the same *list* object

Inside the function, when the two assignment statements execute, the names `x` and `y` are rebound. `x` now refers to an *int* object with value zero and `y` now refers to a *list* object that represents an empty list.

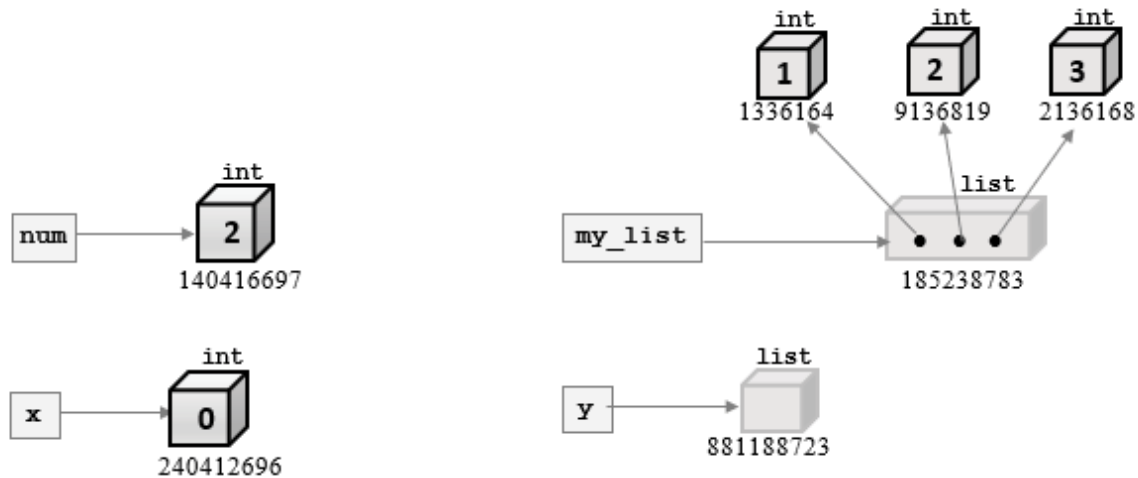


Figure 10.9: Assignment inside the function changes the binding of the parameter

So, we can see that when we assign to a parameter name inside a function, the binding of the parameter name is changed, it is rebound to some other object. It no longer refers to the object to which the argument was referring. After the call is over, the names `x` and `y` are destroyed. The variable `num` is still bound to object 2 and `my_list` is still bound to `[1, 2, 3]`.

Rebinding `x` does not affect the binding of `num`, which was used as the argument. It is still bound to the same object as before the call. Similarly, rebinding `y` has no effect on the binding of `my_list`.

The function cannot rebind the caller's variables which it receives as arguments. The function gets a copy of the reference (location of the object), so it gets access to the referred object, but it has no control over the caller's variable; it cannot change what the caller's variable is referring to. For example, here in our code, the function receives `num` as argument so it gets access to the `int` object, but it cannot change what `num` refers to.

Since the function gets access to the referred object, if your object happens to be mutable, it can be changed inside the function, and the changes will be visible outside the function, too, because the object still lives on. We will cover this in the next section.

The conclusion is that a function cannot change the value of an argument variable by reassigning the corresponding parameter to something else. Inside the function body, if a parameter is reassigned a new value, then it

does not alter the argument, this reassignment simply changes the binding of the parameter.

10.9.4 Immutables vs Mutables as arguments

We know that the function gets access to the object through the reference that is passed; if the referred object is mutable, then the function can make in-place changes in it which will be visible to the caller.

```
def func(y):  
    y.append(4)  
my_list = [1, 2, 3]  
func(my_list)  
print(my_list))
```

Output-

```
[1, 2, 3, 4]
```

The function got access to the list object, and the `append` method inside `func` mutated the referred object. Since both `y` and `my_list` are bound to the same object, the changes are visible through the name `y` also. We can clearly see that the function has affected the caller's variable, since before the function call, `my_list` was `[1, 2, 3]`, and after the function call, it is `[1, 2, 3, 4]`.

Whenever we try to change a variable that is bound to an immutable type, we have to do an assignment, and we know that assignment rebinds the name. This is why when an argument variable refers to an immutable type, any changes in the parameter are not reflected in the argument. Here is an example:

```
def func(x):  
    x *= 2    # Rebinding  
num = 10  
func(num)  
print(num)
```

Output-

10

The fact that the function can change mutable objects through the parameters can be used in situations when we want a function to manipulate our data in some way. For example, the following function `double` doubles the value of each element of list:

```
def double(data):  
    for i in range(len(data)):  
        data[i] *= 2  
x = [1, 2, 3, 4]  
double(x)  
print(x)
```

Output-

```
[2, 4, 6, 8]
```

Inside the function, each element of the list argument is multiplied by 2. So, in this function the list is acting both as the input and the output. When we print the list `x` after the function call, we can see that the elements of the list have doubled.

Therefore, mutable arguments can be used as both input and output for a function. Here is another example:

```
data = {}  
def enter_data(d):  
    while True:  
        id = input('Enter id (0 to quit): ')  
        if id == '0':  
            break  
        name = input('Enter name : ')  
        d[id] = name  
enter_data(data)  
print(data)
```

Here, we are sending a dictionary to the function `enter_data` and the dictionary gets filled with data inside the function. The dictionary is acting both as the input and output.

The conclusion is that when a variable bound to an immutable object is passed as argument, the changes do not propagate to the caller in any way. When a variable to a mutable object is passed as argument, the changes can propagate to the caller if the object can be changed in-place. The following figure summarises the same:

Argument bound to mutable object	Argument bound to immutable object
<u>Assignment to parameter</u> <pre>def func(L): L = [4,5,6] my_list = [1,2,3] func(my_list)</pre> No effect on argument variable	<u>Assignment to parameter</u> <pre>def func(num): num = 10 x = 5 func(x)</pre> No effect on argument variable
<u>Shared object changed in-place via parameter name</u> <pre>def func(L): L.append(9) L.remove(5) my_list = [1,2,3] func(my_list)</pre> Changes visible in argument variable	<u>In-place changes not possible in objects of immutable types</u> X

Figure 10.10: Argument passing

Now, after this whole discussion let us try to understand the all the code snippets that we saw in [Section 10.9.1](#). In functions `func1`, `func3`, `func5`, `func8`, the parameter is rebound by an assignment.

In the call to function `func2`, the argument is a list which is a mutable object, and we have seen that for a list the augmented assignment syntax makes in-place changes. So, in-place changes are being made to the parameter inside the function and that is why we can see the change in argument.

Similarly, in functions `func4`, `func6`, and `func7`, in-place changes are made to the parameter, so we can see the change in the argument.

10.9.5 How to get the changed value of an immutable type

We have seen that if a variable bound to an immutable object is sent as an argument, then the called function cannot change it. But there might be some situations when we want to change such an argument. We can do this by returning and reassigning.

```
def triple(x):  
    x *= 3  
    return x  
  
num = 4  
num = triple(num)  
print(num)
```

Output-

12

We returned the changed value and assigned the returned value to our original variable. This is what we did in strings, since string is an immutable type.

```
s = 'hello'  
s = s.upper()
```

The method `upper()` returns the changed string which we assigned to the original variable.

If we want to change multiple arguments that refer to immutable types, we can easily do as we know that a function can return multiple values. The values returned can be assigned to the original variables. Here is an example:

```
def func(x, y, z):  
    x = 100 - x  
    y *= 2  
    z += 5  
    return x, y, z  
  
n1 = 2
```

```
n2 = 3
n3 = 4
n1, n2, n3 = func(n1, n2, n3)
print(n1, n2, n3)
```

Output-

```
98 6 9
```

10.9.6 How to prevent change in mutable types

We have seen that if we pass an argument that refers to a mutable object, then that function can change the argument. Sometimes we may not want a function to change our original argument. To make sure that our argument is not changed by the called function, we can pass an explicit copy of the object. Let us understand this with the help of an example-

```
def sum_of_squares(L):
    s = 0
    for n in L:
        s += n * n
    return s

numbers = [1, 2, 3, 5, 6]
s = sum_of_squares(numbers)
print('Sum of squares =', s)
print(numbers)
```

Output-

```
Sum of squares = 75
[1, 2, 3, 5, 6]
```

The function returns the sum of squares of a list. It just iterates over the list and keeps on adding squares of numbers in the variable `s` and then it returns `s`. There are no in-place changes made to the list so our variable `numbers` is not changed by the function. The list remained unchanged after the function call and this is normal. We would not expect such a function to

change our list. Now suppose that the writer of the function had used a different logic and had written the function in this way.

```
def sum_of_squares(L):  
    for i in range(len(L)):  
        L[i] *= L[i]  
    return sum(L)  
  
s = sum_of_squares(numbers)  
print('Sum of squares =', s)  
print(numbers)
```

Output-

```
Sum of squares = 75  
[1, 4, 9, 25, 36]
```

In this function, each element of the list is squared and then the function `sum` is called on the list. The function also works perfectly, it returns 75 like the previous function. But this function will change the original list, because in-place changes are being done to the parameter. If we want our list to remain safe, then instead of sending the list, we can send a copy of the list.

```
s = sum_of_squares(numbers[:])
```

Now when we execute our program, we will see that the original list has not changed. This is because now the argument is not a variable, it is an expression that represents a list object. Python will create a new list object by copying the list object referred to by list `numbers`, and the parameter will refer to this new copied list object. This way the function does not get access to the original object. Any in-place modifications made within the function will be applied to the copy, leaving the original object unaffected.

So, whenever we are sending a mutable argument and want to be sure that it is not changed in any way inside the function, we can send a copy of the argument. This will prevent any change to our argument, and so we can safely send it to any function. If we want to send a copy of a dictionary, we can use the dictionary `copy` method.

Passing a separate copy involves time and memory overhead, especially when working with large objects, so you should pass a copy only when really required.

If our argument refers to an immutable object, then we need not worry about our argument being changed by the function, as the function cannot modify the object in any way.

10.9.7 Digression for programmers from other languages

In most of the languages there are two ways of passing arguments:

Pass by value (Call by value): A copy of the variable is passed; the function works on the copy so it cannot change the caller's original variable. Any changes made to the parameter will not have effect on the corresponding argument variable.

Pass by reference (Call by reference): Reference or memory address of the variable is passed; the function gets access to the original variable and works on it. Changes made to the parameter will be reflected in the corresponding argument variable.

As we have seen, Python uses pass by assignments or pass by object references. This mechanism is neither pass by value nor pass by reference exactly. This difference is there because the concept of variables is different in Python and other languages like C or C++. In these languages, data is not an object, it is stored in variables. You can think of a variable as a box that stores the data; a box that has specific location. When we talk of reference in these languages, we are talking about the memory address of the variable (location of the box). In Python, data is stored in objects and variables are just names that refer to those objects. In Python, when we talk about reference, we mean location of an object.

In other languages, we pass references to variables, in Python we pass references to objects.

Let us see the scenarios when the Python's mechanism behaves like call by value and when it behaves like call by reference.

When you pass variables referring to immutable types like integers, strings or tuples to a function, the behaviour is like call-by-value. Function cannot modify the caller's variable. (No copy of the data is made, but you get the effect of call by value without actually copying the data).

When you pass variables referring to mutable type, initially the behaviour is call-by-reference, but as soon as you assign to the parameter name, the behaviour changes to call-by-value.

10.9.8 Advantages of Python's information passing

Like argument passing, returning a value from a function also follows the semantics of assignment statement. The benefit of following the semantics of assignment for passing and returning values from functions is that the objects need not be copied. This makes function calls efficient even when the arguments or return values are big and complex objects. When the objects to be passed or returned are large, copying can be costly, this approach makes the information passing efficient.

10.10 Default Arguments

If there is some argument value that would be used most of the times while calling the function, we can make it a default value for the parameter. This default value will automatically be used as the argument, if the user of the function does not provide the corresponding argument for that parameter. These default values are called default arguments. These values are provided in the header of the function definition, let us see the syntax of specifying a default argument with the help of a simple example:

```
def func(a, b=5):  
    print(a, b)
```

In this function definition we have two parameters, a and b. For the parameter b, we have assigned a default argument value 5. The default argument value is placed after the parameter name with the equal to sign (=)

in between. Now, we can call this function with either two arguments or one argument.

```
func(10, 2)
```

```
func(10)
```

In the call `func(10, 2)`, parameter `a` is assigned value 10 and `b` is assigned value 2. In the call `func(10)`, `a` is assigned value 10; there is no argument value for parameter `b` so the default argument value is used and `b` is assigned value 5.

So, if a parameter is given a default value, then providing argument for it becomes optional. If you provide the argument, then its value will be used, otherwise the default value will be used. In this way you can make some parameters optional in a function. Here is one more example-

```
def simple_interest(principal, time, rate=5):  
    return (principal * rate * time) / 100  
  
s1 = simple_interest(1000, 4, 7)  
s2 = simple_interest(1000, 4)
```

In this function, we are calculating and returning simple interest. We have three parameters from which `rate` is an optional parameter because we have provided a default argument for it. The other two are the required parameters, meaning we have to provide arguments for these parameters during the function call. We have written two function calls; in the first one we have supplied a value for `rate`. In the second call, we have not supplied any value for `rate`, so 5 will be used.

We specify default arguments when there are certain parameters in a function for which a common value is used most of the times. For example, in our `simple_interest` function, the value of `rate` is 5 in most of the calls and so we specified it as the default argument. Thus, the argument value that occurs frequently is specified as the default argument. There is no need of writing the argument on each invocation.

By providing default arguments, we can provide a general functionality to the user and if he wants some special behaviour, he can supply his own arguments.

More than one parameter can have default arguments, for example in our `simple_interest` function, we can provide a default argument for parameter `time` also.

```
def simple_interest(principal, time=2, rate=5):  
    return (principal * rate * time) / 100  
  
s1 = simple_interest(1000, 4, 7)  
s2 = simple_interest(1000, 4)  
s3 = simple_interest(1000)
```

Now we can call the function with three, two or only one argument. In the last call, we have provided the argument only for `principal`, and the default values for `time` and `rate` will be used.

If a parameter has a default argument, then all the parameters following it should also have default arguments. Therefore, parameters that have to be given default value should be placed at the last.

The following definition is wrong because we have a parameter with a default value and after that we have a parameter without default value.

```
def simple_interest(principal, time=2, rate):          #  
wrong  
    return (principal * rate * time) / 100
```

Similarly, the following definition is also wrong.

```
def simple_interest(time=2, rate=5, principal):  
#wrong  
    return (principal * rate * time) / 100
```

All optional parameters must come after the required parameters.

Let us see with the help of some example how the default arguments make our function more flexible and versatile.

```
def display_line():  
    print('-' * 30)  
display_line()
```

We have a simple function that displays a line on the screen. When this function is called, we get a line of 30 dashes. This function can be made more flexible by providing optional parameters.

```
def display_line(character='-') :  
    print(character * 30)
```

We introduced a parameter named `character` and provided a default value for this parameter. Inside the definition, instead of `'-'` we write the parameter name. Now by using this function, we can draw a line of any character.

```
display_line('%')  
display_line('*')  
display_line()
```

We can still call it without any argument, and in that case, it prints a line of dashes. Now the user can draw a line made up of any character, but the length of the line is fixed, it is always 30 characters long. We will make the length also flexible by introducing another optional parameter.

```
def display_line(character='-', length=30):  
    print(character * length)  
display_line('%', 20)  
display_line('*', 50)  
display_line('*')  
display_line()
```

If we don't provide the value of `length`, the default value of 30 is used. And if we don't provide any argument then the default arguments for both the parameters are used. So now the user can draw a line of his own choice and if he just writes `display_line()`, the function will draw a line of 30 dashes.

This is an example of polymorphism, which means one thing many forms. A single function can be called in different ways. This example also shows how our functions can evolve with time while remaining backward compatible.

In our next example we will provide a default argument in the `display_list` function that we had written earlier. In this function, most of the times numbers will be printed in decimal base so we can make 10 as the default argument value for base.

```
numbers = [134, 2567, 366, 521, 689]
def display_list(L, base=10):
    if base == 2:
        ch = 'b'
    elif base == 8:
        ch = 'o'
    elif base == 16:
        ch = 'X'
    else:
        ch = 'd'
    for n in L:
        print(f'{n:{ch}}', end=' ')
    print()
display_list(numbers)
display_list(numbers, 8)
display_list(numbers, 16)
```

When the base is not specified, numbers are displayed in decimal.

Sometimes a parameter value is not required or not applicable in some cases. In these situations, you can make that parameter optional by providing a default value. For example, in the following function, it is optional to provide a sports grade or an arts grade.

```
def result(total, sports=None, arts=None):
    print('Total marks = ', total)
    if sports is not None:
        print('Sports Grade', sports)
    if arts is not None:
        print('Arts Grade', arts)
```

```
result(98, 'A')
result(78, 'B', 'C')
result(88)
```

10.11 Default arguments that could change over time

If you use a default argument that could change over time, then it could lead to unexpected behaviour. Let us understand this with the help of some examples.

```
d = 6
def func(p = d):
    print(p)

func()      # default argument used for parameter p
func(80)    # 80 used as argument for parameter p
d = 100
func()      # default argument used for parameter p
d += 20
func()      # default argument used for parameter p
```

We have a small function definition which has an optional parameter. The variable `d` is used as the default argument for the parameter `p`. You would expect that whenever the function is called without any argument, the parameter `p` should be initialized with the value of variable `d`. This is the output that you would expect from this code.

```
6  80  100  120
```

The first call has no argument and so the parameter `p` should be initialized with the value of `d`, thus `p` should be 6. In the second call, argument is provided, so `p` is 80. In the third call, default argument will be used so `p` should be 100, because `d` is now 100. Similarly in the last call you expect `p` to be 120, because `d` is now 120. This is the behaviour that you expect from this program, but when you run it, it gives the following output.

```
6  80  6  6
```


The reason for this unexpected output is that the default argument gets evaluated only once when the function definition (`def` statement) executes. That same object is bound to the parameter, each time the function needs to use default argument. The default argument is not re-evaluated each time the function is called(function call is executed).

In our example program, when the `def` statement executed, the parameter `p` was bound to integer object 6 because the variable `d` was referring to it. This object was fixed as the default argument for this parameter and so was used in all those calls that did not supply the corresponding argument. The variable `d` was bound to other objects over time but `p` was always bound to the original object.

Let us take one more example to see how we can write problematic functions if we are not aware of this issue. Suppose we write a simple function to log some information along with the time when the information is logged. It has two arguments and we have provided default value for both of them.

```
from time import sleep
from datetime import datetime
def log(information='Everything Ok...',
time=datetime.now()):
    print(information, time)
log('Some problem...', '16:59:49')
log()
sleep(2)
log('Another Problem...')
sleep(3)
log()
```

We want the caller of the function to specify `information` and `time`. If no argument is provided for the parameter `information` then the string 'Everything Ok...' will be used and if value for parameter `time` is not provided then value returned by the method `datetime.now` will be used. This is the sort of output that we expect from this code.

```
Some problem... 16:59:49
```

```
Everything Ok... 2023-06-29 12:15:03.830804
Another Problem... 2023-06-29 12:15:05.833038
Everything Ok... 2023-06-29 12:15:08.833439
```

In the first call, we have provided arguments for both parameters, so those values are used. In the second and fourth calls we have not provided any argument so the default values will be used for both parameters. In the third call, default value will be used for the second parameter. We expect the current date and time returned by the method `datetime.now` to be used as the default argument, when the second argument is missing in the call. The actual output that we get by executing the above code is this-

```
Some problem... 16:59:49
Everything Ok... 2023-06-29 12:16:02.296959
Another Problem... 2023-06-29 12:16:02.296959
Everything Ok... 2023-06-29 12:16:02.296959
```

Each time the function is called without the second argument, the same time is printed. This time that we are getting in all the calls, is actually the time when the `def` statement was executed. When the function definition was executed, the method `datetime.now` got executed and the object returned by it was fixed as the default argument for parameter `time`. This is why whenever we don't provide the second argument, we see the same time getting printed. This is not the kind of behaviour that we expect from our function.

Similar type of problem shows up when a mutable object is specified as the default argument and the function modifies the parameter. Let us see this with the help of a very simple example-

```
def func(a, L=[]):
    L.append(a)
    print(L)
func(10, [1, 2, 3])
func(8, [5, 6])
func(9)
func('Hello')
```

```
func(100)
```

We have a function with two parameters, the second one has an empty list as the default argument. The expectation is that if the second argument is omitted when this function is called, a fresh empty list object will be created and bound to the parameter name. This is the output that we expect.

```
[1, 2, 3, 10]
```

```
[5, 6, 8]
```

```
[9]
```

```
['Hello']
```

```
[100]
```

In the first call, 10 should be appended to the list [1, 2, 3] and printed. In the second call, 8 should be appended to the list [5, 6] and printed. In the last 3 calls, the second argument is not provided so an empty list should be used as the second argument, and the first argument should be appended to that empty list. The output is again different from what is expected.

```
[1, 2, 3, 10]
```

```
[5, 6, 8]
```

```
[9]
```

```
[9, 'Hello']
```

```
[9, 'Hello', 100]
```

When the function definition was executed, an empty list object was created and the same single list object is being used as the default every time the function is called. The function is modifying the list object which is sent as argument. When no argument is sent, the function uses the default argument and modifies it. The same modified list object is used another time when the function is called. As a result, the parameter is not initialized with proper default value each time it is called.

So, we saw three examples where the expected output and the real output were different. And it was because the default argument that we used could change over time. Thus, it is a bad idea to use default argument values that could change over time. When you use a literal of immutable type (eg. 23, 'hello', (2,3,4)) as a default argument then there is no problem because the objects of immutable types cannot be changed. When you specify a literal of

mutable type(eg. [1,2], {}) then problems could arise as the mutable object could be changed inside the function or the function might return it and it could be changed even outside the function. The point is that setting a mutable object as default argument is dangerous as it could change. And so, there is no guarantee that each subsequent call will get the default initial value. If we use a variable as a default argument(as in our first example) then also we don't get what we expect, as the variable can be reassigned.

Now we will see what can we do to get the expected output even if we use something that can change over time as default argument. The common idiom is to use `None` as the default and then test for it explicitly inside the function body. The actual default is assigned to parameter inside the function body if the parameter is `None`. Let us see how we can use this trick in the examples that we have seen.

```
# old function
def func(a, L=[]):
    L.append(a)
    print(L)

# modified function
def func(a, L=None):
    if L is None:
        L = []
    L.append(a)
    print(L)
```

We have set the default to `None`, so parameter `L` is optional. If a list is supplied by the caller, it will be used inside the function and if it is not supplied then `L` will be `None`, and inside the function `L` will refer to a newly created empty list object. Now every time the function is called, a fresh empty list object is created. The default is not shared between subsequent calls. So, if you want your parameter to be bound to a fresh mutable object each time a function is called, you can use this trick.

We can use the same trick in the `log` function.

```
#old function
```

```
def log(information='Everything Ok...',
time=datetime.now()):
    print(information, time)
#modified function
def log(information='Everything Ok...', time=None):
    if time is None:
        time = datetime.now()
    print(information, time)
```

In this function if second argument is not provided then `time` will be `None`, the if condition will be `True` and `time` will be assigned the return value of the method `datetime.now()`. Now if you run your previous code with this modified version of the function, you will get the expected output; the `time` will be different in each function call.

Similarly, we can use this idiom in the first example that we saw.

```
#old function
def func(p = d):
    print(p)
#modified function
def func(p=None):
    if p is None:
        p = d
    print(p, end = '  ')
```

10.12 Positional and Keyword Arguments

In this section, we will see an alternative way of supplying arguments in a function call. We have seen that the argument values that we supply in the function call are matched to parameter names by position, from left to right. For example, in the following code, the first argument is matched to first parameter, second argument is matched to second parameter and third argument is matched to third parameter.

```
def func(name, title, salary):
```

```
    print(f'{name} is a {title} and gets  
{salary}')
```

```
func('Nick', 'manager', 5000)
```

The interpreter matches the arguments based on their position. Arguments matched by their position are called positional arguments. In Python, there is an alternative way to specify arguments in the function call. Here is a function call that uses the alternative way.

```
func(name='Nick', title='manager', salary=5000)
```

In this syntax, we write the parameter name, then equal sign, and then the argument value that we want to assign to the parameter. We clearly specify which value is for which parameter and so now matching of argument and parameter is explicit. If we specify a name in the call that does not match any of parameters in the definition then there will be an error.

When we specify parameter names in the call, the arguments are identified by the parameter name and not by position, so the order of arguments does not matter. We can write the function call with different orders of parameter=argument pairs.

```
func(title='manager', salary=5000, name='Nick')
```

```
func(title='manager', name='Nick', salary=5000)
```

These two calls are equivalent to the previous one and all will give the same output.

The arguments that are matched by position are called positional arguments and the arguments that are matched by parameter name are called keyword arguments (or named arguments). Thus, a positional argument is an argument that is assigned to a parameter based on its position in the argument list while a keyword argument is assigned to a parameter based on the parameter name specified along with the argument.

You can mix positional and keyword arguments in a single call. When you do this, all the positional arguments have to appear before the keyword arguments.

```
func('Nick', salary=5000, title='manager')
```

```
func('Nick', 'manager', salary=5000)
```

The following call will give error as we have used the keyword argument before the positional arguments.

```
func(name='Nick', 'manager', 5000)
```

Both positional and keyword arguments can be used for overriding default values. Let us modify our function definition so that it has some parameters with default values.

```
def func(name, title='developer', salary=3000):  
    print(f'{name} is a {title} and gets {salary}')
```

In this definition, `name` is a required parameter while `title` and `salary` are optional parameters. If you decide to override the default value and supply your own value, you can use any type of argument, positional or keyword argument. First let us use positional arguments.

```
func('Mark')  
func('Mark', 'programmer')  
func('Mark', 'programmer', 4000)
```

Output-

```
Mark is a developer and gets 3000  
Mark is a programmer and gets 3000  
Mark is a programmer and gets 4000
```

In the first call we supplied one positional argument, so default values for both `title` and `salary` are used. In the second call, default value for only `salary` is used, and in the third call, both default values are overridden.

Now we will write the equivalent calls using the keyword arguments.

```
func(name='Mark')  
func(name='Mark', title='programmer')  
func(name='Mark', title='programmer', salary=4000)
```

In the first call, both default values are used, in the second call, only one default value is used and in the third one, none of the default value is used.

Now that we have learned how to provide keyword arguments, a question arises: What are the benefits of using these keyword arguments instead of

always sending arguments by position? Let us explore some advantages of using keyword arguments.

In some places, using explicit parameter names in the call makes the code easier to read.

```
volume_cylinder(10, 12)
calc_interest(20000, 5, 6):
volume_cylinder(radius=10, height=12)
calc_interest(20000, time=5, rate=6):
```

We have two function calls with positional arguments and two equivalent calls that use keyword arguments. In the first call `volume_cylinder(10, 12)`, it is not clear which argument is for `radius` and which is for `height`. In the call `volume_cylinder(radius=10, height=12)`, there is no confusion, the meaning of arguments is obvious.

Similarly, the two calls to the function `calc_interest` will perform the same work, but for someone reading the code, the second one is clearer. So, keyword arguments provide a way to improve readability of the code, wherever it is important, especially in larger programs. The code becomes self-documenting, provided that the parameter names are meaningful. Single letter names like `x`, `y`, `a` will not help in making the call informative.

Another benefit of using keyword arguments is that the caller has the flexibility to specify the arguments in any desired order. With positional arguments the caller needs to remember the meaning of each position, but with keyword arguments you don't have to worry about the order in which the arguments have to be supplied. It is not necessary to specify the arguments in the same order in which they are specified inside the function definition. For example, you can call your `calc_interest` function in any of the following two ways.

```
calc_interest(20000, time=5, rate=6)
calc_interest(20000, rate=6, time=5)
```

To understand the third advantage let us consider the following function again.

```
def func(name, title='developer', salary=3000):
```



```
print(f'{name} is a {title} and gets  
{salary}')
```

In this function definition, we have default arguments for the last two parameters. Now suppose when you call this function, you want to supply your own value for `salary` but want to use the default value for `title`. With positional arguments it is not possible; there is no way to do this. You can't write your call like this-

```
func('Nick', 6000)
```

If you do this, you get this output.

```
Nick is a 6000 and gets 3000
```

Argument 6000 is assigned to `title` and default value of `salary` is used.

So, if you are using positional arguments, you have to write your call like this -

```
func('Nick', 'developer', 6000)
```

Even though you want to use the default value for `title`, you have to specify it because you have to override the default value for `salary`. If you use keyword arguments you can write your call like this.

```
func('Nick', salary=6000)
```

Now there is no need to specify the value for `title` parameter.

Let us look at one more example that will help us gain a clearer understanding of this point.

```
def func(a, b, c=2, d=5, e=8, f=True, g=False):  
    pass
```

In this function, the last 5 parameters have default values, so they are optional. When we call the function, we want to supply our own values for parameters `d` and `g`, and use the defaults for the rest of the optional parameters. With positional arguments, we are forced to write our call like this-

```
func(12, 24, 2, 500, 8, True, True)
```

We have to specify all the arguments, even though parameters `c`, `e` and `f` need just the default value. When you are using positional arguments, and

you want to override a default value, you have to override all defaults before that. But if you use keyword arguments, you can skip over parameters with defaults. We can write this call using keyword arguments like this-

```
func(12, 24, d=500, g=True)
```

Now we can skip the parameters `c`, `e` and `f` and specify only those parameters for which we want to override default value.

So, the third advantage of using keyword arguments is that you can bind some optional parameters to specific values and let other parameters take default values.

Let us see some example programs.

```
def volume_cylinder(radius, height):  
    return 3.14 * radius * radius * height  
print(volume_cylinder(5, 20))  
print(volume_cylinder(radius=5, height=20))  
print(volume_cylinder(height=20, radius=5))
```

We have this function that takes in radius and height of a cylinder and returns its volume. First, we have called it with positional arguments, then we have called it with keyword arguments. We can reverse the order of the arguments when we use keyword arguments.

Now let us take a function that has default arguments.

```
def display_line(character='-', length=30):  
    print(character * length)
```

We have seen this function before. If we want to use default for `character` and specify our own argument for `length`, we can't do that using positional arguments. If we try to do that, we get incorrect results.

```
display_line(40)
```

Output-

```
1200
```

We wanted to display a line of 40 dashes, but we got 1200 as result. It is because 40 was assigned to `character` and default value of `length` was

used. So, inside the function `40 * 30` was printed. We have to specify the dash in the call.

```
display_line('-', 40)
```

If we use keyword argument, then there is no need to specify the first argument.

```
display_line(length=40)
```

Now default is used for character and we have provided our own value for length.

Here is one more example-

```
numbers = [12, -1, 3, 6, 8, 9, 38, -3, 34, -4]
```

```
def display(L, negative=True, odd=True):  
    for n in L:  
        if n < 0 and negative == False:  
            continue  
        if n % 2 != 0 and odd == False:  
            continue  
        print(n, end=' ')  
    print()
```

This function displays the numbers of a list `L`. The last two arguments decide whether negative numbers and odd numbers will be displayed. If the second argument is `True`, negative numbers will be displayed, if it is `False` negative numbers will not be displayed. If the third argument is `True`, odd numbers will be displayed and if it is `False` odd numbers will not be displayed. The default value for both is `True`.

Inside the function we have used `continue` statement to skip negative and odd numbers if the corresponding variables are `False`. Now let us write some calls to this function.

```
display(numbers)  
display(numbers, False)  
display(numbers, False, True)  
display(numbers, True, False)
```

In the first call, we have not supplied the last two arguments, so their default value will be used. Both `negative` and `odd` are `True`, so all the numbers will be displayed. In the second call, negative numbers will not be displayed. The third one is equivalent to second one, negative numbers will not be displayed and odd numbers will be displayed. In the last call, negative numbers will be displayed, but odd numbers will not be displayed. By looking at these calls, we can't tell which parameter is `True` and which one is `False` and so it is not clear which type of numbers we want to display in a particular call. We can make the code more readable by using keyword arguments. Let us write some calls with keyword arguments.

```
display(numbers, odd=False)
display(numbers, negative=False)
display(numbers, negative=False, odd=False)
```

These calls are clearer than the previous calls. So, the keyword arguments make the calls more readable, particularly in the case of Boolean arguments. The keyword arguments are mostly useful when the functions become complex and have many parameters, and when most of them are optional parameters.

Keyword arguments enable you to add new parameters to a function, while remaining backward compatible with existing callers. This is particularly important when the function accepts variable number of arguments.

10.13 Unpacking Arguments

In this section, we will see how to use a container instead of individual argument values in a function call.

```
def result(m1, m2, m3, m4):
    total = m1 + m2 + m3 + m4
    per = total / 4
    print(f'Total Marks = {total}, percentage=
{per:.2f}%')
    print('Pass' if per > 40 else 'Fail')
result(56, 89, 77, 67)
```

We have a function that takes marks in 4 subjects, calculates total marks and percentage, prints them, and then prints Pass or Fail depending on the percentage. We have assumed that maximum marks in each subject is 100, so percentage is calculated by dividing total by 4.

Now suppose we have a list that contains the marks of 4 subjects,

```
marks = [93, 34, 54, 67]
```

We want to send the marks of this list to the result function. Here is the function call for it.

```
result(marks[0], marks[1], marks[2], marks[3])
```

In this call, we are sending all four elements of the list to the function. There is a simpler way of doing this in Python. We can write our call like this-

```
result(*marks)
```

This is equivalent to the previous call. We have added an asterisk before the name of the list, so the list is not sent as a single argument, it is unpacked and the elements of the list become separate positional arguments of the function. The four elements of the list are assigned to four parameters of the function. So, instead of providing individual positional arguments in a function call, you can provide a list, tuple or a set. And to tell the interpreter that you are not sending the list, tuple or set as a single argument, you need to add an asterisk before the list or tuple name.

If you don't put the asterisk in the call, then you will get an error, because now the function is getting only a single argument instead of four arguments.

```
result(marks) #Error
```

If the number of elements in the list that you are unpacking is not equal to the number of parameters, then also you will get an error. For example, suppose we have 6 elements in this list -

```
marks = [93, 34, 54, 67, 56, 89]
```

```
result(*marks)
```

We will get an error because 4 arguments were expected by the function but 6 were given. If we want to provide the first 4 elements from this list as argument, then we can use the slicing operator to get a new list object which will be unpacked.

```
marks = [93, 34, 54, 67, 56, 89]
result(*marks[:4])
```

So, if the number of elements in the list is more than the number of arguments, then we can use slicing to extract the correct number of arguments.

Now let us print the marks list using the print function.

```
marks = [93, 34, 54, 67, 56, 89]
print(marks)
```

Output-

```
[93, 34, 54, 67, 56, 89]
```

There is nothing special in it, we have done it many times. Again, we will send this list to the print function, but this time preceded with an asterisk.

```
print(*marks)
```

Output-

```
93 34 54 67 56 89
```

We can see the difference between the two outputs. In the call `print(marks)`, the print function got only one argument of list type, and in the call `print(*marks)`, it got 6 arguments of int type.

Similarly, there is difference between the calls, `print('Hello')` and `print(*'Hello')`. In the first call, print works only with one string argument, while in the second call, the string is unpacked and the print function gets 5 arguments.

If we use double asterisks before the argument in the call, then we can use a dictionary to provide the keyword arguments. For example, suppose we have a dictionary that contains the marks, and we want to calculate the result for the marks from this dictionary by using our same `result` function.

```
marks2 = {'m1': 93, 'm2': 34, 'm3': 54, 'm4': 67}
def result(m1, m2, m3, m4):
    total = m1 + m2 + m3 + m4
    per = total / 4
```

```
    print(f'Total Marks = {total}, percentage=
{per:.2f}%')
    print('Pass' if per > 40 else 'Fail')
result(**marks2)
```

The dictionary `marks2` has 4 pairs, and the names of keys are the same as the names of parameters in the definition of function `result`, so we can use this dictionary to send arguments to this function. Since `marks2` is a dictionary, we have to precede it with two asterisks to unpack it.

When an argument in a function call is preceded by a single asterisk (*), it means that the argument is a list, tuple, set or string. Such an argument will be unpacked and the contained values will be sent to the function as positional arguments.

When an argument in a function call is preceded by a double asterisk (**), it means that the argument is a dictionary. The dictionary will be unpacked and its key value pairs will be used to provide keyword arguments to the function.

So, we can use a single asterisk to unpack lists, strings or tuples for providing positional arguments, and we can use double asterisk to unpack a dictionary for providing keyword arguments. This is how we can unpack arguments; in the next section we will see how to pack arguments.

10.14 Variable number of positional arguments

By now you must have called `print()` function many times, have you noticed anything special about this `print` function that is not there in the functions that we have written till now.

```
print(1, 2, 3)
print(L, d, x, y, z)
print('Hello', 'world')
print(x)
```

The `print` function can be called with any number of arguments. Similarly, the built-in functions `max`, `min` or `sum` can also be called with any number of arguments. The functions that we have written so far don't have such capability, now we will see how to create functions that can accept any number of arguments.

We again take the function `result` that we saw in the previous section.

```
def result(m1, m2, m3, m4):
    total = m1 + m2 + m3 + m4
    per = total / 4
    print(f'Total Marks = {total}, percentage=
{per:.2f}%')
    print('Pass' if per > 40 else 'Fail')
```

This function has 4 parameters, so we can use it to find the result based on marks of 4 subjects only. If we try to find the result based on marks of 3 subjects or 5 subjects or any other number of subjects, we get an error.

```
result(45, 78, 99)           # Error
result(45, 78, 99, 77, 88)    # Error
```

We want to make this function more flexible so that it works with any number of arguments. For that we will need to make some changes in the definition. In the function header, instead of 4 parameters, we will write a single parameter, preceded by an asterisk.

```
def result(*args):
    .....
```

Now after this change in the function header, the function has become capable of accepting variable number of arguments. All the arguments that will be sent in the function call will be collected in a tuple named `args`. And we can use that tuple inside the function body.

Now to calculate the total of all marks we will call the function `sum`.

```
total = sum(args)
```

To calculate percentage, we will divide `total` by `len(args)` instead of dividing by 4. This is because now we don't have just 4 subject marks, the number of marks will be equal to the length of the tuple `args`.


```
per = total / len(args)
```

Rest of the code remains the same, we will print `args` in the beginning of the function so that we can see that it is actually a tuple. Here is the modified function.

```
def result(*args):  
    print(args)  
    total = sum(args)  
    per = total / len(args)  
    print(f'Total Marks = {total}, percentage=  
{per:.2f}%')  
    print('Pass' if per > 40 else 'Fail')  
result(23, 89, 77, 67, 89, 90)  
result(67, 83, 68)  
result(89)
```

Output-

```
(23, 89, 77, 67, 89, 90)  
Total Marks = 435, percentage=72.50%  
Pass  
(67, 83, 68)  
Total Marks = 218, percentage=72.67%  
Pass  
(89,)  
Total Marks = 89, percentage=89.00%  
Pass
```

The function `result` can now be called with any number of arguments. It can be called even without any arguments and in that case `args` will be an empty tuple. In this case we will get a divide by zero error because we have divided by the length of the tuple which is zero in this case.

So, to gather all the variable number of arguments, you just need to specify a parameter with an asterisk in front of it. You can give this parameter any other name of your choice, but conventionally it is named `args`.

When you need to write a function definition, but you don't know how many arguments it will receive when it will be called, you can use this `*args` parameter. You can combine this parameter with other parameters. For example, suppose our `result` function has two more parameters `name` and `standard`.

```
def result(name, standard, *args):
    print(name, standard)
    print(args)
    total = sum(args)
    per = total / len(args)
    print(f'Total Marks = {total}, percentage=
{per:.2f}%')
    print('Pass' if per > 40 else 'Fail')
result('Anu', 'VI', 34, 66, 88, 99, 344)
result('Dev', 'V', 99, 344)
```

Whenever this function is called, the first argument is assigned to `name`, second argument is assigned to `standard` and then the rest of the arguments, whatever be their number will be gathered in a tuple named `args`. This way we can collect all the extra arguments using `*args`. So, we can use other parameters with `*args`, but all those parameters should come before `*args`.

Now suppose we have these lists named `marks1`, `marks2`, `marks3` all of different sizes.

```
marks1 = [23, 45, 67]
marks2 = [23, 45, 67, 89, 88, 99]
marks3 = [56, 77, 88, 22, 77]
```

In the function call, we can send a list by using the asterisk.

```
result('Anu', 'VI', *marks1)
```

This is what we had done in the previous section on unpacking arguments. In this call, we have an asterisk before the argument named `marks1` so it will be unpacked. The list `marks1` will be unpacked and its elements will

become arguments of the call. This is actually equivalent to the following call.

```
result('Anu', 'VI', 23, 45, 67)
```

Now inside the function, these arguments will be packed in the tuple named args. So now we can send a list of any size to this function, which we were not able to do in the previous section.

```
result('Anu', 'VI', *marks2) # Sending a list of 6
elements
```

```
result('Anu', 'VI', *marks3) # Sending a list of 5
elements
```

Let us see one more example. We will write a function definition to find average of some numbers. We need this function to accept any number of arguments, so we specify the parameter with asterisk.

```
def average(*args):
    return sum(args) / len(args)
```

We can use this function to find the average of any number of arguments.

```
a1 = average(9, 2, 1)
a2 = average(3, 5, 6, 7, 8)
```

If we have a list or a tuple, then we can send it to this function by preceding the name with an asterisk.

```
L = [1, 5, 7, 8, 0, 4, 2, 8, 5]
a3 = average(*L)
```

We can use this function to find average of values of a dictionary also, for example suppose we have this dictionary.

```
d = {'John': 23, 'Ted': 25, 'Sam': 27, 'Nick': 21}
a4 = average(*d.values())
```

In this section we saw how to provide variable number of positional argument values. In the next section, we will see how to provide variable number of keyword arguments.

10.15 Variable number of keyword arguments

We can define a function in such a way that it can accept any number of keyword arguments. Previously, we learned that we could gather additional positional arguments using a single parameter by placing an asterisk before the parameter name. Similarly, to collect additional keyword arguments, we need to prefix the parameter name with two asterisks. While extra positional arguments were collected in a tuple, extra keyword arguments will be gathered in a dictionary.

We have seen that conventionally, the parameter that is used to collect positional arguments is named `args`. Similarly, by convention, the parameter that is used to collect keyword arguments is called `kwargs`. Therefore, in the function header, we generally write `*args` to collect positional arguments and `**kwargs` to collect keyword arguments. Let us see a simple example for this:

```
def func(**kwargs):  
    for x, y in kwargs.items():  
        print(x, y)
```

This is a simple function definition with `**kwargs` in the header. This parameter will be used to collect a variable number of keyword arguments. All those arguments will be collected in a dictionary named `kwargs` which we can use inside the function body. Inside the function, we are just iterating over the `kwargs` dictionary and printing its keys and values. Now, let us see how to call this function.

The definition of `func` has only one parameter and it collects keyword arguments, so while calling this function, we cannot send positional arguments. If we send a positional argument, a `TypeError` will be raised.

```
func(1, 3, 2) # TypeError: func() takes 0  
positional arguments but 3 were given
```

Now let us send keyword arguments:

```
func(a=1, b=2, c=3)
```

These keyword arguments will be collected in the dictionary named `kwargs` with the parameter names as the keys and the argument value as the corresponding values. We can call this function with any number of keyword arguments.

```
func(a=1, b=2, c=3, d=4, e=5, f=6)
```

So, to accept any number of keyword arguments, you have to prefix the parameter name with double asterisk. Instead of sending these keyword arguments explicitly we can send a dictionary in the function call. Since the dictionary has to be unpacked to get the keyword arguments, we need to prefix it with double asterisk. Suppose we have this dictionary:

```
mydict = {'a': 1, 'b': 2, 'c': 3}
```

We can call the function as shown below:

```
func(**mydict)
```

This is equivalent to the following call:

```
func(a=1, b=2, c=3)
```

To get more clarity on this, let us once again consider the definition of our function result.

```
def result(name, standard, *args):  
    total = sum(args)  
    per = total / len(args)  
    print(name, standard, args)  
    print(f'Total Marks = {total}, percentage=  
{per:.2f}%')  
    print('Pass' if per > 40 else 'Fail')
```

We will modify this function definition, so that it can accept variable number of keyword arguments.

```
def result(name, standard, **kwargs):  
    total = sum(kwargs.values())  
    per = total / len(kwargs)  
    print(name, standard, kwargs)  
    print(f'Total Marks = {total}, percentage=  
{per:.2f}%')  
    print('Pass' if per > 40 else 'Fail')
```

In the function header, we have written `**kwargs`. Now total will be the sum of all the values in the dictionary `kwargs`, so we have written

`sum(kwargs.values())`. The total number of values is `len(kwargs)`. We have printed the `kwargs` parameter along with `name` and `standard`.

Now when this function will be called, we have to send two positional arguments which will be assigned to `name` and `standard` first, and after that we can send any number of keyword arguments. Let us call this function:

```
result('Amit', 'VI', physics=89, maths=45)
result('Anuj', 'V', physics=89, maths=45,
chemistry=90, history=87)
```

In the first call, we sent two keyword arguments and in the second call, we have sent four keyword arguments. If we have a dictionary that contains subject names and marks, we can send it to this function by preceding it with double asterisks. Suppose we have the following dictionary:

```
marks = {'Physics': 78, 'Maths': 34, 'Chemistry':
89}
```

Let us send it to our `result` function:

```
result('Amit', 'VI', **marks)
```

The dictionary named `marks` was preceded with two asterisks in the call, so unpacking was done. The dictionary was unpacked and its keys and values were used as keyword arguments for this function. In the definition we have `**kwargs`, so inside the function all keyword arguments were collected in the dictionary named `kwargs`.

Since this function can accept any number of keyword arguments we can send dictionary of any size here. For example, we can send the following dictionary of size 5 to the function.

```
marks1 = {'Physics': 78, 'Maths': 34, 'Chemistry':
89, 'History': 89, 'Geography': 99}
result('Amit', 'VI', **marks1)
```

In a function definition, if we have to place both `*args` parameter and `*kwargs` parameter then the `*args` should be written first.

Here is a summary of unpacking and packing arguments that we learnt in the three sections:

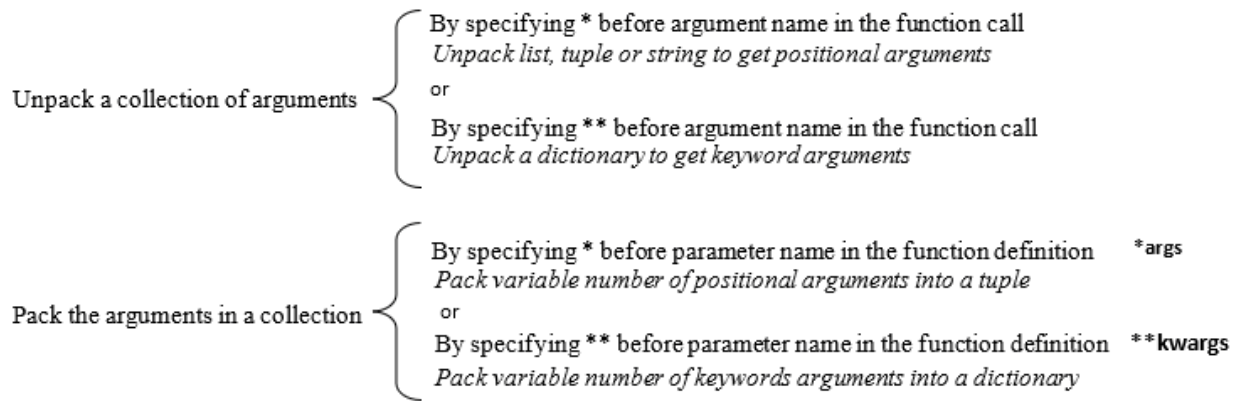


Figure 10.11: Unpacking and packing arguments

In [Section 10.13](#), we saw how to unpack a collection of arguments, in the next two sections, we did the opposite thing and learnt how to pack the arguments in a collection.

For unpacking, we used a single or a double asterisk before the argument name in the function call, for packing we used a single or a double asterisk before the parameter name in the function definition.

By specifying * before argument name in the function call, we can unpack a list, tuple or string to get positional arguments. By specifying ** before argument name in the function call, we can unpack a dictionary to get keywords arguments.

In [Section 10.14](#), we saw how to pack variable number of positional arguments into a tuple, and in [Section 10.15](#) we saw how to pack variable number of keyword arguments into a dictionary.

10.16 Keyword-only arguments

We have learned about keyword arguments and observed that they enhance the clarity of function calls. However, the caller of the function has the option of using either positional or keyword arguments in a function call. In this section, we will see a feature that will force the user to use only keyword arguments for certain parameters.

This feature of keyword-only arguments was introduced in Python 3, and it allows you to specify parameters that will accept only keyword arguments. These parameters will not accept arguments sent by position. Let us see how to specify these types of parameters in our function header.

```
def func(a, b, *args, c, d):  
    print(a, b, args, c, d)
```

In this function definition, arguments for `a` and `b` can be sent by position or by keyword, after that we can have variable number of positional arguments which will be collected in a tuple named `args`, and then the two parameters that are placed after `*args` can accept keyword-only arguments. This is obvious because if we send arguments for the parameters `c` and `d` by position, then those arguments would end up being collected in the tuple named `args`. Therefore, the only way to send arguments for the two parameters `c` and `d`, is by keyword. Let us write a call to this function:

```
func(2, 3, 6, 7, 8, 9, c=100, d=200)
```

Output-

```
2 3 (6, 7, 8, 9) 100 200
```

When this call executes, 2 will be assigned to `a`, 3 to `b`, then 6,7,8,9 will be collected in a tuple named `args` and the last 2 arguments are the keyword-only arguments. Let us try sending the last two arguments by position:

```
func(2, 3, 6, 7, 8, 9, 100, 200)
```

```
TypeError: func() missing 2 required keyword-only  
arguments: 'c' and 'd'
```

The arguments 100 and 200 will be captured by the tuple `args` and the interpreter does not find any argument for parameters `c` and `d`. Thus, the arguments for parameters `c` and `d`, can be sent only by keyword.

If you do not have plans of including a parameter preceded with an asterisk, but still want to include keyword-only arguments then you can specify just a single asterisk by itself in the definition. So, you can write your definition like this:

```
def func(a, b, *, c, d):  
    print(a, b, c, d)
```


The asterisk in this function definition denotes the end of positional arguments and beginning of keyword-only arguments. The last 2 arguments to this function can be sent only as keyword arguments and not as positional arguments. If you want to make all arguments keyword-only arguments, then you can put the asterisk in the beginning.

```
def func(*, a, b, c, d):  
    print(a, b, c, d)
```

Now all arguments must be sent by keyword-only. Let us see some more examples:

```
def display(L, negative=True, odd=True):  
    for n in L:  
        if n < 0 and negative is False:  
            continue  
        if n % 2 != 0 and odd is False:  
            continue  
        print(n, end=' ')  
    print()  
numbers = [12, -1, 3, 6, 8, 9, 38, -3, 34, -4]  
display(numbers, True, False)  
display(numbers, False)  
display(numbers, odd=False)  
display(numbers, negative=False)  
display(numbers, negative=False, odd=False)
```

We saw this program in [Section 10.12](#), when we learnt about positional arguments and keywords arguments. We had learnt that keyword arguments provide greater code clarity especially in these types of confusing cases. With this function definition, the users of the function can send arguments for these parameters by position or by keyword. If you want to force the users to send the last 2 arguments by keyword-only, you can place an asterisk in the header.

```
def display(numbers, *, negative=True, odd=True):
```

Now these types of calls are not possible:

```
display(numbers, True, False)
```

You have to send the last two arguments by keyword-only. By using keyword-only arguments, you can force clarity in the function calls.

```
display(numbers, negative=False)
```

```
display(numbers, negative=False, odd=False)
```

So, there are two ways of specifying keyword-only arguments: place them either after the variable positional arguments(`*args`), or after a single asterisk.

We have several examples of keyword-only arguments in built-in functions. For example, the arguments for `sep` and `end` parameters in the `print` function can be sent by keyword-only.

```
print(1, 2, 3, 4, sep='-', end=';')
```

Here, both `sep` and `end` are keyword-only arguments. We cannot provide them as positional arguments. Similarly, in the `max` and `min` functions also we have keyword-only arguments.

```
Max(1, -2, 3, 6, -9, key=abs)
```

Here we have provided a keyword-only argument, and the comparison will be done on absolute values. Similarly, in the `sorted` built-in function also, we have keyword-only arguments.

The keyword arguments cannot appear after a parameter prefixed with two asterisks, and the double asterisks cannot appear by itself in the argument list like the single asterisk.

10.17 Positional-Only Arguments

Similar to keyword-only arguments feature, there is a feature that forces the caller to provide only positional arguments for certain parameters. This feature was introduced in Python 3.8 and it can be used by placing a forward slash (`/`) in the function header. All parameters that come before the symbol `/` will accept only positional arguments.

```
def func(a, b, /, x, y):  
    print(a, b, x, y)
```

In this function, the parameters `a` and `b` appear before `/`, so we can send only positional arguments for them. The parameters `x` and `y` that are after the `/` are normal parameters, and they can accept both positional and keyword arguments. Here are some correct and incorrect function calls for this function:

```
func(1, 2, 3, 5)      # correct
```

```
func(1, 2, 3, y=5)    # correct
```

```
func(1, b=2, x=3, y=4)
```

```
TypeError: func() got some positional-only  
arguments passed as keyword arguments: 'b'
```

```
func(a=1, b=2, x=3, y=4)
```

```
TypeError: func() got some positional-only  
arguments passed as keyword arguments: 'a, b'
```

You will see this symbol `/` in many built-in functions when you seek help on them. For example, the help on `len` function shows this-

```
>>> help(len)
```

```
len(obj, /)
```

```
Return the number of items in a container
```

The parameter `obj` comes before `/` so it will accept only a positional argument. You cannot call `len` as follows:

```
len(obj = [1,2,3,4])
```

We can have both the symbols `/` and `*` together to mark parameters as positional-only or keyword-only.

```
def func(a,b, /, c,d, *, e,f);  
    pass
```

In this function, the parameters `a` and `b` will accept only positional arguments, `c` and `d` will accept both positional and keyword arguments, `e` and `f` will accept only keyword arguments.

If you see help on the sorted built-in function, you will see both the symbols used in the function header.

```
>>> help(sorted)
```

```
sorted(iterable, /, *, key=None, reverse=False)
```

Return a new list containing all items from the iterable in ascending order.

Now, let us see the motivation behind the inclusion of positional-only arguments feature in the language. Sometimes it is hard to choose meaningful names for the parameters. In such cases, the writer of the function would not want the caller to use those names for calling, as it does not add any readability benefit. The names of positional-only parameters will be used only inside the function body, so they are considered part of the implementation detail rather than part of the function's interface. The writer can choose any name for these parameters, as these names will not be used externally in the call. Therefore, when there are some parameters that do not have any externally-usable name, they can be marked as positional-only.

If a parameter can accept keyword arguments, then the name for that parameter cannot be changed in future. If it is changed, the existing function calls that use that name will break. If a parameter is marked as positional-only, then the function writer has the flexibility to change the name without breaking the caller's code. Therefore, positional-only arguments enable future evolution of the function while maintaining backward compatibility.

We know that positional arguments are matched with parameters based on their order. There can be functions where the logical ordering of arguments matters and you would not want the caller to change the order of arguments in the call. In those cases, you can use positional-only arguments.

Positional arguments are handled faster than keyword arguments, so one of the reasons for having positional-only arguments can also be performance.

Thus, use positional-only arguments when the parameter names do not have any external meaning and you might change them in the future, or when you want the user to always provide the arguments in a certain order. Use keyword-only arguments if parameter names are meaningful and they add clarity to the function call, or when you do not want the caller to worry about the order of arguments in the call.

10.18 Multiple Unpackings in a Python Function Call

Python 3.5 onwards, function calls can support any number of unpackings instead of just one. For example, we could use our average function to find average of multiple lists, sets or tuples:

```
def average(*args):
    print(args)
    return sum(args) / len(args)
x = {98, 23, 85, 56, 12}
y = (48, 98)
z = [67, 89, 43, 78]
print(average(*x, *y, *z))
```

Output-

```
(98, 85, 23, 56, 12, 48, 98, 67, 89, 43, 78)
63.36363636363637
```

Similarly, we could send multiple dictionaries to unpack while calling our result function:

```
def result(name, standard, **kwargs):
    total = sum(kwargs.values())
    per = total / len(kwargs)
    print(name, standard, kwargs)
    print(f'Total Marks = {total}, percentage=
{per}%')
    print('Pass' if per > 40 else 'Fail')
marks1 = {'Physics':78, 'Maths':34, 'Chemistry':89}
marks2 = {'History':89, 'Geography':99}
result('Amit', 'VI', **marks1, **marks2)
```

Output-

```
Amit VI {'Physics': 78, 'Maths': 34, 'Chemistry':  
89, 'History': 89, 'Geography': 99}  
Total Marks = 389, percentage=77.8%  
Pass
```

10.19 Arguments and Parameters summary

We have learnt quite a lot about parameters and arguments, so here is a quick summary of their usage in function definitions and function calls. First, we will see different variations of parameters in the function definition, and then we will see different variations of arguments in the function call.

Parameters in Function Definition

- A. `def func(name):` Match by position or by name
- B. `def func(name=value):` Default argument
- C. `def func(*args):` Collect extra positional arguments in
tuple named `args`
- D. `def func(**kwargs):` Collect extra keyword arguments in
dictionary named `kwargs`
- E. `def func(*args, name[=value]):` Keyword-only
arguments
- F. `def func(*, name[=value]):` Keyword-only arguments
- G. `def func(name[=value], /):` Positional-only arguments

In the function definition, if we write just a parameter name (as in A), it can be matched by position or by keyword syntax. Therefore, we can send either positional argument or keyword argument for the parameter.

If the parameter is followed by an equal sign and a value (as in B), the parameter becomes optional. If argument for it is not provided in the call, the default value provided in the function header will be used. This parameter can also be matched by position or by keyword as the parameter in A. The difference is that the parameter in A is a required parameter while parameter in B is an optional parameter.

If a parameter name is preceded with an asterisk (as in C), it will collect variable positional arguments in a tuple which has the same name as the parameter name. Conventionally the name `args` is used for such a parameter.

If a parameter name is preceded with double asterisks (as in D), it will collect variable keyword arguments in a dictionary which has the same name as the parameter name. Conventionally the name `kwargs` is used for such a parameter.

Parameters that are placed after the asterisk preceded parameter or a single asterisk (as in E and F), are parameters that accept keyword-only arguments.

Parameters that are placed before the forward slash (/) symbol are parameters that accept positional-only arguments.

Now let us see different ways of providing arguments in the function call:

Arguments in function call

H. <code>func(value)</code>	Positional Argument
I. <code>func(name=value)</code>	Keyword Argument
J. <code>func(*x)</code>	Unpacks container x into positional arguments
K. <code>func(**d)</code>	Unpacks dictionary d into keyword arguments
L. <code>func(*x, *y, *z)</code>	Multiple unpackings
M. <code>func(**d1, **d2, **d2)</code>	Multiple unpackings

If we provide just a value (as in H), it is a positional argument. If we provide parameter name and value (as in I), it is a keyword argument. If the argument is a list, tuple or set preceded with an asterisk, then that container is unpacked into positional arguments (as in J). If the argument is a dictionary preceded with double asterisks, then that dictionary is unpacked into keyword arguments (as in K). Multiple unpackings are also allowed as in L and M.

10.20 Function Objects

We know that `def` is an executable statement that creates a function object and assigns it to the function name. The execution of function definition does not execute the function body; the function body executes only when the function is called.

```
>>> def func(a, b):  
...     print(a + b)
```

When we execute this `def` statement, a function object that contains this function's code is created and it is assigned to name `func`. Like everything else in Python, functions are also objects. We can see the type and id of `func` by using the functions `type` and `id`.

```
>>> type(func)  
<class 'function'>  
>>> id(func)  
2220475491296
```

`func` is just a name that is referring to the function object. If you reassign it, you will lose access to your function. So, suppose we write this:

```
>>> func = 2  
>>> func  
2
```

Now `func` is an integer. If you try calling it, it will not work.

```
>>> func()  
TypeError: 'int' object is not callable
```

If we execute the `def` statement again, a function object will be created and assigned to name `func`.

```
>>> def func(a, b):  
...     print(a + b)  
>>> func  
<function func at 0x000001CBB92023E0>
```

Now `func` is again a function. We can assign the function object to some other name also. For example, we can write:

```
>>> add = func
```


Now the name `add` also refers to the same function object to which the name `func` is referring.

```
>>> add
<function func at 0x000001CBB92023E0>
```

The names `add` and `func` are referring to the same function object. We can call the function by adding parentheses to the name `add` or `func`.

```
>>> add(4, 5)
9
>>> func(4, 5)
9
```

We can assign the function object to some other name and then call the function by that name also. If we delete any of these names by using the `del` keyword, the other name will still work.

We can store function objects in a data structure like a list, tuple, set or a dictionary. Function objects are immutable so they can be used as dictionary keys also. Let us define some simple functions.

```
>>> def subtract(a, b):
...     print(a - b)
...
>>> def add(a, b):
...     print(a + b)
...
>>> def multiply(a, b):
...     print(a * b)
...
>>> def divide(a, b):
...     print(a // b)
```

We will store these functions in a list and then call each function by iterating over the list. All the four functions in the list will be executed:

```
>>> functions = [add, subtract, multiply, divide]
```

```
>>> for function in functions:
...     function(3, 4)
```

```
7
-1
12
0
```

We can also pass a function object as an argument to another function. The function that receives the function object as argument, can call that function using that function object. Let us see an example. We have the following function `calculate`, it has three parameters named `fn`, `a1` and `a2`.

```
>>> def calculate(fn, a1, a2):
...     fn(a1, a2)
```

Inside the function body, we have called `fn` with arguments `a1` and `a2`. When the function `calculate` is called, the first argument should be a callable object.

```
>>> calculate(add, 5, 2)
7
>>> calculate(subtract, 5, 2)
3
```

When the call `calculate(add, 5, 2)` is executed, `add` is assigned to parameter `fn`, and 5 and 2 are assigned to parameters `a1` and `a2`. Thus, the statement `fn(a1, a2)` inside the function `func` calls the function `add` with arguments 5 and 2. Similarly, when the call `calculate(subtract, 5, 2)` executes, the function `subtract` is called with arguments 5 and 2.

We can give a default value for `fn`, but for that, we have to place the parameter `fn` at the end.

```
def calculate(a1, a2, fn=add):
    return fn(a1, a2)
```

So now we have provided a default value for `fn`. Let us call it again:

```
>>> def calculate(a1, a2, fn=add):
...     return fn(a1, a2)
...
>>> calculate(1, 2)
3
>>> calculate(1, 2, multiply)
2
```

We can send our own function objects to built-in functions also. The built-in function `min` takes a keyword-only argument for an optional parameter named `key`. If we send the built-in function `abs`, the comparison is done on absolute value.

```
>>> min(-5, 2, -34, key=abs)
2
```

We can create our own absolute function and pass it to the function `min` as argument.

```
>>> def absolute(n):
...     return -n if n < 0 else n
...
>>> min(5, 2, -34, key=absolute)
2
```

We can return function objects from a function. In the following function `func`, we have defined function `fn` depending on the value of argument `x`. If `x` is less than 0, `fn` is defined such that it prints 'Hello', if it is greater than 0, `fn` is defined such that it prints 'Hi' and if `x` is zero, `fn` is defined such that it prints 'Hey'. At the end `fn` is returned from the function `func`.

```
>>> def func(x):
...     if x < 0 :
...         def fn():
...             print('Hello')
```

```

...     elif x > 0:
...         def fn():
...             print('Hi')
...     else:
...         def fn():
...             print('Hey')
...     return fn

```

When this function `func` will be called, only one of these `def` statements will be executed depending on the value of parameter `x`.

```
>>> f = func(6)
```

```
>>> f()
```

```
Hi
```

We called `func` with value 6, so the second `def` statement is executed, and the function object that was created was returned from `func`. The function object returned by `func` is assigned to `f`, so now `f` refers to the function object returned by the call `func(6)`.

Now, suppose we make a small change in the function definition, instead of `return fn` we write `return fn()`.

```

>>> def func(x):
...     if x < 0 :
...         def fn():
...             print('Hello')
...     elif x > 0:
...         def fn():
...             print('Hi')
...     else:
...         def fn():
...             print('Hey')
...     return fn()
...

```

```
>>> f = func(6)
```

```
Hi
```

```
>>> print(f)
```

```
None
```

When the call `func(6)` is executed, appropriate `def` statement executes which defines `fn` and then `fn` is executed. because in the return statement we have written the function call `fn()`. Now the return value of `func` is not a function object, instead the return value of `fn` becomes the return value of `func`. Return value of `fn` is `None` as it has no explicit return statement in its definition. So, `None` is returned from function `func` also.

We cannot write this type of code in traditional compiled languages such as C. In Python, the function definition happens at runtime, so we can conditionally execute the `def` statement. Thus, you can define a function in different ways depending on some conditions.

10.21 Attributes of a function

Functions are objects in Python, and so they have different attributes attached to them. We can use the `dir` function to get all the attributes.

```
>>> def func(a, b, c=1, d=2):
```

```
...     print(a, b, c, d)
```

```
...
```

```
>>> dir(func)
```

```
['__annotations__', '__builtins__', '__call__',  
'__class__', '__closure__', '__code__',  
'__defaults__', '__delattr__', '__dict__',  
'__dir__', '__doc__', '__eq__', '__format__',  
'__ge__', '__get__', '__getattr__',  
'__getstate__', '__globals__', '__gt__',  
'__hash__', '__init__', '__init_subclass__',  
'__kwdefaults__', '__le__', '__lt__', '__module__',  
'__name__', '__ne__', '__new__', '__qualname__',  
'__reduce__', '__reduce_ex__', '__repr__',
```

```
'__setattr__', '__sizeof__', '__str__',  
'__subclasshook__']
```

The attribute `__name__` gives the name of the function from the `def` statement.

```
>>> func.__name__  
'func'  
>>> fn = func  
>>> fn.__name__  
'func'
```

The `__name__` attribute for `fn` also gives the name `'func'` because `__name__` is the attribute of the function object to which `fn` is referring.

The `__module__` attribute tells us about the module of the function.

```
>>> func.__module__  
'__main__'
```

If we import a function from some module, the `__module__` attribute will show that module's name.

```
>>> from math import floor  
>>> floor.__module__  
'math'
```

The `__defaults__` attribute returns a tuple of all the default values specified in the function header.

```
>>> func.__defaults__  
(1, 2)
```

We can also attach our own attributes to a function to record some information.

```
>>> func.author = 'Ted'  
>>> func.creation_date = '9 Nov 2022'
```

We can see these two attributes when we call the function `dir`. To check these attributes, we can use the dot notation.

```
>>> func.author
```

```
'Ted'
>>> func.creation_date
'9 Nov 2022'
```

This way you can set and get your own attributes.

There is a built-in function named `getattr` that will show an attribute of a given function.

```
>>> getattr(func, '__name__' )
'func'
>>> getattr(func, '__module__' )
'__main__'
```

If we want to see a few attributes and their values, we can write a function that takes in a function name and shows some of its attributes.

```
def show_attributes(fn):
    attributes = ['__annotations__',
                  '__defaults__', '__module__', '__name__',
                  '__repr__', '__sizeof__', '__str__']
    for attribute in attributes:
        print(attribute, '->', getattr(fn,
attribute))
    print()
```

In the list named `attributes` we have placed some of the attributes that we want to see for a function and then we are iterating over the list and calling `getattr` for the function that is sent as argument. This function can be called like:

```
show_attributes(func)
show_attributes(divide)
```

10.22 Doctrings

When you use a built-in function, you want to know just what the function does, what arguments it takes and what it returns. You are not concerned about how the function does its job. Similarly, the functions that you write

might be used by some other users. They will want to call your function, so they need to know information like its purpose, arguments, and return value. This is why it is good to provide documentation. Documentation for functions is done by providing documentation strings, which are in short called docstrings. Here is an example of a simple function with a docstring:

```
def add(a, b):  
    ''' Add the two numbers.  
    Input: two numbers  
    Return : sum  
    '''  
    s = a + b  
    return s
```

A docstring is a string literal placed just after the header line and before the function statements. It is usually enclosed in triple quotes so that it can span more than one line. It appears as a tooltip when you try to call the function. When you seek help on the function then also it appears.

```
>>> help(add)  
Help on function add in module __main__:  
add(a, b)  
    Add the two numbers.  
    Input: two numbers  
    Return : sum
```

This docstring is stored in the `__doc__` attribute of the function.

```
>>> print(add.__doc__)  
Add the two numbers.  
    Input: two numbers  
    Return : sum
```

So, you can get the docstring even when the program is running. Unlike comments, docstrings are available at run time. Docstrings are helpful tools for generating automatic documentation. Comments are only for humans to read; they are ignored by the interpreter.

You can write a single line docstring or a multi-line docstring. Even if you write a single line docstring, enclose it in triple quotes, so that later if you need to add more things to it, you do not have to change the quotes.

Conventionally, the first line of a docstring starts with a capital letter and ends with a period. It generally states the purpose of the function in the form of a command or is a summary of the function. If the docstring is a multiline string, then the second line is left blank, and the rest of the explanation is written from the third line. This explanation can be used to describe parameters, return value, any preconditions, or any side effects. You can also include usage examples towards the end of the string. The ending quotes are placed on a separate line. If the docstring fits on a single line only, then the opening and closing quotes should be on the same line. You can refer PEP 257 for more docstring conventions.

Although docstrings can be placed anywhere in your code, they should not be used for writing multiline comments that explain parts of your code. Docstrings should be used for documenting particular components of your code like functions, methods, classes and modules. It is a good coding practice to include a docstring in your non-trivial components as they help in understanding the program.

10.23 Function Annotations

We know that there are no type declarations in Python for parameters and return values. The function definition does not specify the type of parameters or return type. Looking at the header of the function definition will not give the user any clue about the type of parameters they are supposed to pass into the function. So, users of your function have to look at the source code inside the function to know what type of information they are supposed to pass and what type of return value to expect from the function. Even after looking at the code sometimes, it may not be clear what type of arguments are expected from the user. One solution could be to document this information in docstrings. Although there are guidelines in PEP for writing docstrings, there can be differences in the way different programmers document information about the arguments and return type.

From Python 3.0 onwards, Python supports function annotations that can be used to annotate function parameters and return value. These annotations are also known as type hints, as they indicate the argument types and return value type. Let us see the syntax of providing these function annotations. We have a function that takes in three arguments and returns a string.

```
def func(s, i, j):  
    return s[i:j]
```

Let us add annotations to this function:

```
def func(s:str, i:int, j:int) -> str:  
    return s[i:j]
```

The parameter `s` is supposed to be a string, so we place a colon after the parameter name and then write `str`. Parameters `i` and `j` are supposed to be integers so we write `int` for them.

Return type is specified in between the closing parentheses and the colon and it is preceded by an arrow. This function returns a value of type `str`, so we have written `->str` between the closing parentheses and the colon.

We can write descriptive annotations, too. For example, if we want the user to send only values from 0 to 5 for the parameter `i`, then we can write the annotation in string form.

```
def func(s:str, i:'int 0 to 5', j:int) -> str:  
    return s[i:j]
```

Similarly, the annotation for the return type can also be descriptive. So, annotations not only document the expected types but also allow us to specify any type of metadata about the parameters and return type.

If there is default value for any parameter, then it is written after the annotation. In our function `func`, if we want default value for the last two parameters, we can write it like:

```
def func(s:str, i:'int 0 to 5'=0, j:int=3) -> str:  
    return s[i:j]
```

The function annotations are optional, and they are used just for documentation purpose, they do not enforce any type checking by the

interpreter. We can call our function with arguments different from the types specified in the annotations and the interpreter will not complain.

```
func([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 6, 8)
```

In this call, we are sending a list as the first argument and a list will be returned from the function. But the interpreter had no problems, this means that we can use annotations to just indicate the argument types and return types but it does not mean that Python will do type checking. Function annotation is just an informational tool that lets the users know how the function is supposed to be used.

These annotations are stored in annotations attribute of function in the form of a dictionary.

```
>>> func.__annotations__
{'s': <class 'str'>, 'i': 'int 0 to 5', 'j': <class 'int'>, 'return': <class 'str'>}
```

In this dictionary, parameter names are the keys and the annotations are the values. For the return type, key is named return.

Function annotations also appear when we seek help on the function.

```
>>> help(func)
Help on function func in module __main__:
func(s: str, i: 'int 0 to 5' = 0, j: int = 3) ->
str
```

If the function includes docstring also, then it will also be displayed after the annotations.

10.24 Recursive Functions

Recursion is a concept of defining something in terms of itself. By using the technique of recursion, a problem can be solved by solving smaller versions of the same problem repeatedly. The original problem is broken down into subproblems that are simpler and smaller instances of the original problem. Recursion can provide concise and elegant solutions to complex problems. There are certain data structures and algorithms that are best expressed using recursion. In programming, recursion can be implemented by creating a

function that calls itself. Such a function that calls itself inside its function body is called a recursive function. A function calling itself means that a function's execution instance is actually calling other execution instance of the same function.

```
def func() :  
    .....  
    .....  
    func()    # recursive call  
    .....  
    .....
```

The function `func()` is calling itself inside its own function body, so `func()` is a recursive function. When `func` is called, the code inside its definition will be executed and since there is a call to `func`, again the code of `func` will be executed. It seems that this process will go on infinitely, but in practice, a terminating condition is written inside the recursive function, which ends this recursion. This terminating condition is known as the exit condition or the base case. This is the case when the function stops calling itself and finally starts returning.

Recursion proceeds by repeatedly breaking a problem into smaller versions of the same problem, until finally we get the smallest version of the problem that is simple enough to solve. The smallest version of the problem can be solved without recursion, and this is known as the base case. Thus, recursion uses a divide-and-conquer approach to solving a problem.

To write a recursive function for a problem, we should be able to define the solution of the problem in terms of a similar type of a smaller problem. The two main steps in writing a recursive function are:

1. Identify at least one base case and its solution. Base case is a simple case where the solution can be achieved without recursion and so it does not include any recursive call. There can be more than one base case, depending on the problem.
2. Identify the general case or the recursive case. This is the case in which recursive calls are made.

A recursive function calls itself repeatedly until the base condition is satisfied. Every recursive function should have at least one base case, otherwise, the function will keep calling itself endlessly, resulting in infinite recursion. While writing the recursive case, we must ensure that each recursive call takes us closer to the base case, which means that the size of the problem should be downsized at each recursive call. The recursive calls should be made in such a way that eventually the base case is reached. If the base case is not reached, we will have infinite recursion. So, merely defining a base case will not help us avoid infinite recursion; we should implement the function such that the recursive calls progress toward the base case and the base case is finally reached.

Now, we will write some recursive functions and understand how they work. Although, most of the examples may not be very efficient they are classic examples of learning how recursion works. Some of these examples could be implemented more efficiently by using loops but writing recursive solutions will help you develop your recursive thinking and give you an idea of how to approach problems recursively. Initially, when learning recursion, we should trace the function calls and see how the control is transferred to understand the behavior of recursive functions.

A simple example of recursion is the factorial function used in mathematics. Factorial of a positive integer n is the product of all integers from 1 to n .

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

This is the iterative definition of a factorial, we have already written a program for it using a while loop in the exercise of [Chapter 7](#). Now, let us see how to find out the solution of this factorial problem recursively.

We know that $5! = 5 * 4 * 3 * 2 * 1$

We can write $5! = 5 * 4!$ (since $4! = 4 * 3 * 2 * 1$)

Similarly, we can write:

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

In general, we can say that the factorial of a positive integer n is the product of n and factorial of $n-1$.

$$n! = n * (n-1)!$$

The problem of finding factorial of $(n-1)$ is similar to that of finding factorial of n , but it is smaller in size. We have defined the solution of factorial problem in terms of itself. The factorial of 0 is 1 and this can act as the terminating condition or the base case. Thus, the recursive definition of factorial can be written as:

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

Figure 10.12: Recursive definition of factorial

The recursive function for finding factorial is a direct translation of this definition.

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)  
print(fact(3), fact(7))
```

Output-

6 5040

This function returns 1 if the argument provided is 0, otherwise it returns the value of expression $n * \text{fact}(n-1)$. To return the value of $n * \text{fact}(n-1)$, the value of $\text{fact}(n-1)$ has to be calculated for which the function `fact` has to be called again with an argument of $n-1$. This process of calling the function `fact` continues till it is called with an argument of 0. Suppose we want to find the factorial of 3.

Initially <code>fact (3)</code> is called	}	Winding phase
Since $3 > 0$, <code>fact (3)</code> calls <code>fact (2)</code>		
Since $2 > 0$, <code>fact (2)</code> calls <code>fact (1)</code>		
Since $1 > 0$, <code>fact (1)</code> calls <code>fact (0)</code>		

Figure 10.13: *Winding phase of recursion*

Initially, when the function `fact` is called, the argument is 3. So, in the first invocation of `fact`, the value of `n` is 3. Inside this first invocation, there is a call to `fact` with argument `n-1`, so now `fact` is invoked for the second time and this time the argument is 2. Now, the second invocation calls `fact` once more and this time argument is 1. We can observe that with each successive call, the function is invoked with a smaller argument. The third invocation of `fact` calls the `fact` with an argument of 0. In the fourth invocation of `fact`, the condition inside `if` statement becomes `True`, which means that we have reached the base case, so now the recursion stops and the statement `return 1` is executed. The winding phase terminates here and the unwinding phase begins and control starts returning towards the original call.

Now every invocation of `fact` will return a value to the previous invocation of `fact`. These values are returned in the reverse order of function calls.

Value returned by <code>fact (0)</code> to <code>fact (1)</code> = 1	}	Unwinding Phase
Value returned by <code>fact (1)</code> to <code>fact (2)</code> = $1 * \text{fact}(0) = 1 * 1 = 1$		
Value returned by <code>fact (2)</code> to <code>fact (3)</code> = $2 * \text{fact}(1) = 2 * 1 = 2$		
Value returned by <code>fact (3)</code> = $3 * \text{fact}(2) = 3 * 2 = 6$		

Figure 10.14: *Unwinding phase of recursion*

The following figure shows the flow of control when the function `fact` is called with argument of 3.

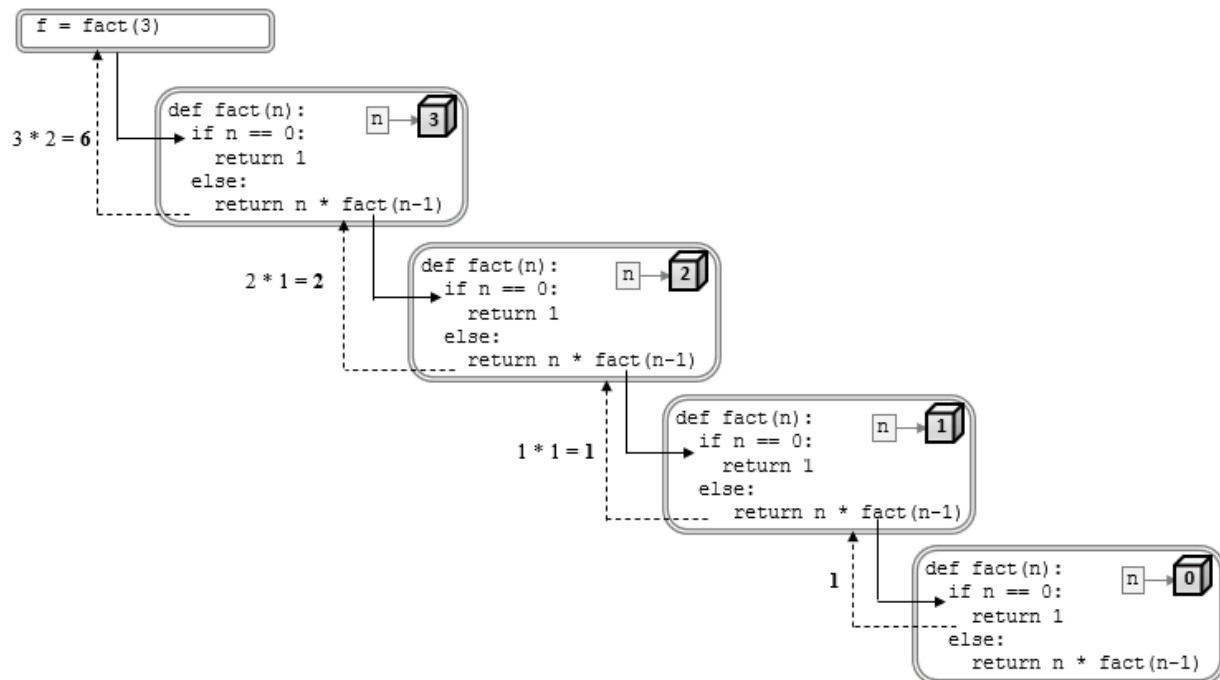


Figure 10.15: Recursive calls

We can see that the recursive functions are called in a manner similar to that of regular functions, but here the same function is called each time. Like normal calls, when a recursive call is made, the current call is suspended and the recursive call is executed. When the recursive call is executed fully, the current call is executed. When the execution of an instance of the recursive function is finished, we return to the previous instance where we had left it.

The function `fact` is called 4 times; each function call is different, and all these invocations have variables of their own. We know that for each function call, separate variables are created, and this is true for recursive calls also. When a function is called recursively, for each instance, a new set of formal parameters and local variables is created. Their names are same but they are different variables. These values are remembered till the end of function call so that these values are available while returning. In our example, we can see that there are four instances of `fact`, but each instance has its own copy of formal parameter `n`.

Recursive functions work in two phases - winding phase and unwinding phase. Winding phase begins when the recursive function is called for the first time, each recursive call continues the winding phase. In this phase the function keeps on calling itself and no return statements are executed in this

phase. This phase terminates when the terminating condition (base case) becomes true in a call. After this, the unwinding phase begins and all the recursive calls start returning in reverse order till the first instance of function returns. In unwinding phase, the control returns through each instance of the function. In [Figure 10.15](#), the winding phase is shown with solid arrows and unwinding phase with dotted arrows. In some algorithms we need to perform some work while returning from recursive calls, in such cases we can put that particular code in the unwinding phase i.e. just after the recursive call.

Next, we will write a recursive function to find out the sum of digits of a number. This problem can be defined recursively as:

sumdigits(n) = least significant digit of n + sumdigits (n with least significant digit removed)

The sum of digits of a single digit number is the number itself, and this can be used as the base case.

To find the sum of digits of 23546, the steps would be:

sumdigits(23546) = 6 + sumdigits(2354)

sumdigits(2354) = 4 + sumdigits(235)

sumdigits(235) = 5 + sumdigits(23)

sumdigits(23) = 3 + sumdigits(2)

sumdigits(2) = 2

Least significant digit of an integer n can be extracted by writing the expression $n \% 10$. The recursive call has to be made with the least significant digit removed and this can be done by calling the function with argument $(n/10)$. The base case would be when the function is called with a one-digit argument.

```
def sum_digits(n):  
    if n//10 == 0: # n is a single digit number  
        return n  
    else:  
        return n % 10 + sum_digits(n//10)
```

```
print(sum_digits(5432))
```

Output-

14

Our next recursive function will print the Fibonacci series. As we have seen earlier, Fibonacci series is a sequence of numbers in which each number is the sum of previous two numbers.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The problem of finding the n^{th} Fibonacci number can be recursively defined as:

$$\text{fib}(n) = \begin{cases} 1 & n=0 \quad (\text{Base case}) \\ 1 & n=1 \quad (\text{Base case}) \\ \text{fib}(n-1) + \text{fib}(n-2) & n>1 \quad (\text{Recursive case}) \end{cases}$$

Figure 10.16: Recursive definition of Fibonacci numbers

```
def fib(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
for i in range(10):
    print(fib(i), end=' ')
```

Output-

1 1 2 3 5 8 13 21 34 55

In this problem, we have two base cases and in the recursive case we have two recursive calls. The following figure shows the recursive calls of the function `fib` when it is called with argument 5.

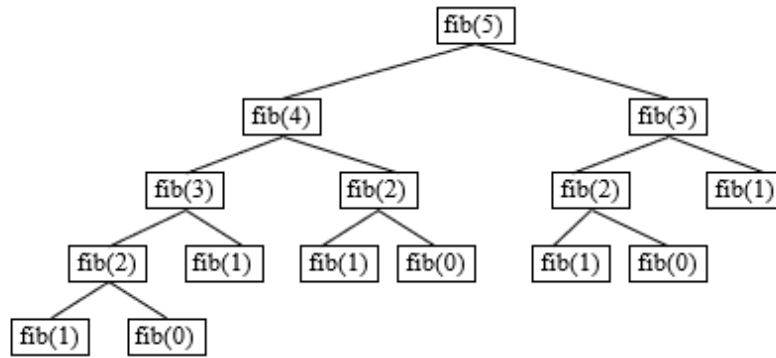


Figure 10.17: Recursive calls

This implementation of Fibonacci is not efficient as it performs same computations repeatedly, for example in the above tree we can see that the value of `fib(2)` was computed 3 times. The performance of these type of functions can be improved by using an optimization technique called memoization in which the results of function calls are stored, and the cached results are used when required again.

We know that when a function calls another function, the called function executes and the calling function resumes only when the called has been fully executed. The state of each active function invocation is maintained using a call stack. By using a call stack, Python can maintain the order in which the calls are to be executed. Stack is LIFO (Last In First Out) data structure; the item that is inserted (pushed) last is the first one to be removed (popped).

Whenever a function is called, a function frame or activation record is created that contains information about the local variables and parameters of the function and the return address. This frame is pushed on the stack and when the function call finishes executing, the frame is popped off the stack. In the winding phase, the stack grows as new activation records are created and pushed for each invocation of the function. In the unwinding phase, the activation records are popped from the stack in LIFO manner till the original call returns. Thus, each function call requires some memory, and in recursive functions many function calls are made and so there is a memory overhead associated with recursive calls.

There is a limit on the depth of recursion. If the depth of recursive calls exceeds this limit, Python will raise a `RecursionError`. To get this limit, you can use the function `getrecursionlimit` from `sys` module. You

can change the limit with the function `setrecursionlimit` of `sys` module. However, you cannot set this too high as the depth of recursive calls is limited by your system. The program might crash if the recursion limit exceeds the capabilities of your platform. So, if you have a recursive function that is giving a `RecursionError` due to maximum recursion depth exceeded, it is not advisable to increase the limit using `setrecursion`. It is best to write functions that are within the recursion limit.

```
>>> fact(99999)
RecursionError: maximum recursion depth exceeded
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(1500)
>>> sys.getrecursionlimit()
1500
```

Problems that require repetition can be implemented either recursively or iteratively. Recursion involves pushing and popping activation records of all the currently active recursive calls on the stack. Thus, the recursive version of a problem is usually slower than the iterative one because of the time spent in pushing and popping these activation records. It also consumes more memory as it uses space in the run time stack to store these activation records. If the recursion is too deep, we get a `RecursionError`. The iterative versions do not have to pay for this function call overhead and so are faster and require less space.

Recursive solutions might involve more execution overhead than their iterative counterparts, but their main advantage is that they simplify the code and make it more compact and elegant. Recursive algorithms are easier to understand because the code is shorter and clearer. Recursion should be used when the underlying problem is inherently recursive in nature (e.g. visiting nested directories) or when the data structure on which we are operating is recursively defined (e.g. trees and graphs). For some problems which are complex, iterative algorithms are harder to implement and it is easier to solve them recursively. In these cases, recursion offers a better way of

writing our code which is both logical and easier to understand and maintain. Sometimes it may be worth sacrificing efficiency for code readability. It is up to the programmers to select the correct approach that suits their requirements depending on the specific problem and the memory and performance constraints.

Exercise

1. Values that are passed in a function call are called:
(A) parameters (B) arguments
2. A function can be called only once in a program.
(A) True (B) False
3. The code of a function is executed when:
(A) the def statement executes
(B) the function call executes
4. When a function call returns, what happens to the local variable names?
(A) they no longer exist
(B) they continue to exist
5. Identifiers that are specified in function definition inside the parentheses are called:
(A) parameters (B) arguments
6. When a function does not explicitly specify a return statement, _____ is returned from the function?
(A) 0
(B) Nothing
(C) None
7. _____ arguments provide a sort of documentation for the function call.
(A) Positional (B) Keyword

8. When an argument is passed to a function, the object referred to by the argument is copied.
(A) True (B) False
9. Argument passing follows the semantics of the assignment statement.
(A) True (B) False
10. A function cannot change the binding of the caller's variable.
(A) True (B) False
11. If a list is passed to a function that changes the list in-place, then the argument list will _____.
(A) change (B) not change
12. Number of arguments can vary in different calls of the same function.
(A) True (B) False
13. All optional parameters must be placed _____ the required parameters.
(A) before (B) after
14. Keyword arguments should be placed _____ the positional arguments.
(A) before (B) after
15. Keyword arguments cannot be used for overriding default values.
(A) True (B) False
16. To accept any number of keyword arguments, the parameter name should be preceded with an asterisk.
(A) True (B) False
17. A function returns None:
(A) if a return with no expression is executed
(B) if the function terminates by reaching the end of the function body
(C) if return None is executed
(D) in all the above three cases

18. How many local variables are there in this function definition?

```
def func(x, y):  
    a = 8  
    b = 4  
    print(x + y + a * b)
```

(A) 0

(B) 2

(C) 4

19. What type of arguments should not be passed to this function?

```
def repeat(a, b):  
    print(a * b)
```

(A) string and integer

(B) integer and string

(C) float and integer

(D) float and string

20. What does this function do?

```
def func(number):  
    return number % 2 == 1
```

(A) returns True if number is odd

(B) returns True if number is even

(C) Gives Error

21. How many arguments do we need to supply while calling this function?

```
def func(x, y=3, z=10):  
    pass
```

(A) 1 or 2 or 3 arguments

(B) exactly 3 arguments

22. Which of these is not a valid call for this function?

```
def func(x=1, y=2):  
    pass
```

(A) func() (C) func(11, 8)

(B) func(11) (D) func(1, 1, 8)

23. With how many arguments can this function be called?

```
def func(a, y, *args):  
    pass
```

(A) 1 or 2

(B) 1 or 2 or more than 2

(C) 2 or more than 2

24. What will be the output when the following function is called without any arguments?

```
def func(*args):  
    print(args)
```

(A) gives error

(B) prints an empty pair of parentheses

25. Is there anything wrong with this function definition?

```
def continue():  
    print('Do you want to continue ? ')
```

26. Is this function definition correct?

```
def func(x=1, y):  
    pass
```

27. Will this function work in the same way if you remove the else?

```
def absolute(a):  
    if a < 0:
```



```
        return -a
    else:
        return a
```

28. `def func(a, b):`
 `print(a, b)`

Are the following calls equivalent?

`func(2, 3)`

`func(2, b=3)`

`func(a=2, b=3)`

`func(b=3, a=2)`

29. Is it possible to use default values for parameters `c` and `f` and provide our own values for parameters `d` and `e`?

```
def func(a, b, c=10, d=90, e=True, f=False):
    pass
```

30. `def func(a, b):`
 `print(a + b)`

Is the following function call valid?

`func(5, a=10)`

31. Will this code execute without any error?

```
def func():
    something
```

What will be the output of code given in questions 32 to 63.

32. `print('Welcome')`
 `func()`
`def func():`
 `print('Hello World')`

```

    print('Bye')
33. def func(x, y):
    print(x * y, end=' ')
    print(x)
    func(2, 3)
34. def func(a, b, c):
    return a + b + c, a * b * c
    print(func(2, 3, 4))
35. def func():
    print('Hello', end=' ')
    print(func())
36. def add(a, b):
    return a + b
    x = add(add(add(2, 3), 5), 8)
    print(x)
37. def f(x, y):
    return x + y
    def func(a, b, c, d):
    return f(a * b, c * d)
    print(func(1, 2, 3, 4))
38. D1 = {1: 'a', 2: 'b'}
    D2 = {1: 'a', 2: 'b'}
    def func1(d):
    d = {}
    def func2(d):
    d.clear()

```

```

func1(D1)
func2(D2)
print(D1, D2)
39. def func(x, y):
    x += y
    L1 = [5, 6]
    L2 = [7, 8]
    func(L1, L2)
    t1 = (5, 6)
    t2 = (7, 8)
    func(t1, t2)
    print(L1, t1)
40. my_dict = {1: 'a', 2: 'b', 3: 'c'}
    def func(d):
        d = {}
        d[1] = 100
    func(my_dict)
    print(my_dict)
41. def func(L):
    L.append(10)
    L = [7, 8, 9]
    L.append(10)
    numbers = [1, 2, 3, 4]
    func(numbers)
    print(numbers)
42. def func(L1, L2):42

```

```

        L1 = L1 * 2
        L2 *= 2
evens = [2, 4, 6, 8]
odds = [1, 3, 5, 7]
func(evens, odds)
print(evens, odds)
43.data1 = {1: 'a', 2: 'b', 3: 'c'}
    data2 = {1: 11, 2: 22, 3: 33}
    def func(d):
        d[2] = 'xxxx'
    func(data1)
    func(data2.copy())
    print(data1, data2)
44.def func(x):
    x = x * 3
    num1 = 10
    num2 = 1.5
    mylist = [1, 2, 3]
    s = 'hello'
    func(num1)
    func(num2)
    func(mylist)
    func(s)
    print(num1, num2, mylist, s)
45.def func(number, list1, list2):
    number = number + 1

```

```

        list1 = list1 * 2
        list2.append(100)
n = 35
my_list = [1,2,3,4]
your_list = [10,20,30,40]
func(n, my_list, your_list)
print(n, my_list, your_list)
46.def func(a, b=50, c=10):
    return a + b // c
    print(func(5))
47.def func(a, b, *x):
    print(x * 2)
    func(1, 2, 3, 4)
48.def func(a, d={}):
    d[a] = 0
    print(d, end=' ')
    func(10)
    func(20)
49.def func(*args):
    print(args)
    d = {'a': 1, 'b': 2, 'c': 3}
    func(*d)
50.def func(x, y, **z):
    print(x, y, z)
    func(2, 3)
51.def func(*args, **kwargs):

```

```

        print(args, kwargs)
    func(1, 2, 3, x=5, y=10)
52. def func():
    print('Hello')
    x = func
    del func
    x()
53. def func(n):
    print('hello ' * n)
    def f(x, y):
        x(y)
    f(func, 4)
54. def result(name, standard, *args):
    total = sum(args)
    print(f'{name}, {standard}, Total
Marks = {total}')
    result('Anu', 80, 95, 76, standard='V')
55. def func(a, b=8):
    print(a, b)
    func(4, 6)
    func((4, 6))
56. def func(x, y):
    x.append(1)
    y = []
    list1 = [1, 2]
    list2 = [1, 2]

```

```

    func(list1, list2)
    print(list1, list2)
57. def func(a, n):
    if n == 0:
        return 1
    else:
        return a * func(a, n-1)
    print(func(3, 2), func(4, 3), func(5, 1))
58. def func(a):
    if a >= 5:
        print('Hello', end=' ')
    else:
        priiint('Hi', end = ' ')
    func(10)
    func(100)
59. def func():
    func.count += 1
    func.count = 0
    func()
    func()
    func()
    print(func.count)
60. def func1(x, y):
    def f(a, b):
        return a + b
    return f(x, y)

```

```

def func2(x, y):
    def f(a, b):
        return a + b
    return f
j = func1(2, 3)
k = func2(2, 3)
print(type(j), type(k))
61. def greet():
    print('Hello', end=' ')
greet()
def greet():
    print('Hi', end=' ')
greet()
def greet(name):
    print('Hey', name)
greet('Jack')
62. def display1(n):
    if n == 0:
        return
    print(n, end = ' ')
    display1(n - 1)
def display2(n):
    if n == 0:
        return
    display2(n - 1)
    print(n, end = ' ')

```



```

display1(5)
print()
display2(5)
63. M = [[1, 6, 2, 3],
          [7, 5, 6, 9],
          [8, 9, 3, 2]
         ]
T = [list(t) for t in zip(*M)]
print(T)

```

64. Is there any error in the following code?

```

def subtract(a, b):
    print(a - b)
def add(a, b):
    print(a + b)
def multiply(a, b):
    print(a * b)
def divide(a, b):
    print(a // b)
d = {'a': add, 's': subtract, 'm': multiply,
     'd': divide}
choice = ''
while choice != 'q':
    print('a - Add')
    print('s - Subtract')
    print('m - Multiply')
    print('d - Divide')

```

```

print('q - Quit\n')
choice = input('Enter your choice :')
if choice == 'q':
    break
x = int(input('Enter a number : '))
y = int(input('Enter another number : '))
d[choice](x, y)

```

65. Write a function that multiplies all the entries of a list by a number.
66. Write a function that takes a number and returns the sum of digits in it.
67. Write a function `do_nothing()` that does nothing when executed.
68. Write a function that takes in a string and returns number of vowels and consonants in that string.
69. Write a function `is_prime` that takes in an integer and returns True if the argument is prime, otherwise returns False.
70. Write a function that returns factorial of a number.
71. Write a function that takes two arguments and returns sum, difference and product of those two arguments.
72. Write a function named `find` that takes in a list and a value. It should return True if that value is found in the list and False otherwise. Does your function work for strings, tuples, sets and dictionaries too?
73. Write a function named `fizzbuzz` that takes an integer as argument and returns 'Fizz' if that integer is divisible by 3, returns 'Buzz' if it is divisible by 5 and returns 'FizzBuzz' if it is divisible by both 3 and 5, otherwise it returns the integer itself. Use you function `fizzbuzz` in the following code.

```

def func(x):
    for i in range(1, x + 1):
        print(fizzbuzz(i))

```

```
func(50)
```

74. Write a function that takes in a list of integers and returns the number of even and odd numbers from that list.
75. If two consecutive odd numbers are both prime (e.g. (3,5) (17, 19)) then they are known as twin primes. Write a function that returns a tuple containing all twin primes in a given range. Use the `is_prime` function defined in question 69.
76. In the section on returning multiple values, we had written this function.

```
def max_min_avg(L):  
    return max(L), min(L), sum(L)/len(L)
```

Modify this function so that it can work with variable number of arguments.

77. Give reason for the output of the second function call.

```
def result(*args, grade=False):  
    total = sum(args)  
    per = total / len(args)  
    print(f'Total Marks = {total}, percentage  
= {per}%')  
    if grade == False:  
        return  
    if per > 80:  
        print('Grade A')  
    elif per > 50:  
        print('Grade B')  
    else:  
        print('Grade C')  
result(90, 90, 90, grade=True)
```

```
result(90, 90, 90, True)
```

Output-

```
Total Marks = 270, percentage = 90.0%
```

```
Grade A
```

```
Total Marks = 271, percentage = 67.75%
```

78. Write a function that accepts any number of integers passed to it and returns their product.
79. Write a function that takes in a variable number of strings and returns a list of all those strings in reverse form (use list comprehension).
80. In the function definition of the following function `display()`, make changes such that the user is forced to send keyword arguments for the last two parameters.

```
def display(L, start='', end=''):
    for i in L:
        if i.startswith(start) and
i.endswith(end):
            print(i, end=' ')
display(dir(str), 'is', 'r')
```

81. This function draws a box of asterisks of size 5 by 9. Make it more flexible so that it can draw a box of any size.

```
def draw_box():
    for i in range(5):
        for j in range(10):
            print('*', end='')
        print()
draw_box()
```

82. The following two functions find and return the median of a list of numbers. Which one will surprise the user and how can the user be

sure that the list that is sent to the function remains safe.

```
def median1(numbers):
    numbers.sort()
    mid = len(numbers)//2
    if len(numbers) % 2 != 0:
        return numbers[mid]
    return (numbers[mid-1] + numbers[mid])
/ 2

def median2(numbers):
    numbers = sorted(numbers)
    mid = len(numbers)//2
    if len(numbers) % 2 != 0:
        return numbers[mid]
    return (numbers[mid-1] + numbers[mid])
/ 2

nums1 = [2, 4, 5, 8, 6, 6, 3, 9]
nums2 = [2, 4, 5, 8, 6, 6, 3, 9]
print(median1(nums1), end=' ')
print(median2(nums2), end=' ')
print(nums1, end=' ')
print(nums2)
```

83. Rewrite the function `median2` of the previous exercise so that it accepts variable number of arguments.
84. The following function calculates and returns the compound interest. The value of rate is hardcoded inside the function. Change the definition so that caller gets to supply his own value of rate. Make this change in such a way that the existing function calls such as

`compound_interest(1000, 2)` don't stop working, they should continue to use 5 as the rate.

```
def compound_interest(principal, time):  
    amount = principal * pow((1 + 5 / 100),  
time)  
    return amount - principal  
print(compound_interest(1000, 2))
```

85. Can you write this function in a more concise form.

```
def func(a, b):  
    if a < b:  
        return True  
    else:  
        return False
```

86. Write a recursive function that computes the sum of integers from 1 to n.

87. Write a recursive function that inputs a decimal number and converts it to a string in binary, octal or hexadecimal base.

88. The greatest common divisor of two integers is the greatest integer that divides both of them without any remainder. It can be computed by using Euclid's remainder algorithm which states that-

$$\text{GCD}(a, b) = \begin{cases} a & \text{if } b = 0 \\ \text{GCD}(b, a \% b) & \text{otherwise} \end{cases}$$

Figure 10.18: Euclid's remainder algorithm

Write a recursive function that returns the GCD of two numbers.

89. Write a recursive function that computes the sum of all the integers in a list.

90. Write a recursive function that computes the sum of all integers in a nested list structure. For the following list, the function should return

29.

```
[[2,3], [1,4,'k', 3], [2], 'x', [5, 6, 2, 1]]
```

91. In the chapter, we saw the recursive functions for finding factorial and sum of digits. If we remove else from them, will they work in the same way.

```
def fact(n):  
    if n == 0:  
        return 1  
    return n * fact(n - 1)  
  
def sum_digits(n):  
    if n // 10 == 0:  
        return n  
    return n % 10 + sum_digits(n // 10)
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Modules and Packages

11

So far, we have been using a single file to write our entire program. This approach works as long as the program is small, but when a program starts growing in length, it becomes difficult to manage the whole code in a single file. Real-world applications contain thousands of lines of code, and if you write it all in a single file, then it would be difficult to understand, maintain, and update. Thus, when we have a lot of code in our program, it is convenient to store it in more than one file. Python modules help us to organize the code of our program in different files and they also make the code reusable. You can create your own modules or use existing modules. In this chapter, we will see how to create and use modules, and we will also learn about the concept of packages, which helps in organising modules.

11.1 Modules

Any file with the `.py` extension is considered a module, there is no special syntax required to make it a module. The file can contain any valid Python code, but it mostly contains functions, class definitions, and global variables.

Suppose we have a lengthy program containing many function definitions. We can better manage the program if we split it into different files. In the following example, we have shifted some function definitions to the file `module1.py` and some to the file `module2.py`. This makes our main program short and manageable.

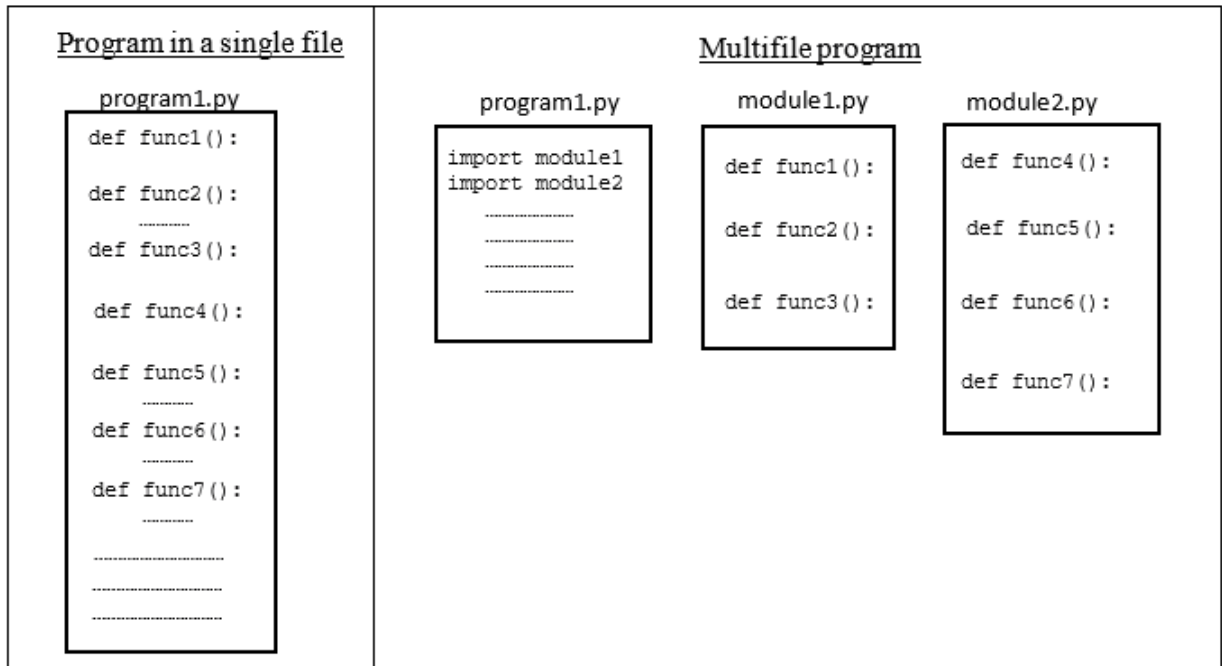


Figure 11.1: Single file program and multifile program

We changed our single file program to a multifile program consisting of three files. The file `program1.py` is the main module or the main script that is used to run the program. The other two files, `module1.py` and `module2.py`, are the modules that will be used in the main script. To make the code of these modules available in the main script, we need to import them by writing the `import` statement.

```
import module1
import module2
```

To import a module, we write the `import` keyword followed by the module name. The name of the module is the name of the file without the `.py` extension. For example, if the file name is `module1.py`, then the module name is just `module1`. There are different variations of the `import` statement, we will see the detailed syntax and usage in a short while.

Modules help us to organize our big program into small files that are easily manageable. While developing a program, you can group related pieces of code together and place them in separate modules. Having code in different modules is essential when multiple programmers work on the same project simultaneously. They can develop and test separate modules, which can be

combined later to create the whole program. Also, it is easier to develop, modify, test, and debug separate modules. Bugs in a system can be traced to a particular module, making debugging easier.

Modular development enables code reusability across projects. You can create modules that contain code which can be reused in different programs. In the example we have seen, we placed the related function definitions in separate modules, thus making those groups of definitions reusable. If we need those definitions in other programs, then instead of copying them, we can just import the module. So, modules make the code easier to understand and maintain and also allow us to reuse and share code.

11.2 Types of modules

We can have basically three types of modules in Python:

- User defined modules
- Standard modules
- Third-party modules

User defined modules are the modules that are written by the programmer for their requirements. The modules `module1` and `module2` that we saw in the previous section are examples of user-defined modules.

There are many standard modules available with Python that you can use in your main script or in any of your own modules. The term ‘Batteries included’ is often used for Python because it comes with this standard library of modules that can be used to perform different types of tasks. There are many standard modules that perform generic programming tasks specific to the web, GUI, files, text pattern matching, and more. We have already used some standard modules like `pprint`, `random` and `math`. These modules are available with Python installation, but they are not a part of the core language, you need to import them to use them in your programs. Before writing any utility, you can check if something like this already exists in the standard library.

Third-party modules are available from external sources, and they must be installed before they can be imported and used in our programs. For

`example`, `numpy`, `pandas` and `matplotlib` are packages that need to be installed separately.

To see the list of modules available, you can write this on the interactive prompt:

```
>>> help('modules')
```

It is better to make use of the predefined functionality present in the standard library or external libraries instead of creating everything from scratch. Library modules present robust and high-quality software that can make your development faster.

The process of importing and using these modules is the same for all the three types, even if the module is written in a language other than Python.

Python also supports extension modules which include source code written in other languages such as C, C++ or Java. The client using these extension modules can use them in exactly the same way as modules written in Python. In this book, we will discuss how to write modules in Python.

11.3 Exploring modules

To see what a module offers, you can import it in interactive shell and use the built-in `dir` function or `help` function. This way you can explore any user defined or predefined module in the interactive mode.

```
>>> import random
```

```
>>> dir(random)
```

```
['BPF', '.....', 'randint', 'random', 'randrange',  
'sample', 'seed', 'setstate', 'shuffle',  
'triangular', 'uniform', 'vonmisesvariate',  
'weibullvariate']
```

The function `dir` returns a sorted list of strings that contains all the names defined at the top level in a module. It also contains the default Python attributes associated with a module.

After viewing the names, if you want to know more about a particular item in the module, you can call the `help` function for the same.

```
>>> help(random.randint)
```

Help on method randint in module random:

randint(a, b) method of random.Random instance

Return random integer in range [a, b], including both end points.

To get the documentation related to the whole module, you can send the module name to the help function.

```
>>> help(random)
```

After importing a module, if you type its name on the interactive prompt, you will get to see the path of the file from where the module was loaded.

```
>>> random
```

```
<module 'random' from  
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python310  
\\lib\\random.py'>
```

The library modules are generally present in `lib` directory, where Python is installed. You can view the whole file by following the specified path, but make sure not to change anything in these predefined standard modules; otherwise, your programs might stop working. Viewing library code is not required for using the library, but it can sometimes help you understand how a particular library function works. These modules are written by professionals, and reading the library code can teach you programming tricks and can help you improve your programming skills.

Some standard modules are written in C and are integrated with the Python interpreter for efficiency and other reasons, for example `math` and `itertools` modules are built-in modules written in C.

```
>>> import math
```

```
>>> math
```

```
<module 'math' (built-in)>
```

We cannot see any source file here because this module is built into the interpreter. However, for the users this does not make a difference, the usage of all the modules is the same irrespective of the language they are written in or whether they are built into the interpreter or loaded from a file.

11.4 Creating and naming a new module

To create a module, you do not have to do anything special. Just create a .py file and place your code in it. Any Python file with a .py extension is already a module. While creating a module, you should keep in mind that the module name should be a valid Python name.

If you use a file named `123text.py` as a module, you will get an error upon writing your import statement because the name `123text` is not a valid Python identifier name as it starts with a digit. This file name is acceptable if you intend to execute this file as a script, but it is unacceptable if you have to import it as a module. So, name of a file that needs to be used as module should be a valid Python name. Module names should be generally short and all-lowercase names, and underscores can be used to improve readability.

11.5 Importing a module

We have seen that a program needs to use the `import` statement to get access to the code inside the module. The general syntax of `import` statement is to write the `import` keyword followed by the module name.

```
import module_name
```

This will import the entire module which means that any name defined at the top level of the module is available for use as long as that name is prefixed with the module name. We will create some files to see how to use the names from an imported module.

We have created two files named `words.py` and `numerals.py` that contain some function definitions.

```
----- words.py-----
```

```

# count words
def count(string):
    return len(string.split(' '))

# first word
def first(string):
    return string.split(' ')[0]

# last word
def last(string):
    return string.split(' ')[-1]

# sorted words
def ordered(string):
    return ' '.join(sorted(string.split(' ')))

# return a string with each word reversed
def reverse(string):
    words = string.split(' ')
    return ' '.join([word[::-1] for word in words])

```

-----numerals.py-----

```

def is_even(num):
    return num % 2 == 0

def reverse(num):
    return int(str(num)[::-1])

def sum_digits(num):
    s = 0
    while num != 0:
        s += num % 10
        num //= 10
    return s

```

```
def factorial(num):
    fact = 1
    while num > 0:
        fact *= num
        num -= 1
    return fact
```

The following file named `my_program.py` is the main module, it needs to use the functions defined in the two modules named `numerals` and `words`, so it imports these two modules. All the three files are in the same directory.

```
-----myprogram.py-----
import numerals
import words
print(numerals.factorial(5))
print(numerals.reverse(3459))
s = 'This is my book'
print(words.count(s))
print(words.reverse(s))
```

We could have imported the two modules in the same `import` statement by separating them with a comma, but it is generally better to import each module on a separate line. We can write these `import` statements anywhere inside the file, even inside functions or if statements, but it is customary to place them at the top of the file.

After importing the modules, we can use the functions defined in these modules. To use any name that is defined inside the module, we need to prefix it with the module name and a dot. For example, if we need to use the `factorial` function from the `numerals` module, we have to write `numerals.factorial(5)`. This will call the `factorial` function from the `numerals` module. Similarly, we can call other functions from both these modules.

We have a `reverse` function in both `numerals` and `words` modules, and we imported both these modules in our main script, which means that both the `reverse` functions are available in this file. However, there was no name conflict because each function is prefixed with its module name.

11.6 Importing all names from a module

The `from` statement is a variation of the `import` statement, it allows any program to access the names defined in the imported module directly without prefixing with the module name. If we want to access all the names of a module, then we can write this form of `from` statement.

```
from module_name import *
```

All the names defined at the top level of the imported module become available in the global scope of the importing module. Let us again take the example of modules that we saw in the previous section. In the file `myprogram1.py`, we will use the `from` statement with wildcard character `*` to import everything from the `numerals` module.

-----myprogram1.py-----

```
from numerals import *  
print(factorial(5))  
print(reverse(3459))
```

All the module level names of the `numerals` module become global variables in our file `myprogram.py`. All these names are directly accessible, which means that there is no need to prefix the names with the module name and dot. In fact, the module name is not present in the current scope.

This form of import seems more convenient as it requires less typing but it can cause name conflict with names defined in this program or with built-in names and names imported from other modules. We know that if there are two objects bound to the same name, Python does not show any error, it just rebinds the name. So, if there is a function in your program that has the same name as a function that is imported, then whichever name is encountered

later will overwrite the previous one. For example, now suppose we import the module `words` also using the `from` statement with wildcard character `*`:

-----myprogram2.py-----

```
from numerals import *
from words import *
print(factorial(5))
print(reverse(3459))
s = 'This is my book'
print(count(s))
print(reverse(s))
```

Output-

```
120
```

```
AttributeError: 'int' object has no attribute
'split'
```

When we execute this file, the call to `factorial` function will work but the call `reverse(3459)` will not work.

The `words` module also has a function named `reverse`, so there is a name conflict. In our program, first we have imported the `numerals` module and so the name `reverse` is bound to the function present in the `numerals` module. After this, the `words` module is imported, which also has a function named `reverse` and so now the name `reverse` is rebound to the function present in the `words` module. So, now in the program, the name `reverse` refers to the function from the `words` module and due to this reason, the call `reverse(3459)` will not work as the argument is an integer and the `reverse` function of `words` module needs a string as argument.

This is why, it is not a good programming practice to use this form for importing everything from the module. Using this is risky because when you import a large module that you have not written, there will be many

unknown names introduced in your scope that can lead to name clashes resulting in confusion and unexpected behaviour.

This form also reduces the readability of the code. You will not be able to know where an identifier is coming from. For example, you do not know the origin of the name `factorial` or `reverse`, you have no idea about whether it has been defined in this file or it has been imported, or from which module it has been imported.

Module-qualified names are safer as there is no chance of name conflict, and they also make the code more readable and clearer. The use of the module name and the dot clearly indicates that we are referring to a name that is defined in the imported module. So, it is best to avoid the wildcard `*` approach and either import the whole module and access names using the module name or import only some specific names (we will see how to do this shortly). However, you can use the `*` form to save typing while you are experimenting in the interactive sessions.

The import `*` form is allowed only at the module level. If you try to write this inside a class or function definition, it will result in a `SyntaxError`.

11.7 Restricting names that can be imported

A name that is prefixed with single underscore is not accessible to any importing module that uses the import `*` syntax for importing. Prefixing a name with an underscore indicates that the name is private (for internal use of the module) and should not be accessed by the client code. Names which do not have a leading underscore are meant for public use and form the module's interface.

In our `numerals` module, if we add a function named `_func` that starts with an underscore, and import the module using the `*` syntax then everything but `_func` will be imported.

```
-----numerals.py-----
```

```
.....
```

```
.....
```

```
def _func() :
```

```

    pass

-----myprogram.py-----

from numerals import *
print(factorial(5))
print(reverse(3459))
__func()

```

Output-

```

120

9543

NameError: name '__func' is not defined

```

When we execute this file, the function `factorial` and `reverse` will work, and after that we get a `NameError` stating that name `'__func'` is not defined which means that this name was not imported.

There is nothing strictly private in Python, prefixing with underscore is just a convention; you can access these names directly. However, it is best to respect the convention as the private names are used in implementation which could be changed in future releases. While using a module, it is best to stick to the public interface as it is not likely to change in any future release. You will see a similar convention when learning about classes.

Apart from this underscore approach, there is another way that is used to control access to names while importing using the `import *` syntax. To restrict the names that can be imported from a module, you can define an attribute named `__all__` in the module. It is a list of strings, and if this `__all__` is defined in the module, only then the names in this list will be made accessible to the importing module that uses the `import *` syntax. Let us add this list to our `numerals` module:

```

-----numerals.py-----

__all__ = ['is_even', 'sum_digits']

.....

.....

```

Now only these two names will be imported when an import is performed using the * syntax. The names should be listed as strings, meaning they should be placed inside quotes.

```
----- myprogram.py -----  
  
from numerals import *  
print(sum_digits(234))  
print(factorial(5))
```

Output-

```
9  
  
NameError: name 'factorial' is not defined
```

If we execute this, we will get `NameError` for the name `factorial`. You can open any library file to see this `__all__` defined at the top.

We saw two approaches to control the visibility of names when import is done using the * syntax. If `__all__` is defined, then only the names in this list will be imported and if it is not defined then all names that do not begin with an underscore are imported. These approaches work only with * syntax. Any name can be always be imported directly as we will see in the next section.

11.8 Importing individual names from a module

If you want to import only some names from the module, then after the `import` keyword you can specify a comma separated list of names that you want to import.

```
from module import name1, name2, .....
```

This `from` statement has specific names instead of the wildcard character *. So instead of importing everything from the module, we can import just one or more names which we intend to use. The names are directly accessible in the importing module.

```
-----myprogram.py-----
```

```
from numerals import sum_digits, factorial
print(sum_digits(234))
print(factorial(5))
```

Here we imported `sum_digits` and `factorial` from the `numerals` module, so now only these two functions from the module are available in this file.

We can refer to the names without the module name, so there are chances of name conflict. But unlike the `import *` syntax, here you know exactly which names you are importing and this makes name collisions less likely than the `import *` syntax. We can import a name directly when we are sure that nothing else with that name is present in our scope.

If you want to import many names from a module, you can optionally enclose the names inside parentheses to continue your logical line, or you can use the backslash character `\` for line continuation.

11.9 Using an alias while importing

When we import an entire module using the `import module_name` syntax, we need to prefix the names with the module name which can be cumbersome if the module name is long. In such cases, we can rename the module to a shorter name while importing. This can be done by using the `as` keyword in the import statement. For example, the following statement imports our `numerals` module with the alias `num`:

```
import numerals as num
```

The module `numerals` is imported in the regular way but now it is available as `num`. To use a name from the module, you need to add `num` as a prefix to it, instead of `numerals`. For example, to call the function `factorial`, you have to write `num.factorial`. If you write `numerals.factorial`, it will not work as `numerals` is not recognised in the scope.

We can use this form to provide a shorter name for our module. This can save typing if the module name is long, and it allows us to write more

concise code. Any user-defined module, standard module or third-party module can be imported using an alias.

This feature can be useful for testing newer versions of modules without disturbing the code. For example, suppose `primes` is the current working version of the module, and `new_primes` is the newer version that is developed. In your program, you just need to change the importing line from `import primes` to `import new_primes as primes`. By doing this, you will not need to make any changes in the rest of the program.

This feature can also be used when you want to experiment with different implementations of a module. For example, suppose `primes1`, `primes2` and `primes3` are three different modules that contain same functions but implemented in different ways. We can easily switch to any implementation by using the aliasing feature:

```
import primes1 as primes
```

or

```
import primes2 as primes
```

You can also use this feature to import interchangeable versions of a module conditionally for using the same name in your code.

```
if condition:
```

```
    import primes1 as primes
```

```
else:
```

```
    import primes2 as primes
```

```
# Rest of the code uses the name primes
```

You can also use aliasing while importing individual names using the `from` statement.

```
from numerals import reverse as rev, factorial as fact
```

```
from words import count as cnt, reverse as reverse_word
```

Now that the original name is not recognised in the scope, you have to use the alias. You can use this feature to avoid any conflict with an existing name. If any name that you have to import conflicts with an existing name in your program or in an imported module, then you can import the name with an alias. You can also use this feature to shorten names that are too long.

11.10 Documenting a module

It is good to document a module if it is going to be used by other programmers who were not involved in the development of the module. To document a module, you can add a docstring in the beginning of the module. It is a string enclosed in triple quotes and is used to give information about the module and its contents. The first line of the docstring should generally state the purpose of the module and after that you can document other things that the user should know about the module. This docstring is available to any importing file in the form of attribute `__doc__`. It also shows up when `help()` function is called with the module name after importing.

```
# numerals.py

"""This module defines functions related to
numbers"""

.....

.....

>>> import numerals
>>> numerals.__doc__
'This module defines functions related to numbers'
>>> help(numerals)
Help on module numerals:

NAME

    numerals - This module defines functions
related to numbers
```

Since most of the modules are meant to be reused, it is good to document them.

11.11 Module search Path

When you import a module in a file, Python needs to locate it for loading it into memory. So far, we have been creating our modules in the same directory in which we have our program. This way Python can easily find and import them. Apart from our program's directory, there are other places also where Python looks for the module that is being imported.

Let us see how Python searches for a module that we import. Suppose you import a module named `my_module` in your script. It first looks for a built-in module with this name. If it does not find a built-in module named `my_module` then it will search in the directories listed in `sys.path` which is a list defined in the standard module `sys`. Python configures it at the time of program startup. We can import `sys` module and see what is there in `sys.path`.

```
import sys

for dir in sys.path:
    print(dir)
```

Output-

```
['C:\\Users\\deepali\\Myfolder',
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python311\\python311.zip',
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python311\\Lib',
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python311\\DLLs',
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python311',
'C:\\Users\\deepali\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site-packages']
```


This is the list of directories that Python will search while importing a module. The first entry `path[0]` is the directory that contains the program that is being executed (script that invoked the interpreter). If the interpreter is invoked interactively and there is no script that is being run, then `path[0]` is an empty string which represents the current working directory. After this, there are directories specified in the `PYTHONPATH` environment variable if it is set, directories that contain standard library modules and site-packages directory that contains the third-party modules that you might have installed. Python searches for the imported module in these directories in the order in which the paths are listed in `sys.path`. It, first, looks in the first path, and continues searching in the second path and so on, if not found. It keeps searching in all the paths specified in the list till the module is found or till all the paths are searched. If the module is not found in any of these directories, then `ModuleNotFoundError` is generated which is a type of `ImportError`.

The paths listed in the `sys.path` list may be different for you. They are dependent on your operating system, Python installation and Python version. In IDLE, the Path browser in File menu will show you the paths that Python will search when you import a module.

If we want a directory to be searched by Python while importing, then we can add it to the module search path by appending the path of that directory to the `sys.path` list at run time.

```
sys.path.append('C:\\Deepali')
```

If we execute this statement before the `import` statement in our program, then Python will search for the imported module in this directory also. However, this insertion is for a single program run only.

Another way to let Python locate your module is to add your module in any of the directories specified in the `sys.path` list. It is not advisable to place your modules in standard library directories; you can save them in site packages which stores third part libraries.

So, if you want to store your modules in some location other than your current directory, you have to make sure that your directory is there in the list of search paths specified by `sys.path`.

Python searches the paths of `sys.path` in order and wherever Python first finds the module, it will load it from there. We have seen that our program's directory appears first in `sys.path`. This means that if you create any module in your program's directory that has the same name as any of library modules, then your module will be found first. This name conflict will make the library module inaccessible. For example, suppose you define a module named `fractions` and Python finds it before the library module then you will be not be able to access the library's `fractions` module. So, you should generally not name your modules the same name as any library module. Do this only in case you want to deliberately hide the library module and instead use your own.

A different kind of problem will arise if you name your module as one of the built in modules. In this case, your module will be inaccessible because Python looks in built-in modules before searching any directories. For example, if you name you module as `math`, then Python will never look for any file to import it as there is a built-in module with the name `math`.

`sys.builtin_module_names` will give you the names of all the all built in modules, and `sys.stdlib_module_names` will give you the names of the standard library modules. An easier way to see whether a name is used by any predefined module, is to import it on the prompt. If a module name is successfully imported then it means that a module with that name already exists and you should use another name for your module to avoid any name conflict. If `ModuleNotFoundError` is raised then it is safe to use that name for your own module.

11.12 Module object

Let us see what happens after Python has located the module. When Python finds an imported module, a module object is created in memory and the code inside the module is executed. The module object is analogous to a function object that is created when a function is executed. A module object has named attributes that you can bind and reference. So, like all other things in Python, modules are also objects. A module is a first-class object and can be bound to a variable. It can be an item of a container or an attribute of an object. It can be passed as argument to a function, or can be returned from a function call.

When you write an import statement like `import numerals`, Python looks for the file `numerals.py`, and it creates a module object and binds the name `numerals` to that object in the current scope. We can use the `dir` function to see all the names in the current scope. The `dir` function when used without arguments gives an alphabetically sorted list of names in the current scope.

```
import numerals

print(dir())

print(numerals)
```

Output-

```
['__annotations__', '__builtins__', '__doc__',
 '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'numerals']

<module 'numerals' from
'E:\\Pyprograms\\numerals.py'>
```

In the output of `dir()`, we can see that name `numerals` has been added to the current scope after the import statement. Printing the identifier `numerals` shows that it refers to a module object. Other names in the output of `dir()` are there by default when a script is executed. If you import the `numerals` module with an alias, the module object is bound to that alias.

```
import numerals as num

print(dir())

print(num)
```

Output-

```
['__annotations__', '__builtins__', '__doc__',
 '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'num']

<module 'numerals' from
'E:\\Pyprograms\\numerals.py'>
```

Python searches for module `numerals`, it creates a module object and binds the name `num` to that object in the current scope. We can see that the name `num` is in current scope, it is referring to the module object `numerals`.

When, a module is imported for a given program run for the first time, the body of the module is executed. During the execution of the module's code, the module object already exists, and as the execution progresses, it gets filled with attributes. All function definitions that are at the top level are executed and the function names become the attributes of the module object. Similarly, if there are any variables defined at the global level in the module or if there are any class definitions then those names also become the attributes of the module object. We can use the `dir` function to see all the attributes of the module object.

```
>>> dir(num)

['__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'factorial', 'is_even',
 'reverse', 'sum_digits']
```

We can see the names of the four functions that we had defined inside the module. All the names defined at the top level of the module become the attributes of the module. Other names that we can see are attributes which are automatically set when the module object is created before the execution of the module body. For example `__doc__` represents the docstring of the module, `__name__` represents the name of the module and `__file__` represents the file from where the module is loaded. Built in modules do not have `__file__` attribute because they are not loaded from a file, they are built into the interpreter. For example, the `math` module will not have this attribute.

```
>>> __name__
'__main__'
>>> num.__name__
'numerals'
>>> num.__doc__
```

```
'This module defines functions related to numbers'  
>>> num.__file__  
'E:\\Pyprograms\\numerals.py'
```

In our example, we have used the `import` statement to import the module. We know that the `from` statement can also be used for importing. The `from` statement also imports the whole module file in a similar way, but it has an extra step at the end, in which all or selected names are copied in the importing scope.

```
from numerals import is_even, factorial  
print(dir())
```

Output-

```
['__annotations__', '__builtins__', '__doc__',  
'__file__', '__loader__', '__name__',  
'__package__', '__spec__', 'factorial', 'is_even']
```

When a module is imported first time in a program, all the top-level commands inside the module are executed. Python places the module object in a special dictionary called `sys.modules`. It contains all the built in modules, and all the modules that you have imported in your script or on interactive prompt. You can print the keys of this dictionary to see the names of the modules.

```
>>> import sys  
>>> sys.modules.keys()
```

Whenever Python encounters an `import` statement, it first looks in `sys.modules` dictionary to see whether the module was already imported. If it was not imported, it looks for a source file in the paths specified in the `sys.path` list. If it was already imported in the program directly or indirectly by any other imported module, there is no effect of the `import` statement, the code inside the module is not executed again.

11.13 Byte compiled version of a module

When you import a module, Python will create a .pyc file for the module which is the compiled Python file. It is the byte compiled version of the module and you can generally see this file in `__pycache__` subdirectory which is created in the same directory as the .py file. This bytecode version is created and stored only for imported modules and not for the main script. Python creates this byte compiled version so that it does not have to compile the file every time the module is loaded by different programs. If an up-to-date .pyc file exists for a module, Python will use that for loading the module instead of the .py file. If the .pyc does not exist or is out of date or was created from a different version then Python will load the module from the .py file and regenerate and save the new compiled version. This is an automatic process and is done to speed up any loading of modules required in future. You can see the `__pycache__` subdirectory in the library directories also, it contains the compiler version of library modules.

11.14 Reloading a module

Each module is loaded into memory only once during an interpreter session or during a program run, regardless of the number of times it is imported into a program. If multiple imports occur, the module's code will not be executed again and again.

Suppose during an interactive session, you have imported a module, and the code of the module is changed while you are using these modules. You might want to use the updated module code by importing it again, but this is not possible since any imports that are done after the first import just use the already loaded module object, the module is not reloaded and its code is not executed again. You have to restart the interpreter session or execute the program again to reload the module. However, you can force a reload by using the `reload` function from the `importlib` module. This way we can get the updated version of the already loaded module without exiting the interpreter session.

```
>>> import module1
>>> from importlib import reload
>>> reload(module1)
```

11.15 Scripts and modules

Python programs consist of .py files that contain code. There is one top level file called the script or the main module (`myprogram.py` in the figure), it contains the main program, and there are other files called modules which can be used in this main file. Here is the structure of a typical multifile program in Python:

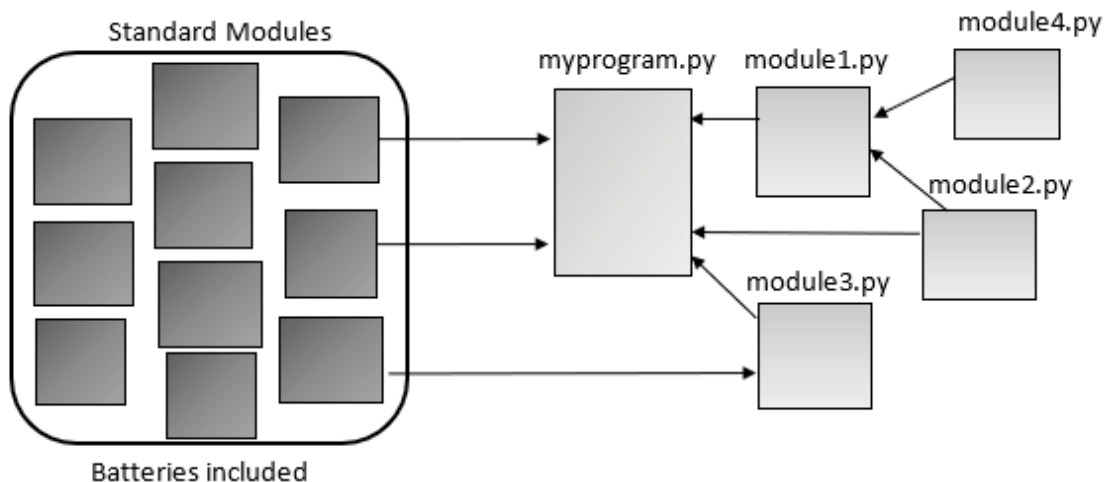


Figure 11.2: Multifile program in Python

The script (application file or main module) contains the main control flow of your application, it is the file that you run to start your application. So, when you launch your application using the command line or using the IDLE Run Module menu or F5, this is the file that will be executed from top to bottom. This top-level file or the script uses the code defined in other modules by importing them. A module in turn can import other modules also. For example, `module1` is importing `module2` and `module4`. The standard library modules and the third-party modules can also be used by the main script or any of the modules.

A module is just an ordinary Python file, so it can be imported as well as executed directly. For instance, the user defined modules `module1`, `module2`, `module3`, `module4` in our example can also be executed directly, like a script. A module generally contains reusable code in the form of function and class definitions, but it may contain other runnable code too. There can be situations when you want to use the same file as an executable script and an importable module. An example of this is testing, you might

want to run a module as a script to test the functions inside it. For example, `module1` contains some function definitions, you can add some statements at the end of the module to test those functions. Similarly, you can add testing code to other modules also that you write. You can execute these modules directly as scripts and see the output of your test code. Different modules can be tested individually as stand-alone programs and when the whole system is integrated, they can be imported. However, there is a problem in this approach. First, let us identify the problem with the help of an example and then we will see the solution for that.

-----`prime.py`-----

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    else:
        return True

def primes(x, y):
    return [n for n in range(x, y + 1) if
is_prime(n)]

def twin_primes(x, y):
    tp = []
    for i in range(x, y + 1):
        if is_prime(i) and is_prime(i + 2):
            tp.append((i, i + 2))
    return tp
```

This file named `prime.py` has these three functions defined in it. The function `is_prime` returns `True` if its argument is a prime number otherwise it returns `False`. The function `primes` returns a list of all prime numbers from `x` to `y`, and the last function returns a list of all twin primes from `x` to `y`. We have imported and used this module in the next program:

-----myprogram.py-----

```
import prime
n1 = 10
n2 = 50

print(f'List of primes from {n1} to {n2}')
print(prime.primes(n1, n2))

print(f'List of twin primes from {n1} to {n2}')
print(prime.twin_primes(n1, n2))
```

Output-

List of primes from 10 to 50

```
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

List of twin primes from 10 to 50

```
[(11, 13), (17, 19), (29, 31), (41, 43)]
```

The file `prime.py` is a `.py` file so it can be run directly like a script. When we run it, we will not see any output because there are only function definitions in the file. Now in the file `prime.py`, let us add some code to test the function definitions.

```
print(is_prime(4))
print(is_prime(5))
print(primes(20, 40))
print(twin_primes(3, 61))
```

This code tests the three functions that we have written in the file. Now when we execute the file `prime.py` directly as a script, we will get some output on the screen and the output shows that our functions are working correctly and are giving the expected result. This is the output that we will see:

False

True

```
[23, 29, 31, 37]
```

```
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61)]
```

This way we can add testing code to our modules to test the functions that we have written in it. After modifying our file `prime.py`, when we will execute our file `myprogram.py`, we will get some unexpected output. Here is the output that we will get:

```
False
```

```
True
```

```
[23, 29, 31, 37]
```

```
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61)]
```

```
List of primes from 10 to 50
```

```
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
List of twin primes from 10 to 50
```

```
[(11, 13), (17, 19), (29, 31), (41, 43)]
```

Before getting the real output of `myprogram.py`, we get some unwanted extra output which is the result of importing the module `prime`. When a module is imported, everything that is at indentation level zero in that module, is executed. So, when we imported the module `prime`, the three `def` statements in it were executed and then the `print` function calls were also executed.

We want the `def` statements to be executed when we import the module because we will need the function definitions, but we do not want the `print` calls (test code) to be executed.

We will have to delete the test code from our module to avoid this problem. But the test code is important because in future we might make changes in the function definitions and then we will have to run these tests again. If we delete them, we will have to write them all over again. It seems that we will have to write a separate testing file for each module, but it is not required because Python has an elegant solution to this problem.

Our problem is that we want the testing statements to be executed only when the file is run as a standalone script, and not when the file is imported as a module. The solution is to place the testing code inside an `if` statement:

```
if __name__ == '__main__':  
    print(is_prime(4))  
    print(is_prime(5))  
    print(primes(20, 40))  
    print(twin_primes(3, 61))
```

Now when we run the file `myprogram.py` in which we have imported the `prime` module, these `print` calls of the `prime` module will not be executed. When we execute the file `prime.py` directly as a script, these calls will be executed. We got the solution to our problem, now let us see why this solution works.

We have seen earlier that the module object has some built in attributes, which includes `__name__` that represents the name of the module. Python will set the value of this variable depending on how the code of the module is executed. If the module's code is executed because it has been imported, Python initializes `__name__` with the name of the module and if a module is run as a standalone script, `__name__` is initialized to `__main__`.

When our file `prime.py` is executed as a standalone script, `__name__` is equal to `__main__`, the `if` condition is `True` and so the `print` calls execute. When the file `prime.py` is used for importing, `__name__` is equal to `prime`, the `if` condition is `False` and so anything written inside the `if` construct will not execute. Anything that is at the top level of the file and not inside this `if` construct, will always be executed whether the file is imported or executed. So, the function definitions that we have in our file `prime.py` will be executed whether the file is executed as a script or imported as a module.

So, if you want to place any testing code that should not be executed when the module is imported, you can place it at the bottom of the file inside the `if` statement with condition `__name__ == '__main__':`. This idiom is commonly applied when you want to use a Python file both as an importable

module and an executable script. You do not need to write a separate file for testing the module.

It is also a common pattern to define a function that contains all the testing code, and call that function inside the `if` statement.

```
def main():  
    print(is_prime(4))  
    print(is_prime(5))  
    print(primes(20, 40))  
    print(twin_primes(3, 61))  
if __name__ == '__main__':  
    main()
```

11.16 Packages

When there are many function and class definitions in our program, we organize them in different modules. When there are many modules in our program, we can organize them in packages. We can place related groups of modules in separate packages. As the concept of directories in an operating system helps us organize our files, the concept of packages helps us organize modules in a hierarchical directory structure which Python can recognize and import. Organizing our modules in packages helps avoid conflicts between module names.

A package is just a directory that contains modules and a file called `__init__.py`. A package can contain other packages also which are sometimes referred to as subpackages. The file `__init__.py` may be empty or it can contain some comments or initialization code for the package. This file will be executed when the package or its contents are imported.

To define a package, create a directory that has the same name as the package and then create a file `__init__.py` in that directory. You can place your modules in this directory. The name of the package should be a valid Python identifier.

11.17 Importing a package and its contents

Physically, package is a directory that contains modules and other packages. Conceptually a package is just a module that contains other modules. For the user, there is not much distinction between a package and a module, because the same syntax is used for accessing and importing. For the user, a package is very much like a module, and the modules and subpackages present inside it are like module attributes that can be accessed using the dot.

Here is an example of a package that we will use to demonstrate importing:

```
pkg/  
    __init__.py  
    module1.py  
    module2.py  
    module3.py  
    module4.py
```

We can import individual modules from a package by using the from statement.

```
-----myprogram.py-----  
  
from pkg import module1, module2  
  
module1.func1()  
  
module2.f1()
```

The names of modules are introduced in the current scope. We can use the functions defined inside the module by prefixing them with the module name. Python will be able to locate the package only if it is present in one of the directories contained in the `sys.path` list.

An alternative way of importing individual modules is by using the dot syntax.

```
import pkg.module1, pkg.module2  
  
pkg.module1.func1()
```

```
pkg.module2.f1()
```

We have to use the `pkg` prefix whenever we need to access `module1` or `module2`. When we used the `from` statement to import the modules, there was no need to add this prefix as the names of modules were introduced in the current scope.

We can use the `from` statement to import individual names from a module that is inside the package.

```
from pkg.module1 import func1  
  
func1()
```

Now, the name of the function is in the current scope so there is no need to prefix it with the package name and module name. However, this can create problem if some other module or package also uses the same name, so it is better not to import names directly unless you are sure that there will be no name conflict.

We can use aliasing to provide short names for fully qualified names that are long, however this increases the chances of name collisions.

```
import pkg.subpkgA.moduleX as modX
```

It is better to keep the hierarchy flat so that the users can access the required items without qualifying them with too many names. It is not good to keep your API (modules that users need to access) too deep inside the package.

You can also import the package by using just the package name in the import statement.

```
import pkg
```

Importing the package like this does not import the modules inside it automatically. This only brings the name `pkg` into the current scope and imports `__init__.py` file from the package directory, so the code inside this file is executed. If your design expects your package to be imported like this, you can place code to import individual modules in the `__init__.py` file.

```
----- __init__.py -----  
  
import pkg.module1
```

```
import pkg.module2
```

Now when `import pkg` is encountered in your program, the file `__init__.py` inside it will be executed and it will import only `module1` and `module2` from the package. This way the author of the package can decide which modules should be imported when a complete package is imported. Modules that are used for internal purposes can be hidden from the user, and only API can be exposed. You can also use `__init__.py` to expose any object defined inside modules like functions or classes.

It is also possible to import a package or its modules with an alias using the `as` keyword. For example, the NumPy package is often imported with name `np`.

```
import numpy as np
```

The wildcard character can also be used for importing.

```
from pkg import *
```

We have seen this `import` statement at the module level and there it imported everything from the module that did not start with an underscore. The behaviour of this statement at package level is different; it does not import any module present inside the package. If there is a `__all__` list defined inside the `__init__.py` file, then only this statement will import the modules present in that list. For example, suppose the following statement is present inside `__init__.py`.

```
__all__ = ['module1 ', 'module2']
```

The modules `module1` and `module2` will be imported when `import *` syntax is used, other modules inside the package will not be imported. `import *` is not a recommended style, the safe approach is to import module names directly from the package. Module names can be prefixed with an underscore to indicate that they represent some internal implementation details, and should not be imported.

The `__init__.py` file inside the package will be executed whenever the name `pkg` appears first time in an import, i.e. when anything is imported from the package or when the complete package is imported. You can put any initialization code that has to be executed once. As we have seen we can use this file to present an API to the user while hiding the internal details.

11.18 Subpackages

A hierarchical structure of packages and subpackages can help you to organize modules of your project and avoid import name conflicts. Here is the hierarchy of a package that contains two subpackages. Each package and subpackage contains the file `__init__.py`.

```
pkg/  
    __init__.py  
    module1.py  
    module2.py  
    subpkgA/  
        __init__.py  
        moduleX.py  
        moduleY.py  
        moduleZ.py  
    subpkgB/  
        __init__.py  
        moduleP.py  
        moduleQ.py  
        moduleX.py
```

The syntax that we have seen for packages applies for subpackages, too. Now, we need an additional dot to access the subpackage. For example, to `import moduleX` from `subpkgA` you can write one of the following import statements:

```
from pkg.subpkgA import moduleX  
  
import pkg.subpkgA.moduleX
```

When writing import statements such as `import item1.item2.item3.item`, each item should be a package, except for

the last one. The last item can be a module or a package, but it cannot be a function or a class defined inside the previous item. For example, you cannot write the following statement to import function `func1` from `moduleX`.

```
import pkg.subpkgA.moduleX.func1 # can't write
this
```

We have to use a `from` statement to import particular names from inside modules.

Modules in different packages can have same names, for example both `subpkgA` and `subpkgB` have a module named `moduleX`, and both are completely different. Since they are accessed using the dot syntax, the distinction is clear.

If we import the package using the `import pkg` statement, the subpackages will not be automatically imported. They will be imported if there is importing code in the `__init__.py` file present inside `pkg`. The `__init__.py` can include statements such as-

```
----- __init__.py inside pkg -----
import pkg.module1
import pkg.subpkgA.moduleX
import pkg.subpkgB
```

11.19 Relative imports

The importing that we have done till now is absolute importing. When we have a complex package with a hierarchical structure that includes many subpackages and modules, we can use relative importing in the `from` statement to import modules that are a part of the same package. For example, inside `moduleQ`, suppose we want to import three modules: `moduleP` which is in the same package as `moduleQ`, `module1` which is in the parent package, and `moduleZ` which is in a sibling package. We can import them using either the absolute import or by using the relative import in the `from` statement.

```
-----moduleQ.py-----
```

```
from . import moduleP
from .. import module1
from ..subpkgA import moduleZ
```

A single dot after the `from` keyword represents the package of the current module, double dot represents the parent package and similarly each additional dot steps up one package. In the above example, `.` represents the package of `moduleQ`, `..` represents the parent package of `moduleQ` and `..subpkgA` represents the subpackage `subpkgA` of parent package. These statements are equivalent to the following absolute imports:

```
from pkg.subpkgB import moduleP
from pkg import module1
from pkg.subpkgA import moduleZ
```

Relative imports can be done only for importing inside a package and can be applied only to the `from` statement, they cannot be applied to `import` statements. The main module that is intended to be executed as script should always use absolute imports because its name is `__main__`, and in relative imports the importing is done based on the name of the current module.

Absolute imports are more readable and descriptive but tend to become lengthy and verbose in complex packages. By using relative imports, you can do intra-package accessing without hardcoding the package name. This can be beneficial if in future the top-level package name is changed or the structure is reorganised.

If you have some code, like function definitions or constants that need to be used by all modules in the package, you can make a common module that contain all these objects that have to be shared. This file can be imported in all the modules of the package.

Exercise

1. What is a module?
 - A. A built-in function

- B. A data type
 - C. A Python file containing code
2. How do you create a new module in Python?
- A. Use the `create_module()` function
 - B. Save a file with `.py` extension
 - C. Save a file with `.mod` extension
3. How do you import all names from a module in Python?
- A. `import all module_name`
 - B. `from module_name import *`
 - C. `from module_name import all`
4. Which of these statements can be used to import individual names from a module in Python?
- A. `import name1, name2 from module_name`
 - B. `from module_name import name1, name2`
5. Which keyword creates an alias while importing a module in Python?
- A. `rename` C. `with`
 - B. `as`
6. Which function is used to reload a previously imported module in Python?
- A. `reload()` C. `refresh()`
 - B. `update()`
7. Can you use a file named `date-time.py` as a module.
- A. Yes B. No
8. What is the purpose of the `__all__` variable in a module?
- A. It stores all the names defined in the module.
 - B. It stores the names that should be imported using the statement `from module import *`

C. It stores the names that should not be imported using the statement
`from module import *`

9. The code written inside the code block of `if __name__ == '__main__':` will be executed when

A. The file is executed as a script

B. The file is imported as a module

10. The statement `from pkg import *` will import all the modules in the package

A. True B. False

11. The statement `import pkg.subpkgA.module1.func` will import the function `func` present inside `module1`.

A. True B. False

12. Which keyword is used to import a module from a package in Python?

A. `in` C. `from`

B. `with`

13. How do you import a module from a package ?

A. `import module_name from package`

B. `from package import module_name`

14. If the module's code is executed because it has been imported, Python initializes `__name__` with

A. `__main__`

B. name of the module

15. Absolute import involves importing modules based on their relative location to the current module.

A. True B. False

16. Which one of these shows a relative import?

A. `from ..subpkg import moduleX`

```
B. from pkg.subpkg import moduleX
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



12.1 Namespaces

We know that everything in Python is an object. Strings, lists, dictionaries, and even functions and modules are objects. All these objects are identified and accessed by names defined in the program. As your program grows larger, the number of names in the program will increase, which increases the chances of name clashes. For example, the name `message` may be defined in two separate functions or in two separate modules. Python creates and uses namespaces to manage all the names in a program and avoid any name collisions. It keeps track of all the names by implicitly adding them to different namespaces, mapping each name to its corresponding object. This concept of namespaces allows us to use the same name simultaneously for different objects in different parts of our program, without causing any name conflicts.

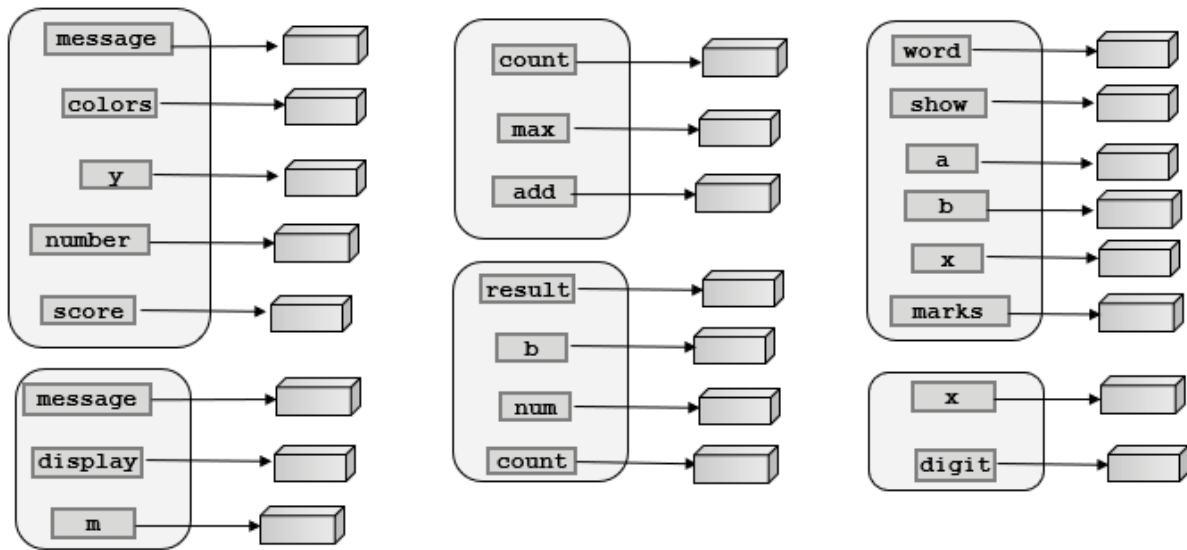


Figure 12.1: Namespaces

You can think of a namespace as just a space for mapping names to objects. Each name in your program lives inside a specific namespace. These namespaces are automatically created at different moments during the execution of a program. At any instant, while the program is running, multiple namespaces can be active. These namespaces are independent and completely isolated, so we can have the same name in two or more namespaces without a problem. Whenever you define a name, Python will store the name object binding in one of these namespaces, and whenever you use a name in your program, it will be searched in one of these namespaces. These namespaces serve as lookup tables for names.

All the names in a namespace will be unique, but in different namespaces, names can be the same. For example, the name `message` can be present in two namespaces, but both names will be different; they can be bound to different objects. Due to the concept of namespaces, there are very few chances of name clashes, there will be a name clash only if a name appears more than once in the same namespace.

When we define a name, the name object binding will go to a namespace. The particular name object binding goes to which namespace will depend on where we have defined the name. Now, let us see what namespaces Python creates at run time.

When the interpreter is invoked, a built-in namespace is created and it exists until you exit Python. This namespace contains predefined built-in names such as `print`, `id`, `input`, `int`, `max` and many built-in exception names. The built-in namespace exists until the interpreter terminates and this is why we can use these names in our program anytime and anywhere.

When you execute your script, a global namespace is created that contains all the names that you define at the top level of your executing script. Some default dunder names are also automatically included in this namespace by Python. The global namespace also remains in existence until the interpreter is terminated.

Local namespaces are created when functions are called. Each function call introduces a new local namespace and it exists only till the function is running. The local namespace of a function includes the function's parameters and any other names that are defined within the body of the function. A local namespace is deleted when the function's execution is finished, all the name object bindings in it are forgotten. Next time when the function is called, a fresh namespace will be created. So, local namespaces are created when required and are deleted when no longer needed. Note that a local namespace is created when a function is called, not when it is defined. Let us take an example and see how the namespaces are created.

-----test.py-----

```
message = 'Hello'
def add(a, b):
    s = a + b
    print(s)
x = 500
def func(a, b, c):
    print(message)
    s = a + b*c + x
    print(s)
add(1, 2)
func(4, 5, 3)
```


The built-in namespace is there for every Python program. When the program starts executing, a global namespace will be created.

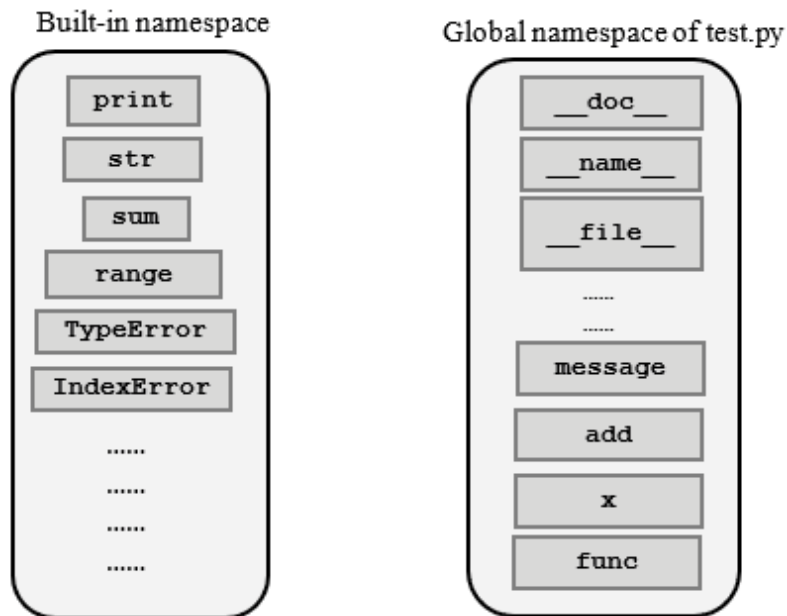


Figure 12.2: Built-in namespace and global namespace

The name `message` is assigned at the top level of the file so it is put in the global namespace. After this, the `def` statement executes; it creates a function object and assigns it to the function name. This adds one more name in our global namespace. Now, the variable `x` is defined at the top level, so we have `x` also in global namespace, after this the `def` statement executes which adds the name `func` in the global namespace.

When the call to function `add` is executed, a local namespace is created. This namespace contains parameters `a` and `b` and the variable `s` that is defined inside this function.

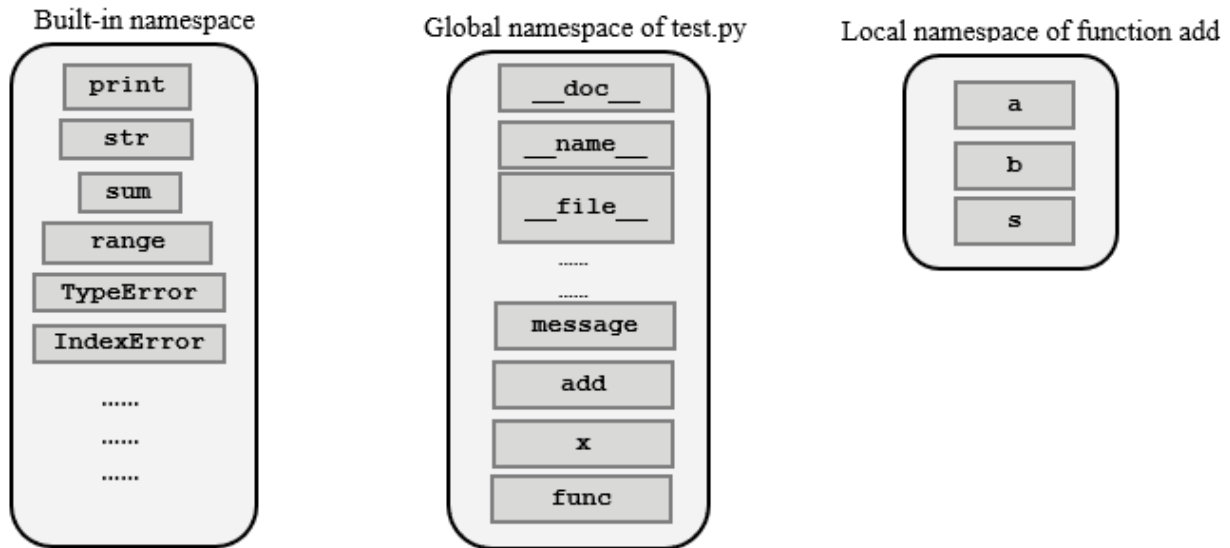


Figure 12.3: Built-in namespace, global namespace and local namespace

When the function finishes execution, this local namespace vanishes and the names `a`, `b` and `s` do not exist anymore. Any objects that they are referring to will be garbage collected if those objects are not referenced anywhere else, otherwise the reference count for those objects will be decreased by one.

After the function `add` finishes execution, the function `func` is called. A new local namespace is created which includes the three parameter names `a`, `b`, `c`, and the variable `s`. When the function execution is over, this local namespace will also be destroyed.

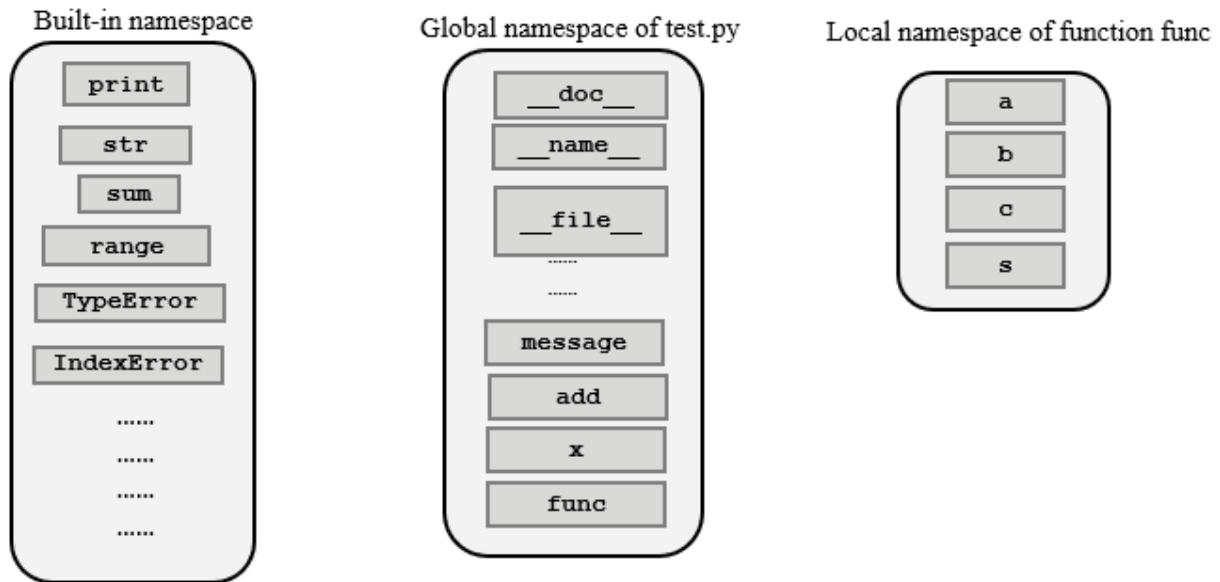


Figure 12.4: Built-in namespace, global namespace and local namespace

Each module has its own global namespace, that is why global names defined in one module do not interfere with global names defined in another module. Global namespace for a module is created when the module is first imported and normally it also lasts until the interpreter quits. Global namespace of a module consists of all the names defined at the top level of that module. Now, suppose in your program, after the call to function `func`, you import a module by using the `import` statement:

```
import prime
```

When this `import` statement executes, the name of this module is introduced in the global namespace of the file where the module is imported, and a separate global namespace for this module is created which contains all the names defined at the top level of this module. The module namespace will normally last until the interpreter quits.

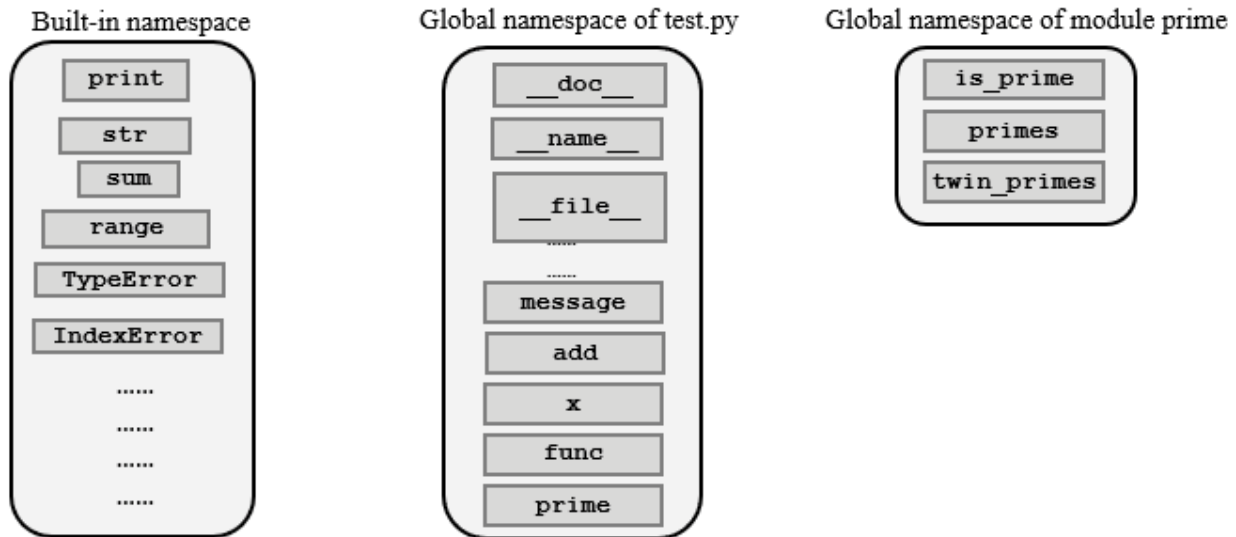


Figure 12.5: Built-in namespace and global namespaces

Each imported module has its own global namespace which is separate from the global namespace of the main module. If the importing module needs to use any name from any of these global namespaces, the name has to be prefixed with the module name. We have already seen this in the previous chapter. So, there can be many global namespaces when your program is executing. One global namespace that will always be there is the namespace corresponding to the `__main__` module, which is your main module (executing script), and there may be other global namespaces, each corresponding to an imported module. The namespace that belongs to your main module is created when the program starts executing, and a module namespace is created when it is first imported.

In the previous chapter, we saw the `from` statement that is used to import specific names or all names from a module. Now suppose we write these two statements in our program:

```
from words import reverse, count
from math import *
```

These `from` statements will create separate global namespaces for the two modules and they will also insert the imported names into the global namespace of importing module. So, from the module `words`, two names, `reverse` and `count`, are included directly in our global namespace and from the module `math`, all the global names are included, and this is why

we can use all these names directly without qualifying them with the module name. This makes the global namespace of our current module crowded and can lead to overwriting of existing names in the case of name clashes. This is the reason why it is considered a bad practice to use this form of import as it pollutes the global namespace of the importing module. Note that the `from` statement does not insert the name of the module into the global namespace of the importing module.

We know that the name of the module for our executing script is `__main__`. Any code that you type at the interactive prompt is also considered part of the module `__main__`, all names that you define interactively are global variables that are available in the whole interactive session. They live in the global namespace of `__main__` module, when you restart the session, this global namespace will be recreated. You must have noticed that after we run the program, the global names of our program are available on the interactive prompt until we restart.

When we will study about classes and objects, we will see that each class and each object have its own namespace to store the attribute names.

12.2 Inspecting namespaces

Local and global namespaces are usually implemented through dictionaries, where names are keys and values are corresponding objects to which the names are bound. Built-in namespace is implemented with the help of a module. The built-in names live inside the standard library module named `builtins`. We can import this module and use the `dir` function to see all the predefined built-in names.

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError',
..... 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

The `dir` function can be used to get the names in a global or local namespace. This function gives us just the keys of the dictionary, to get the full dictionary we can use the `vars` function. Without arguments, the functions `dir` and `vars` work on the most locally enclosing scope in which they are executed.

```
x = 3
def func(a):
    y = 10
    print(dir())
    print(vars())
func(5)
print(dir())
print(vars())
```

Output-

```
['a', 'y']
{'a': 5, 'y': 10}
[....., 'func', 'x']
{....., 'x': 3, 'func': <function func at
0x00000262D456E3E0>}
```

These functions can also accept an argument such as a module, class or object name and will return names in that context. To access the global namespace of an imported module, we can send the module name to these functions as arguments. The `__dict__` attribute of a module also gives access to the module's namespace dictionary.

There are two built-in functions called `locals` and `globals` that can be used to examine the names contained in local and global namespaces. `globals()` returns a dictionary that contains names in the global namespace of the module and `locals()`, when placed inside a function, returns a dictionary that contains local names accessible from that function.

```
x = 3
def func(a):
    y = 10
```

```

    print(locals())
func(5)
print(globals())

```

Output-

```

{'a': 5, 'y': 10}
{..... 'x': 3, 'func': <function func at
0x000001B91D72E3E0>}

```

If we want to retrieve only the names, we can use the `keys` method.

```

print(locals().keys())
print(globals().keys())

```

If `locals()` is called outside a function at the top level of the program, it behaves like the `globals()` function. If `globals()` is called inside a function, it returns a dictionary that contains all names that can be accessed globally from that function. If the function is defined in a separate module, it gives the global names of the module where the function was defined and not the global names of the module where the function is called.

Now let us inspect the global namespace after importing modules.

```

x = 3
def func(a):
    y = 10
    print(locals())
func(5)
print(globals())
import prime
from words import *
print(globals())

```

Output-

```

{'a': 5, 'y': 10}
{'__name__': '__main__', '__doc__': None, .....
'x': 3, 'func': <function func at
0x00000232C9122480>}

```

```
{'__name__': '__main__', '__doc__': None, .....
'x': 3, 'func': <function func at
0x00000232C9122480>, 'prime': <module 'prime' from
'E:\\Deepali\\prime.py'>, 'count': <function count
at 0x00000232C9122980>, 'first': <function first at
0x00000232C9122A20>, 'last': <function last at
0x00000232C9122AC0>, 'ordered': <function ordered
at 0x00000232C9122B60>, 'reverse': <function
reverse at 0x00000232C9122C00>}
```

We can see the module name `prime` has been added to the global namespace and all the names of the module words have also been added.

12.3 Scope

A name cannot be accessed from just anywhere inside a program. Every name-object binding has a scope and this scope determines the part of the program where you can access that particular name without using any prefix. Scope of a name depends on where it has been defined inside the file. Names that are assigned outside all functions, at the top level of a file, have global scope and they can be accessed throughout the file. Names that are assigned inside a function have local scope, and these names can be accessed only inside the function in which they are defined.

```
message = 'Hello'
def add(a, b):
    print(message)
    s = a + b
    print(s)
x = 500
def func(a, b, c):
    print(message)
    s = a + b*c + x
    print(s)
print(message)
```



```
print(x)
add(1, 2)
func(4, 5, 3)
```

Here in this program, the names `message` and `x` are defined outside any function so they have global scope, they can be accessed anywhere inside the file: inside any function or outside functions. They are global variables. The function names `add` and `func` also have global scope, they are visible throughout the file.

The names `a`, `b` and `s` inside the function `add` have local scope, they can be accessed inside this function only. If we try to use any of these names outside the function, we will get an error because these names are visible only inside the function. Similarly, the names `a`, `b`, `c` and `s` inside the function `func` have local scope, they are visible only inside the function, they cannot be accessed anywhere outside. All these variables are local variables.

The names `a`, `b`, and `s` inside the `add` function are different from the names `a`, `b`, and `s` of function `func`. Although they have the same names, they are separate variables. They have nothing to do with each other because we know that they live in separate namespaces. These variables cannot be accessed outside their function.

In Python, global scope means just the file scope (or the module scope). We can have programs that involve various files, but the names defined globally in one file will not be visible to other files. This is why we can define variables with the same names in different modules without name conflict. So, global scope in Python is just file scope or module scope, and global variables are just module-level variables.

Names with global scope live in the module's global namespace, and names with local scope live in their own local namespace. This concept of scope and namespace makes sure that variables with same names can appear inside different functions or different modules without any name conflict. However, if possible, you should avoid using the same name in different scopes, as it can sometimes be confusing. Overusing global variables is also discouraged in larger programs as it can lead to less readable code and hard to trace bugs.

Since global variables can be modified and accessed anywhere inside the file, it is difficult to understand a portion of the program in isolation. Fixing bugs caused by the wrong value of global variables would be difficult, as you would have to examine all the places where it is changed, and that place could be any part of your file. For a local variable, there is only a portion of the program where it can be changed. So, global variables should be used only in situations when they are absolutely necessary. Functions should generally communicate with each other with the help of arguments and return values instead of using global variables.

There is one more scope called the nested scope or enclosing function scope. The nested function scope comes into picture when functions are nested i.e., when a function is defined inside another function definition. In the following example, we have a function `f()` defined inside the definition of function `func()`:

```
z = 10
def func():
    x = 10
    y = 20
    def f():
        a = 5
        print(a)
        print(x, y)
    f()
func()
```

Output-

```
5
10 20
```

We know that `def` statement is an executable statement, so it can be written at any place where a Python statement can be written and therefore it is valid to write a `def` statement inside another `def` statement.

In the function `func()`, we have defined two variables `x` and `y` and then we have defined the function `f`. After that, we have called the function `f` inside

`func()`, and then `func` is called at the top level.

When the function `func()` is called, the `def` statement inside it is executed, it creates a function object and assigns it to name `f`. Now this name `f` is in the local scope of the function `func`. It is available only till this function is executing, when the function call terminates, the inner function is no longer available. So, we can call this function `f` only from inside the function `func`. It cannot be used anywhere else in the program.

The function `f` defines a variable `a` and prints its value, and it also successfully prints the values of `x` and `y` which are defined in the outer function. This shows that the names defined in the outer function are available in the inner function. If a name is defined in a function, then the scope of that name extends to all the inner functions. The code in the inner functions can access local variables defined in the outer function. This is nested scope or enclosing function scope.

List, set and dictionary comprehensions and generator expressions (we will talk about them later) have their own local scope. Variables defined in these expressions are not available outside these expressions. This is different from what happens in a `for` loop statement; variables defined in a `for` loop are available even after the loop finishes.

```
for i in [1, 2, 3]:
    print(i)
print(f'i is {i}')
squares = [x * x for x in range(1,4)]
print(x)
```

Output-

```
1
2
3
i is 3
NameError: name 'x' is not defined
```

Variables assigned within the comprehension expressions are local. However, these expressions can access the variables in the surrounding

scope. The names local to these expressions do not mask the names in the surrounding scope.

```
x = [1, 2, 3]
squares = [x * x for x in x]
print(squares, x)
```

Output-

```
[1, 4, 9] [1, 2, 3]
```

12.4 Name Resolution

We know that when a program is running, there are multiple namespaces that are active. When we use a name inside the program, Python needs to look for that name in the appropriate namespace and fetch the object that it is referring to. Python uses the concept of scope to search the namespaces. Scope determines the namespaces that are accessible for searching. If a name has local scope, then it will be searched in its own local namespace or the local namespaces of enclosing functions, or in global and built-in namespaces. If a name has global scope, then it will be searched in global and built-in namespaces. This process is called name resolution. Python follows a rule for name resolution commonly known as LEGB rule. This rule specifies the order in which namespaces are searched while looking for a name.

```
L : Local namespace
E : Enclosing local namespaces(if any)
G : Global namespace
B : Built in namespace
```

The LEGB rule is named after the first letter of the different namespaces that Python will search. If the name that you are using has local scope, then Python first looks for the name in its own local namespace, then in the local namespaces of the enclosing functions starting from the nearest enclosing function (if there are any), followed by the global namespace of the current module, and then finally in the built-in namespace. It stops the search at the first place where the name is found. If the name has a global scope, it is first searched in global namespace and then in the built-in namespace. If the

name is not found in any of the namespaces, then a `NameError` is raised. Let us understand this with the help of a simple example.

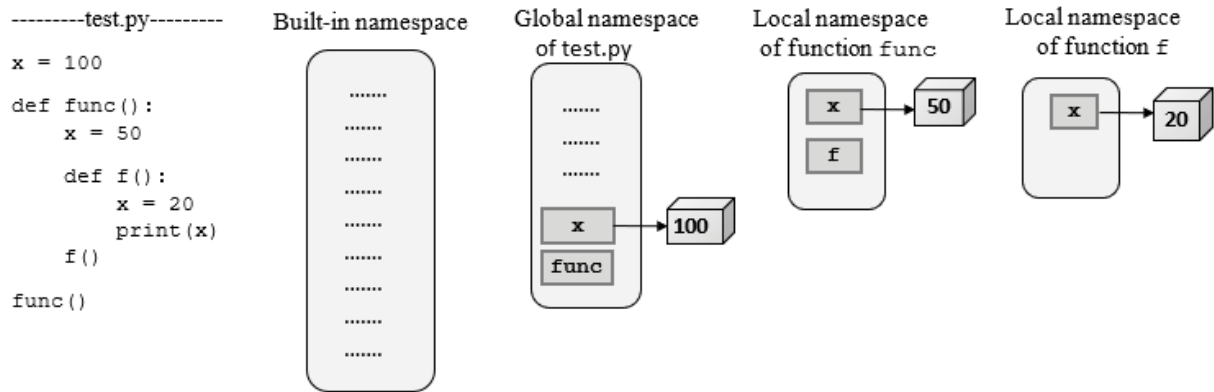


Figure 12.6: Built-in, global and local namespaces

In this small program, we have `x` as global variable, then we have a function named `func` which also has a variable `x` and there is another function `f` defined inside `func`. This function `f` also has defined a variable `x` and we are printing `x` inside this function. After the definition of `f`, we are calling `f` inside `func` and at the end we are calling `func`.

In the global namespace, we have `x` and `func`. In local namespace of `func` we have `x` and `f`, and in local namespace of function `f` we have the name `x`. So, at the time of execution of `print(x)`, all these namespaces will be existing and all of them contain name `x`, the name is same but they refer to different objects. Due to the concept of namespaces, there was no name clash, each `x` was put in a separate namespace depending on the place where it was assigned.

All the three `x` are accessible inside `f` so Python will follow the LEGB rule when it sees `x` in the `print` function. First, it will search in the local namespace of function `f`, it finds the name `x` there so it will use that binding and print 20.

Now suppose we delete the statement `x = 20` from the function `f`, so now `x` is not there in the local namespace of `f`. Now when Python will execute the `print` function, it will first try to find `x` in local namespace. It does not find it there, so it looks in the namespace of enclosing function. `x` is there inside `func`, so it uses that `x` and prints 50.

Now suppose we delete the statement `x = 50` from `func`. Now when Python will execute the `print` function, it will first look in local namespace, it is not there then it will look in the enclosing function's namespace, it is not there also, so it goes one more level up and looks in the global namespace, it finds `x` there so it prints 100.

Now, suppose we delete the statement `x = 100` also. Now when Python will execute the `print` function, it first looks for `x` in the local namespace and does not find there. It, then, looks in the enclosing function's namespace, does not find there either, and then looks in the global namespace. It is not there too, so it looks in the built-in namespace, `x` is not there in this namespace also so it shows `NameError`.

<pre>x = 100 def func(): x = 50 def f(): x = 20 print(x) f() func() Output- 20</pre>	<pre>x = 100 def func(): x = 50 def f(): print(x) f() func() Output- 50</pre>	<pre>x = 100 def func(): def f(): print(x) f() func() Output- 100</pre>	<pre>def func(): def f(): print(x) f() func() Output- NameError: name 'x' is not defined</pre>

Figure 12.7: Name Resolution

So, this is how Python does name resolution by following the LEGB rule. A consequence of this rule is that local names can mask global and built-in names and global names can mask built-in names. If you reassign any of the built-in names in your program then you will lose the original functionality of that name. Let us see this with the help of an example:

We know that there is a built-in function `max` that can be used to find out maximum value.

```
print(max(1, 2, 3, 4))
print(max([1, 2, 3]))
```

Output-

4
3

Now let us define our own `max` function in the file.

```
def max(x, y):  
    if x < y:  
        print('Maximum value is ', y)  
    elif x > y:  
        print('Maximum value is ', x)  
    else:  
        print('Both are equal')  
max(1, 2, 3, 4)  
max([1, 2, 3])
```

Output-

`TypeError: max() takes 2 positional arguments but 4 were given`

Now these calls do not work, because the original built-in version of `max` is hidden. The version that we have defined takes two arguments so these calls fail.

When the interpreter tried to execute the call, it looked for the name `max` according to the LEGB rule. The statement is not there inside any function so there is no local scope or enclosing scope to search. It starts with the G of LEGB and finds the name `max` in the global namespace so the search is stopped and our version of `max` is used.

If we delete our definition of `max`, then Python first looks in the global namespace, does not find it there so goes to the built-in namespace and finds it there and uses the built-in version. It is generally not a good idea to redefine the built in names, but sometimes you may do it to customize the way things work.

12.5 global statement

The `global` statement allows you to create or change a global variable from within a function. Let us understand with the help of examples how it works and why is it required:

```
x = 100
```

```
def func():  
    print(x)  
func()  
print(x)
```

Output-

```
100  
100
```

In this code we have a global variable `x` whose value is 100. We have defined a function `func` inside which we are printing `x` and then we have called the function `func`. In the output, we can see that the value of the global variable `x` was printed by the function. There is nothing new in this, we already know that a global variable can be accessed from any function. These types of variables that are used in a code block in which they are not defined are called free variables.

Now before printing `x`, let us assign another value to it inside the function.

```
x = 100  
def func():  
    x = 30  
    print(x)  
func()  
print(x)
```

Output-

```
30  
100
```

The `print` call that is outside the function prints 100 which shows that the global variable `x` has not been reassigned. When the assignment statement that we have placed inside the function was executed, it created a new local variable in the function. We can check this by printing the values of `locals()` and `globals()` function.

```
x = 100  
def func():
```



```

x = 30
print(x)
print(locals())
print(globals())
func()
print(x)

```

Output-

```

30
{'x': 30}
{ ....., 'x': 100 , .....}
100

```

A new name `x` was introduced in the local namespace of the function. The assignment inside the function did not rebind the global variable, it created a new local variable.

Now let us put the assignment after the `print` call inside the function.

```

x = 100
def func():
    print(x)
    x = 30
func()
print(x)

```

By looking at this code, it seems that first it will print global `x` and then create the local `x`, but it gives the following error:

```

UnboundLocalError: cannot access local variable 'x'
where it is not associated with a value

```

This means that `x` cannot be both global and local inside the function. If a name is assigned a value anywhere inside the function, then that name is considered to be a local variable inside the whole function. There are no variable declarations in Python, it just assumes that any variable assigned anywhere in the function is local. Using a variable before it has been assigned results in an error. If we want to reassign the global `x` inside the

function, we have to inform Python about this by writing the `global` statement. In the global statement, we write the `global` keyword followed by the name of the variable.

```
x = 100
def func():
    global x
    x = 30
    print(x)
    print(locals())
    print(globals())
func()
print(x)
```

Output-

```
30
{}
{ ..... , 'x': 30 , ..... }
30
```

By writing this `global` statement, we are telling the interpreter that we want to use the global variable `x`, so do not create any local variable with this name. We can see that the `locals` dictionary is empty, no local `x` was created and the global variable was actually reassigned.

The `global` declaration is a namespace declaration which indicates that the specified name lives in the global namespace and should be rebound there instead of introducing a new name in the local namespace.

It is possible to specify more than one global variable by using the same `global` statement. So, we could specify more global variables by using a comma.

```
global x, y, z
```

You can freely use a global name inside a function, but if you want to reassign a global variable inside a function, you need to declare it global by

writing the `global` statement. Without the `global` declaration, the assignment will create a new local variable.

This requirement of a `global` declaration for reassigning a global variable is actually good, otherwise you might unknowingly change a global variable leading to problems. This could happen if you are unaware of a global variable that has the same as the local variable that you are creating inside the function, you would think that a local is being created but actually it will the reassign the global variable.

The `global` declaration is required only if you have to reassign the global variable; a mutable global variable can be changed in-place inside the function without the `global` declaration.

```
numbers = [10, 20]
def func():
    numbers.append(30)
    print(numbers)
func()
print(numbers)
```

Output-

```
[10, 20, 30]
[10, 20, 30]
```

Although the `global` declaration is not required for accessing or mutating a global variable inside a function, it is good to write the declaration in these cases also as it provides clarity to the reader of the program. The `global` declaration makes it explicit that a global variable is being used.

We can also use the keyword `global` to create a global variable inside a function. Generally global variables are created by assigning to a name at module level code, but it is also possible to create a global variable inside a function by using the combination of a `global` statement and an assignment.

```
def func():
    global y
    y = 10
```

```
func()
print(y)
```

If `y` does not exist in the global scope before the function call, then the assignment will create a new global variable `y` and will give it value 10. If `y` already exists in the global scope, then the assignment will reassign global `y`. So, you can declare a name global even if it does not exist in the global scope. It can be created later by an assignment.

Although it is not a good programming practice to have a local and global with the same name, it is useful to be aware of what happens if such a situation occurs and we need to access both local and global versions inside the function. When a local variable and a global variable both have the same name, the local variable shadows the global variable inside the function. So, inside the function we can access local variable and outside the function we can access the global variable. If we want to access the global from inside the function, we cannot do with the `global` statement because then the name will refer to the global variable, and local will not be available. We can use the function `globals` for this. If we want to refer to a global variable `x` inside the function, we can use `globals()['x']`.

```
x = 5
def func():
    x = 10
    print(x)
    print(globals()['x'])
    globals()['x'] = 20
```

```
func()
print(x)
```

Output-

10

5

20

12.6 nonlocal statement

In the previous section, we saw that the `global` statement allows us to reassign global names. There is a similar statement that uses the keyword `nonlocal` and it allows us to reassign names that are in the enclosing function scope. Like `global` statement, `nonlocal` statement is also a namespace declaration which indicates that the specified variable lives in some enclosing function scope. Let us understand this with the help of an example:

```
def func():
    x = 100
    def f():
        print(x)
    f()
    print(x)

func()
```

Output-

```
100
100
```

The function `func` has a local variable `x`, and inside this function we have defined another function `f`. The inner function `f` just prints the value of `x`.

We know that the code inside the inner function can access local variables defined in the outer function. So, when `print(x)` will be executed, the local variable `x` of `func` will be printed.

Now, let us make a change in our inner function, before printing `x` we will reassign it.

```
def func():
    x = 100
    def f():
        x = 30
        print(x)
```

```
    f()
    print(x)
func()
```

Output-

```
30
100
```

We can see that variable `x` of `func` was not changed. The inner function created a new local variable named `x`. So, when a function is nested inside another function, the inner function can freely use any name defined in the enclosing function, but it cannot reassign it. An attempt to do so creates a local variable. To prevent the creation of a new local variable, we need to declare the variable as `nonlocal` by writing the `nonlocal` statement.

```
def func():
    x = 100
    def f():
        nonlocal x
        x = 30
        print(x)
    f()
    print(x)
func()
```

Output-

```
30
30
```

Now, the inner function was able to reassign the `x` defined in the outer function. So, you can see that the job of `nonlocal` statement is similar to that of `global` statement, only the scopes involved are different. A `global` declaration is required when you need to reassign a global variable, and a `nonlocal` declaration is required when you need to reassign a variable in an enclosing scope.

There are two differences between the `global` statement and `nonlocal` statement. The first difference is that in a `global` statement you can write a variable name even if it does not exist in the global space, while in a `nonlocal` statement you cannot write a variable name that does not exist in an enclosing function.

```
def func():  
    global x  
    x = 10  
  
func()  
print(x)
```

Here `x` is global because it is used in the `global` statement, after that it was created by the assignment statement. It is created inside the function, but it is global. So, you can declare a name global even if it does not exist in the global scope. It can be created later by an assignment.

The behaviour of `nonlocal` is different in this case. You can declare a name `nonlocal` only if it exists in any of the enclosing function scope. You cannot create it inside the inner function.

```
def func():  
    def f():  
        nonlocal x  
        x = 10  
    f()  
    print(x)  
  
func()
```

Here we are trying to create a `nonlocal x` inside the inner function, this will give us an error.

So, any name that is listed in the `nonlocal` declaration, should have been defined in an enclosing function.

The other difference between `global` and `nonlocal` statement is related to the searching of name. If a name is declared `global`, then the search for it starts at the global scope and continues in the built-in scope. If a name is

declared nonlocal, search is not done in the global or built in scopes. It is searched only in the enclosing function scopes.

Exercise

1. A local namespace is created when
 - (A) a function definition is executed.
 - (B) when a function is called.
2. Global scope spans across modules.
 - (A) True (B) False
3. Which statement will you use if you want to assign to a global variable inside a function?
 - (A) nonlocal
 - (B) global
 - (C) return
4. No new scope is introduced by if else, for and while constructs.
 - (A) True (B) False
5. If there is a local variable and a global variable with the same name, then inside the function the _____ variable hides the _____ variable.
 - (A) local, global (B) global, local
6. When a function has finished executing and its local namespace is deleted, all the names in it and the objects that they refer to are deleted.
 - (A) True (B) False
7. How many names are there in the global namespace of the module that contains the following code?

```
x = 4
```

```
y = 5
```

```
def func1():
```



```
pass
def func2():
    pass
```

(A) 2

(B) 4

(C) More than 4

8. `x = 10`

```
def func(a, b, c):
    return a + b * c
func(x, 1, 2)
```

The name `func` has _____ scope.

(A) local (B) global

What will be the output of the code given in questions from 9 to 23 ?

9. `c1 = len(globals())`

```
import numbers
c2 = len(globals())
print(c2 - c1)
```

10. `a = 10`

```
def func():
    print(a)
func()
```

11. `def func():`

```
    x = 10
    def f():
        print(x)
    f()
```

```
12. def func():
    def f():
        y = 30
        print(y)
    func()
13. a = 10
    def func():
        a = 20
        print(a, end=' ')
    func()
    print(a, end=' ')
14. a = 10
    def func():
        global a
        a = 20
        print(a, end=' ')
    func()
    print(a, end=' ')
15. print(min([4,3,1,5]))
    def min(x, y):
        return x if x < y else y
    print(min([4,3,1,5]))
16. a = 10
    def func():
        a += 1
        print(a, end=' ')
```

```

    func()
    print(a, end=' ')
17. def func():
        global n
        n = 5
    print(n)
    func()
18. def func():
        global n
        n = 5
    func()
    print(n)
19. def func():
        def f():
            nonlocal n
            n = 5
        f()
        print(n)
    func()
20. n = 5
    def func1():
        def func2():
            nonlocal n
            print(n, end=' ')
        func2()
    func1()

```

```
21. n = 5
    def func1():
        n = 10
        def func2():
            n = 15
            print(n, end=' ')
        print(n, end=' ')
        func2()
    func1()
    print(n, end=' ')
```

```
22. m = 5
    n = 5
    def func1():
        m = 10
        n = 10
        def func2():
            nonlocal m
            global n
            print(m, n, end=' ')
        print(m, n, end=' ')
        func2()
    func1()
```

```
23. x = 35
    def func(a, b):
        y = 30
```

```

def f1():
    pass
    def f2():
        pass
    print(len(locals()))
func(1, 2)

```

24. From the following two code snippets, which one will show error ?

<pre> def func(): for i in range(1, 2): range(1, 1): print(i, end=' ') print(i) func() </pre>	<pre> def func(): for i in range(1, 1): print(i, end=' ') print(i) func() </pre>

(A) Only 1 (C) Both 1 and 2

(B) Only 2 (D) Neither 1 nor 2

25. From the following two code snippets, which one will show error?

<pre> L = [1, 2] def func2(): L = [6, 7] func2() print(L) </pre>	<pre> L = [1, 2] def func1(): L.append(3) func1() print(L) </pre>

(A) Only 1 (C) Both 1 and 2

(B) Only 2 (D) Neither 1 nor 2

When you run a Python program, the data within your program is stored in objects referenced by variables. All this data is in primary memory, which is volatile. This is why any data generated by your program is gone when the program finishes executing, or the computer is turned off; it is not available when you execute your program next time. If you want your data to exist even after your program ends, you must store it permanently in non-volatile memory. You can do this by writing your data into a file stored on permanent storage like a hard disk or CD.

Till now, we have been reading and writing to the standard input and output. We were reading data from the keyboard, processing that data, and writing the information on the screen. In this chapter, we will see how to take input from files and how to send our output to files. We will learn how to write programs that can create files, write data into files, and read the data stored in files.

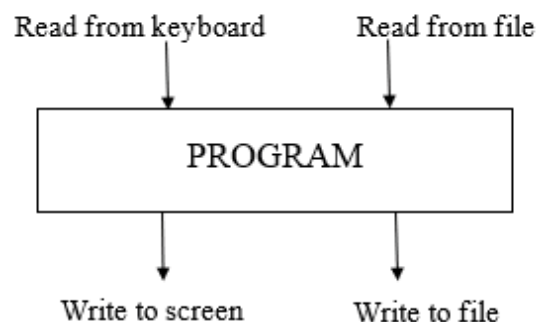


Figure 13.1: Input and output

Working with files mainly consists of three steps:

- Open a File
- Perform read/write operations on the file
- Close the File

Opening a file establishes a connection between the Python program and the external file, and closing the file breaks this connection. You can open a file by using the built-in function `open`. This function returns a file object that serves as a link between your Python program and the file. This object has different methods for reading and writing data, so the read-write operations can be performed using those methods. For closing the file, you can use the `close` method of the file object. Let us see a very simple example of reading and writing a file.

```
fout = open('data.txt', 'w')  
fout.write('This is my first file')  
fout.close()
```

First, we have called the `open` function. The first argument to this function is the name of the file that we want to open. The next argument is the mode in which we want to open the file. This mode is a string that describes the way in which the file will be used. Since we are going to write something in the file, we will open the file in write mode, which we can specify by using the letter `'w'`. If the file named `data.txt` does not exist, then it will be created and if a file with this name already exists then its contents will be erased, and we will get an empty file for writing. The function `open` returns a file object which we have assigned to the name `fout`. Now, the name `fout` refers to the file object returned by `open`, and by using this object we can write to our file `data.txt`.

Next, we have called the `write` method on the file object. The string that is sent as argument to this method is the text that we want to write in the file. After this we closed the file by calling the `close` method on the file object. When we run this program, a file named `data.txt` will be created and the given text will be written to the file. This file will be created in your current working directory, i.e., the same directory in which you are running your program. You can open the file using any text editor and view or edit its contents. So, we have seen how to write data to a text file from inside our

program; now, let us see how to read data from an existing file. The following program will read the file that we have just created:

```
fin = open('data.txt', 'r')
s = fin.read()
print(s)
fin.close()
```

Output-

```
This is my first file
```

We have called the `open` function with the file name to be opened and the second argument this time is `'r'`. This opens the file in read mode which provides read-only access to the file. This is the default mode for the `open` function so even if we do not provide any second argument, it means the same thing. The object returned by `open` is assigned to the name `fin`, and then we have called the `read` method on this object. This method returns the whole text of the file in the form of a string. We have assigned the return value to name `s` and then printed the string `s`. In the end, we have just closed the file. When we run this program, we can see the file's contents on our output screen.

The file object that was created has different attributes. For example, the `name` attribute returns the name of the file that is used in the call to `open` function. The `mode` attribute returns the mode in which the file was opened and `closed` attribute returns `True` if the file is closed.

```
>>> fin.name
'data.txt'
>>> fin.mode
'r'
>>> fin.closed
True
```

As usual, you can use the `dir` function to see everything related to this object.


```
>>> dir(fin)
['_CHUNK_SIZE', '__class__', '__del__',
 '__delattr__', '__dict__', '__dir__', '__doc__',
 '__enter__', '__eq__', '__exit__', '__format__',
 '__ge__', '__getattribute__', '__getstate__',
 '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__',
 '__lt__', '__ne__', '__new__', '__next__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_checkClosed',
 '_checkReadable', '_checkSeekable',
 '_checkWritable', '_finalizing', 'buffer', 'close',
 'closed', 'detach', 'encoding', 'errors', 'fileno',
 'flush', 'isatty', 'line_buffering', 'mode',
 'name', 'newlines', 'read', 'readable', 'readline',
 'readlines', 'reconfigure', 'seek', 'seekable',
 'tell', 'truncate', 'writable', 'write',
 'write_through', 'writelines']
```

The type of the file object depends on the mode which is used in the `open` function. This was a brief introduction to how files work in Python. In the coming sections, we will explore everything in more detail.

13.1 Opening a File

We know that if we need to access a file in our program, first we have to open it by using the built-in function `open`. The first argument to this function is a string containing the file's name. If you open a new file for writing, then the file is created in your current directory. If you open an existing file for reading or writing, Python looks for it in your current directory. If you want to create a file in a location other than your current directory, or if you want to read a file that is not in your current directory then you have to provide a path before the filename. A path is a hierarchy of directories that specifies a location on the file system. In the following call to `open` function, we have specified a file name with full path.

```
open('C:\dir1\dir2\data.txt', 'w')
```

We have seen earlier that if a backslash is followed by any escape character like `n` or `t`, then the combination will be replaced by the escape sequence. In the following example, we will get an invalid argument error because `\t` and `\n` are recognized as escape sequences and are replaced by their respective characters.

```
open('C:\textfiles\newfile.txt', 'w')
```

To avoid this, we could use double backslashes to separate the directories in the path or we could use a raw string. Double backslashes in a string are interpreted as a single backslash.

```
open('C:\\textfiles\\newfile.txt', 'w')
```

```
open(r'C:\textfiles\newfile.txt', 'w')
```

Windows operating system uses backslashes to separate the directories in the path, while macOS and Linux use forward slashes. Here is how we would specify a path on macOS.

```
open('/Users/xyz/dir1/dir2/data.txt', 'w')
```

Although the directory separator in a path is platform-specific, in Python, you can always use a forward slash, and it will be automatically converted to a backslash if required by the operating system. This means that forward slashes will work on Windows too.

```
open('C:/textfiles/newfile.txt', 'w')
```

The path we have specified here is an absolute path, which means that it is a complete path that starts from the root directory (such as `C:` or `E:` on Windows, and `/` on Linux or macOS) and ends at the directory where the file is stored. If the file is present in any subdirectory inside our current directory, then we could also specify a relative path which is a path that is relative to our current directory.

```
open('dir1/data.txt', 'w')
```

This will open a file `data.txt` that is present inside the subdirectory `dir1` of our current directory.

To make your program more flexible, you can let the user enter the filename instead of hardcoding the file name in your program. This way, we can use the same program to process different files.

```
filename= input('Enter the name of the file to be  
opened : ')
```

```
f = open(filename)
```

If the file does not exist, the `open` function will raise an error. To handle this error, we can enclose the code in the try except block, which is discussed later in the chapter on exception handling.

13.2 File opening modes

We have seen that the second argument to `open` function is the mode in which the file is opened. This argument specifies whether the file is opened for reading, writing, or appending. It also specifies whether the file is to be treated as a text file or a binary file. We have seen two modes, `'w'` and `'r'`. There are other modes also in which a file can be opened, so now let us see the details of all the modes:

`'r'` - read mode(default)

`'w'` - write mode

`'a'` - append mode

`'x'` - exclusive creation

We know that the mode `'r'` opens an existing file for reading only; the file should already exist. If you open a file in this mode, then you cannot write anything to it. It is the default mode, so if you do not provide any mode in the `open` function then this mode will be used.

The mode `'w'` opens a file for writing only, if the file does not exist then it creates a new file, if the file exists, then any content present in the file is erased. You cannot read from a file if you open it in this mode.

The mode `'a'` is the append mode. It will also open a file for writing only, but unlike the mode `'w'`, it will not erase the contents of the file if it already exists. If the file does not exist, then it creates a new file, and if the file

exists, then whatever you write to the file will be added at the end of the file. In this mode also, you cannot read from the file.

The mode 'x' is for exclusive creation. It is like the 'w' mode; it creates a new file but fails if the file already exists. So, it will create a new file only if the file with the given name does not exist. If the file exists, then it raises `FileExistsError`.

You can add a + sign to these modes if you want to perform both reading and writing on the same file. These are called update modes.

'r+' 'w+' 'a+' 'x+'

Text		Binary
r	rt	rb
w	wt	wb
a	at	ab
x	xt	xb
r+	r+t	r+b
w+	w+t	w+b
a+	a+t	a+b
x+	x+t	x+b

Figure 13.2: File opening modes

The mode 'r+' opens a file for both reading and writing and it works only on existing files. It will not create a file if it does not exist.

The mode 'w+' opens a file for both reading and writing. If the file already exists then the data in it is erased, otherwise a new file is created for reading and writing.

The mode 'a+' opens a file for both reading and writing, it will create a new file or append the contents at the end of the file.

The mode 'x+' also opens a file for both reading and writing, and it behaves like the exclusive creation mode.

In Python, files are broadly classified as text files and binary files. You can append letter `t` or `b` to the mode strings for working with text or binary files. For example, `'wt'` will open a text file for writing, and `'rb'` will open a binary file for reading. Text mode is the default, so you can skip the `t` if you want. Thus, adding a `t` or nothing means text and adding `b` means binary.

In [Section 13.4](#), we will see the differences between text and binary files.

13.3 Buffering

When you write data to a file through your program, that data is not directly transferred to the file. It is first placed in an area of primary memory which is called buffer.

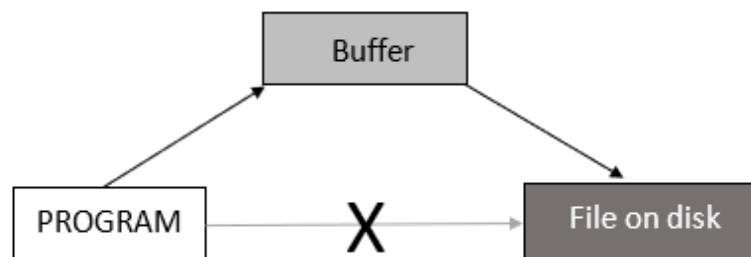


Figure 13.3: Buffering

The area is automatically associated with the file when it is opened. When the buffer becomes full, then only the data is written to the physical file. So, your data is written in chunks. This technique of buffering makes writing to files more efficient; it is done to increase the performance.

You can control buffering by providing a third argument to the `open` function. If the third argument is `0`, then buffering is disabled and data is transferred immediately to the file. This can reduce performance and it is allowed only in binary mode. If the buffering argument is `1`, line buffering is performed which means that the buffer is flushed every time you write a `'\n'` to the file, this is usable only in text mode. If this argument is any integer greater than one, then buffering is performed with that integer as the buffer size. If a negative value is given or this argument value is not provided in the call, then buffer size is the system default.

The `open` function takes some other arguments also which are all optional, but the first two arguments, file name and mode are the ones that you will mostly use.

13.4 Binary and Text Files

In [Section 13.2](#), we saw that we can open a file in either text mode or binary mode. When data is transferred in binary mode, no processing of data is performed by Python, you will get what is there in the file unprocessed. When data is transferred in text mode, some translations are performed by Python while reading and writing. Normally text files should be opened in text mode and binary files in binary mode. Let us see what are text and binary files.

A text file contains readable characters that are structured as lines of text, it also contains the non-printing newline character which indicates the end of each line. A text file is human readable and editable. These files can be read or written using any text editor. These files contain lines of text separated by newline characters and they do not contain any text formatting information like font, colors or size. For the computer, text file is just a sequence of characters, where newline is also a character. It is a special non printing character that makes the text appear on the next line. Inside our program, whenever we write to a text file, we have to write the content in the form of a string and whenever we read, we get the content of the file in string form in our program. Some examples of text files are .txt files, .py files and .csv files.

A binary file contains raw binary data that can be understood only by a computer program. These files can store non-textual data, examples are mp3 or image files, MS word files, pdfs, spreadsheets or executable files. These files are not human-readable or editable. If you try to open a pdf or an MS Word file using a plain text editor, you will see incomprehensible data on the screen. These files can be written and read by specific programs only.

We know that any file stored on storage media contains data in the form of bytes. When Python reads a file in text mode, it reads the bytes and converts them to text form which is human readable while when it reads a file in binary mode no such conversion is performed, the raw binary data is

provided to the program. Text files have a simple and fixed format, they contain lines of text separated by newlines while binary files have no such fixed format and that is why we need to process them differently based on their format. This requires proper understanding of the format of the particular binary file.

To interpret different formats of binary files, Python has different modules like `shelve`, `pickle` and `struct`; these modules can be used to read and write data to binary files. There are several third-party packages also that can be used to process pdf files, image files or other types of binary files in Python, examples are `PyPDF2` and `PIL`. You can just write data directly to the binary file in the form of a bytes string that contains hex codes. However, this low-level data transfer is not very practical.

When you open a file in text mode, there may be end of line and Unicode translations when data is written to a file, and when the file is read back, these translations are reversed. When a text file is written, the newline character(`'\n'`) is replaced by the platform-specific line ending. For example, in Windows, line ending is represented by the sequence `'\r\n'`. So, while writing to the file, `'\n'` is converted to this sequence and when reading the file, this sequence is converted back to `'\n'`. Similarly, while writing to a file, the Unicode characters are translated into raw bytes, which is called encoding, and they are decoded when the file is read. You can specify an encoding argument in the `open` function. Python recognizes many encodings such as ASCII, Latin-1, utf-8, utf-16, utf-32, and many more.

```
>>> open('data.txt', 'w', encoding='utf-8')
```

If no encoding is specified, the default platform dependent encoding is used. The function `getencoding` from the `locale` module will give you the encoding for your platform. For this Windows system, the following encoding is returned:

```
>>> import locale
>>> locale.getencoding()
'cp1252'
```

With this encoding, if we try to read a text file that contains characters from different languages encoded in UTF-8, we will get a

UnicodeDecodeError.

```
>>> f = open('myfile.txt', 'r')
```

```
>>> s = f.read()
```

```
..... UnicodeDecodeError: 'charmap' codec can't  
decode byte 0x8f in position 59: .....
```

We did not provide any encoding argument so Python used the default encoding which is 'cp1252' in this case. This encoding was unable to decode the characters that are there in the file, and so the UnicodeDecodeError was raised. In the file, the text was copied from a source (web page) that was UTF-8 encoded and so if we use 'utf-8' encoding while opening the file, the text can be read successfully.

```
>>> f = open('myfile.txt', 'r', encoding = 'utf-8')
```

```
>>> s = f.read()
```

```
>>> print(s)
```

मेहनत इतनी खामोशी से करो कि सफलता शोर मचा दे

치지 않은 공은 100% 골인되지 않는다.

Тише едешь—дальше будешь.

Since the encoding for different systems can be different, the code that does not provide an encoding argument might work on some platforms and fail on others. For example, the encoding used on most Mac and Linux systems is UTF-8 so the text file encoded in UTF-8 will be successfully read on those systems even if we do not provide the encoding argument. However, on Windows systems the encoding is CP-1252 and so we will get an error while reading a text file encoded in UTF-8. Therefore, it is important to provide an encoding argument so that the interpreter can correctly decode the file for you. Python supports many encodings; you can use any encoding that serves your purpose but UTF-8 is the standard these days and is the recommended encoding.

All these translations are automatically done by Python if the file is opened in text mode, user just needs to open the file and can start reading or writing. These end of line translations and Unicode encoding/decoding are turned off in binary mode. Since these translations are not performed in binary mode,

binary input/output is faster than the text input/output. When you open files that contain binary data like image files or executables, be careful not to use the text mode as these translations will corrupt the data. Use binary mode for such files as it will access the raw binary data without any alteration.

While opening a file in binary mode, you cannot provide an encoding as it is not required. In binary mode, we read and write bytes and not strings, so there is no conversion to be done in binary mode and there is no need of any encoding argument. In text mode, the bytes stored in the file have to be converted to strings in our program while reading, and strings in our program have to be converted to bytes while writing, so Python needs to know how to perform these conversions, and that is why encoding argument is needed in text files.

You can open your text file in binary mode to see what bytes are actually stored in it.

```
>>> f = open('myfile.txt', 'rb')
>>> s = f.read()
>>> print(s)
```

When you work in text mode, Python expects and produces objects of type `str`. This means that while writing to the file you can write objects of type `str` only and while reading, the content of file is automatically translated and returned as `str`. When you work in binary mode, Python expects objects of type `bytes` or `bytearray` and produces objects of type `bytes`. So, when you read from a binary file, content is returned raw and unchanged in the form of `bytes` objects.

We have seen the type `str` in detail, it is an immutable sequence of Unicode characters, it is used for handling textual data in Python. For handling binary data, we have the string type `bytes` and `bytearray`. A `bytes` object is an immutable sequence of single 8-bit bytes. It supports most of the `str` operations and displays as ASCII whenever possible. A `bytes` literal is written by preceding a string literal with the letter `'b'`. The type `bytes` is immutable, the type `bytearray` is the mutable version of `bytes` type. So `str` type represents the text string in Python and `bytes` and `bytearray` types represent binary strings in Python. When we work in text mode, we

give and get `str` object, and in binary mode, we give and get `bytes` objects.

13.5 Closing a file

We have seen that when we are finished working with a file, we should close it by calling the `close` method on the file object. After a file is closed, any attempts to use the file object will automatically fail. Closing the file is important as it ensures that the data is properly written to the file and all the system resources attached with the file object are released.

We know that Python's built-in garbage collector will reclaim an object's memory space if it is no longer referenced. This applies to file objects also, but it is a good practice to explicitly close the file after you are done working with it. Closing a file also means that the file has been released by our program so that it can be used in another program. Closing the file becomes important when you are writing some data to a file. If there is any buffered output in the memory, then the call to `close` method automatically flushes it to the disk. Let us see this with the help of an example:

```
f = open('time.txt', 'w')
f.write('Time is precious.')
```

In this short program, we have opened the file `time.txt` in write mode and we have written some text to the file. When we run this program, it will execute successfully. After this we run another program where we are reading this file.

```
f = open('time.txt', 'r')
s = f.read()
print(s)
```

In this program, we have opened the file in read mode and we are printing the text that is read from the file. When we run this, nothing is printed which means that the string `s` is empty, there was nothing in the file which could be read. When we open the file `time.txt` in text editor, we can see that it is empty.

This happened because when we write something to the file, the data is first sent to the buffer and when the buffer is full, then only the data is written to the file. The data that we are writing is very less data, it did not fill the buffer so the data was not transferred to the file and so our file was empty. Now this time, let us increase the amount of data that we are writing to the file.

```
f = open('time.txt', 'w')
f.write('Time is precious.' * 1000)
f.write('Waste it wisely')
```

After executing this program, when we read the data again, we can see the output which means that now the data has been written to the file, but we cannot see the last line in the output ('Waste it wisely'). This line was not written to the file because it did not completely fill the buffer. To ensure that all the data is written to the file, you need to flush the buffer and the flushing is automatically done when the file is closed. So, now this time let us insert the call to close method:

```
f = open('time.txt', 'w')
f.write('Time is precious.' * 1000)
f.write('Waste it wisely' )
f.close()
```

After executing this program, when we read the file, we can see the last line also. This whole observation was on IDLE. On a different IDE or system, you might get your data written to the file, even when you write a single line and do not close the file. But this writing is not guaranteed, and you cannot rely on it, so it is always good to close the file.

Thus, closing a file not only releases the resources attached to it. It also ensures that any contents that you have written to the file are saved in it. Any data that is there in the memory buffer is transferred to the physical file on the disk. If you forget to close the file, you might lose some data. If you want to flush the output buffer without closing the file, you can use the flush method.

```
f.flush()
```

13.6 with statement

We have seen that when we have to perform any operation on a file, we need to open the file then perform that operation and then close the file. If we forget to close the file or some exception occurs while working on the file then the file will not be closed which might result in loss of data and resource leakage. Since closing of file is important, we can write our file operations inside a `with` statement which ensures that the file is always properly closed. Here is an example of a file reading operation code and equivalent code using the `with` statement.

```
f = open('data.txt', 'r')          with
open('data.txt', 'r') as f:
s = f.read()                      s = f.read()
print(s)                          print(s)
f.close()
```

The `with` statement consists of a heading and an indented block of statements. In the heading, we have the `with` keyword followed by the call to `open` function. The file object returned by `open` will be assigned to the name that follows the `as` keyword. Inside the `with` block, you can place all your file operation statements that work on the opened file. If you use the `with` construct, there is no need to explicitly call the `close` method because when the block ends, the `close` method is automatically called. The file is closed properly even if there is an exception raised inside the block. This is why it is a good practice to place your file processing statements inside a `with` block.

If you want to work simultaneously with two files, you can write a `with` statement inside another `with` statement.

```
f1 = open('data.txt', 'r')          with
open('data.txt', 'r') as f1:
f2 = open('new.txt', 'w')           with
open('new.txt', 'w') as f2:
s = f1.read()                      s = f1.read()
```

```
f2.write(s)                f2.write(s)
f1.close()
f2.close()
```

In the code on the left side, we have opened two files, we are reading from one file and writing to another and then we have closed both the files. In the code that is written on the right side using nested `with` statements, there is no need to write the two calls to `close` method.

This code can also be written using a single `with`, by placing a comma in between.

```
with open('data.txt', 'r') as f1, open('new.txt',
    'w') as f2:
    s = f1.read()
    f2.write(s)
```

In the following example we have checked the `closed` attribute of the file object after the `with` statement, and it shows that the file gets closed after the `with` block finishes.

```
with open('time.txt', 'w') as fout:
    fout.write('Time is precious')
print(fout.closed)
with open('time.txt', 'r') as fin:
    s = fin.read()
    print(s)
print(fin.closed)
```

Output-

True

Time is precious

True

We have opened the file `time.txt` in write mode and then opened the same file in read mode and printed the data read from it. We have not called the `close` method anywhere. The output shows that the data was properly written and read from the file. The `closed` attribute of these two objects `fout` and `fin` is `True` which shows that the files were automatically closed because of the `with` statement.

13.7 Random Access

On your computer, when you open a file for reading or writing something, for example a word file or a file in notepad, you can see a cursor that can be moved around in the file. It is the place where all the action takes place in the file. If you want to write something at the end of the file, you take the cursor to the end of the file. If you want to read something from the 3rd page of the file you take your cursor there. You control this cursor by using keys or your mouse. So, you can jump around in the file and read and write at specific locations. Basically, you have random access to the file.

When we are working with files in Python, we are reading and writing data through our program. We do not see the file directly, but we might want to move around the file to write or read at specific locations. We can assume that there is an invisible cursor moving around in the file when we are working with it. The position of this invisible cursor is maintained by the file object. The file object maintains the current position where the read and write operations are performed on the file, so it keeps track of our current position in the file.

Generally, when a file is opened in any mode other than the append mode, the cursor is at offset 0, which means that it is at the beginning of the file. As you perform read and write operations, this cursor proceeds forwards. When you read or write `n` bytes of data, the cursor moves `n` bytes forward.

To know the current position of the cursor, you can call the method `tell` and to change the position of the cursor, you can call the method `seek`.

The call `f.tell()` returns the current position of the cursor in the file, where position is an integer offset in bytes from the beginning of the file.

The method `seek` lets us move the cursor to a different location for the next read/write operation. The call `f.seek(n)` changes the file position for next operation to integer offset `n`. This offset is measured in bytes, from the beginning of the file.

The method `seek` can take a second optional argument also that specifies the reference point relative to which the cursor is moved.

```
f.seek(n, from_where=0)
```

If this second argument is 0, reference point is the beginning of the file, this is the default value. So, if you provide 0 or do not provide any value, cursor is moved `n` bytes away from the beginning of the file. If this argument is 1, reference point is current location so cursor is moved `n` bytes away from the current location. If this argument is 2, reference point is the end of the file which means that the cursor is moved `n` bytes away from the end of the file.

In the `os` module, there are three names `SEEK_SET`, `SEEK_CUR`, `SEEK_END` with values of 0, 1 and 2. If you want more readability in your `seek` calls, you can use these names instead of integers for the second argument.

If the offset `n` is positive, the cursor is moved forward and if it is negative, it is moved backward in the file. Let us see some examples of `seek` calls.

<code>f.seek(0)</code>	Moves the cursor back to the beginning of the file
<code>f.seek(0, 2)</code>	Moves the cursor to the end of the file
<code>f.seek(50)</code>	Moves the cursor 50 bytes forward from the beginning of the file
<code>f.seek(20, 1)</code>	Moves the cursor 20 bytes forward from the current location
<code>f.seek(-20, 1)</code>	Moves the cursor 20 bytes backward from the current location
<code>f.seek(-10, 2)</code>	Moves the cursor 10 bytes backward from the end of the file

Table 13.1: Calls to seek method

When you have read the whole file, cursor goes to the end of the file, if you want to read the file again you can rewind the file and go to beginning by using `f.seek(0)`. If you are at the beginning of the file and want to append some data, you can reach the end by using `fseek(0, 2)`.

You can use a value that was previously returned by `tell`, as an offset from the beginning. This way you can go back to a previous location. In the following code, the value returned by `f.tell()` is used as an argument in the `seek` method:

```
pos = f.tell()

.....

.....

f.seek(pos)
```

13.8 Using seek in text mode

If a file is opened in text mode, then the second argument to `seek` cannot be 1 or 2. This means that you cannot use the current location or end of file as the reference points in text mode. You can seek only relative to the beginning of the file. So, while working in text mode, if you use 1 or 2 as the second argument there will be an error. There is an exception to this, you can seek to the end of the file, so the following call to `seek` is valid even though it has 2 as the second argument:

```
f.seek(0, 2)  # seeking to the end of the file
```

The first argument to `seek` is an offset value, and in text mode the allowed values for offset are only those values that are returned by `tell` or you can use zero as an offset. Only these values are legal, any other offset value can cause undefined behaviour. This happens because when a file is opened in text mode, line end translations and Unicode encodings are performed. This is why the `seek` method may not set the position correctly, if you provide an offset that is not a result of any previous call to `tell`.

So, if you are working in text mode you can easily use `seek(0)` to go to beginning of the file and `seek(0, 2)` to go to the end of the file or you can go to a previous location in the file by using a value returned by `tell`. If you use any other value as the offset then you may get unexpected results.

If you have a file that has no line ends and contains only ASCII characters (one byte per character), any value for offset works in text mode. In these

files, n^{th} byte corresponds to the n^{th} character. If your file contains line ends or encoded Unicode characters that use varying number of bytes, then different offset values might not work properly. In these files, n^{th} byte does not correspond to the n^{th} character. Let us see with the help of an example, we have two files one has text written in Chinese and the other has text written in English:

----- myfile1.txt-----

All is well.

----- myfile2.txt-----

猿も木から落ちる

-----myprog.py-----

```
with open('myfile1.txt','r', encoding='utf-8') as
f:
```

```
    f.seek(4)
```

```
    print(f.read())
```

```
with open('myfile2.txt','r', encoding='utf-8') as
f:
```

```
    f.seek(4)
```

```
    print(f.read())
```

Output-

is well.

UnicodeDecodeError: 'utf-8' codec can't decode byte 0x82 in position 0: invalid start byte

Since English symbols take 1 byte per character, the call to seek works and the data is read successfully. The first 4 characters correspond to the first four bytes. We get a `UnicodeDecodeError` when we used seek on the second file as it contains Chinese symbols that use multiple bytes; each symbol is taking 3 bytes. We can confirm this on interactive prompt by using the `encode` method of `str` type, this method encodes the string according to the provided encoding standard.

```
>>> len('猿も木から落ちる')
8
>>> len(str.encode('猿も木から落ちる', 'utf-8'))
24
>>> len('All is well.')
12
>>> len(str.encode('All is well.', 'utf-8'))
12
```

The sequence of calls `f.seek(4)` and `f.read()` in `myfile2.txt` failed because we were in the middle of a character while reading. The calls `f.seek(3)` or `fseek(6)` would work as they would take us to the starting of a character.

In binary mode, the method `seek` will always work correctly, for any value of offset and the second argument can take any of the three values 0, 1 or 2.

13.9 Calling seek in append mode

Generally, when you open any file, the cursor is at the beginning of the file except in `a` and `a+` modes. In these modes, the cursor is at the end when the file is opened. In `'a'` and `'a+'` modes, data will always be written at the end. So, calling `seek` has no effect in mode `'a'`. In mode `'a+'`, calling `seek` has no effect, if the next operation writes data but it works if the next operation reads data.

So, in append modes (`'a'` and `'a+'`), data will always be written at the end, irrespective of any call to `seek`.

13.10 Reading and writing to the same file

Random access to files becomes more important when we are reading and writing to the same file. We have seen that adding a `'+'` sign to the file mode allows both reading and writing. For example, the modes `'w+'`,

'r+', 'a+', 'w+b', 'r+b' and 'a+b' can be used for reading and writing to a file at the same time.

Let us understand the difference between these three types of modes ('w+', 'r+', 'a+'). In 'w+' and 'w+b' modes, a new file will be created, if the file does not exist. If the file exists, all its data will be deleted. So, in these modes, you will always get a blank file for reading and writing. Initially, there will be nothing to read. First, you have to write something, and then you can move the cursor to read.

If you want to open an existing file for reading and writing, you should use the 'r+' or 'r+b' mode as these modes will not delete the existing contents of the file. In 'a+' and 'a+b' modes also, the existing contents will not be deleted. In 'a+' and 'a+b' mode, you can read the file at any place, but you can only write at the end of the file.

When you open a file in 'r+' or 'r+b' mode, the cursor is initially placed at the beginning of the file. If you open the file and just start writing, then you will overwrite the existing contents of the file. To avoid this, you should move the cursor to a place where you want to write the data. The cursor can be moved by using `seek` or by reading the file. If you read the whole file, the cursor will be moved to the end.

When you open a file in 'a+' or 'a+b' mode, the cursor is initially placed at the end of the file. So, if you open a file in this mode and try to read from it straightaway then you will not get anything, because the cursor is at the end. You need to move it to be able to read something.

Let us see some examples:

```
>>> f = open('testfile.bin', 'w+b')
>>> f.write(b'abcdefghijklmn')
14
>>> f.write(b'123456789')
9
```

We have created a new file in 'w+b' mode and have written some binary data in it. The string is preceded with letter b, so it is of `bytes` type. Let us see where the cursor is by using the `tell` method.

```
>>> f.tell()
```

```
23
```

Now let us try to read the file.

```
>>> f.read()
```

```
b''
```

We get an empty `bytes` string, because the cursor is at the end of the file. Let us move the cursor to the beginning by using the `seek` method.

```
>>> f.seek(0)
```

```
0
```

Now the cursor is at the beginning, let us read the file now.

```
>>> f.read()
```

```
b'abcdefghijklmn123456789'
```

Now the whole file was read, and the cursor again moved to the end.

```
>>> f.tell()
```

```
23
```

Let us take the cursor 9 bytes backwards from the end. We need to go backwards so we will give a negative offset, and we need to move from the end of the file so the second argument will be 2.

```
>>> f.seek(-9, 2)
```

```
14
```

We can provide an argument to the `read` method to read specified number of bytes, so now let us read 3 bytes from the file.

```
>>> f.read(3)
```

```
b'123'
```

This read moves the cursor 3 bytes forward because we read 3 bytes from the file.

```
>>> f.tell()
```

17

To go to the end of the file you can write:

```
>>> f.seek(0, 2)
```

23

Let us close this file and now we will work in text mode.

```
>>> f.close()
```

Now we will open another file `data1.txt` in `'a+'` mode, there is no `b` in the mode argument so this is text mode.

```
>>> f = open('data1.txt', 'a+')
```

```
>>> f.tell()
```

21

We can see that the cursor is not at the beginning of the file, it is at the end of the file. To read the file, let us take the cursor to the start of the file.

```
>>> f.seek(0)
```

0

```
>>> f.read()
```

```
'This is first line.\n'
```

Now, let us append some data to the file.

```
>>> f.write('This is next line\n')
```

18

Let us go back to the start of the file and read it.

```
>>> f.seek(0)
```

0

```
>>> f.read()
```

```
'This is first line.\nThis is next line\n'
```

Cursor has come to the end because of this read operation, so again we go to the start of the file and write some data.

```
>>> f.seek(0)
```

```
0
```

```
>>> f.write('This is last line\n')
```

```
18
```

We go to start and read the file.

```
>>> f.seek(0)
```

```
0
```

```
>>> f.read()
```

```
'This is first line.\nThis is next line\nThis is
last line\n'
```

The string 'This is last line' was written at the end of the file even when the cursor was at the start of the file. So, in append mode, the data will always be written at the end only no matter where your cursor is. Now let us close this file and open it again in 'r+' mode.

```
>>> f.close()
```

```
>>> f = open('data1.txt', 'r+')
```

```
>>> f.tell()
```

```
0
```

The cursor is at the beginning of the file. Now we want to write some data to the end of the file. If we just start writing, then the data at the start of the file will be overwritten. So first we will take the cursor to the end of the file and then write.

```
>>> f.seek(0, 2)
```

```
59
```

```
>>> f.write('This is a new line\n')
```

```
19
```

```
>>> f.seek(0)
```

```
0
```

```
>>> f.read()

'This is first line.\nThis is next line\nThis is
last line\nThis is a new line\n'

>>> f.close()
```

13.11 Reading a File using read()

To be able to read from a file you have to open the file in any one of these modes.

```
'r'           'r+'           'w+'           'x+'           'a+'           'rb'
'r+b'         'w+b'         'x+b'         'a+b'
```

We have previously encountered the `read` method that is used to read data from a file. It reads the contents of the file from the current location till the end of file, into a string and returns that string. If you are working in text mode, you will get a string of type `str` and in binary mode you will get a string of `bytes` type.

The `read` method can take an optional argument, which represents the number of characters to read in text mode, and number of bytes to read in binary mode. The call `read(n)` starts reading at the current location and reads up to next `n` characters (or `n` bytes) into a string and returns that string. If there are not enough characters left in the file, then it reads all the remaining characters. If this argument is negative or omitted, the rest of the file is read.

If end of file has been reached and then `f.read()` is called, and it will return an empty string.

We have the following file `mydata.txt` and we will read this file in our program using different methods.

```
----- mydata.txt-----
```

Beautiful is better than ugly.

명시적인 것이 암시적인 것보다 낫습니다.

단순한 것이 복잡한 것보다 낫습니다.

Flat is better than nested.

In the following code we have used the `read` method without any argument to read the entire file, we have done this earlier as well:

```
with open('mydata.txt', 'r', encoding='utf-8') as f:

    print(f.read())
```

Output-

Beautiful is better than ugly.

명시적인 것이 암시적인 것보다 낫습니다.

단순한 것이 복잡한 것보다 낫습니다.

Flat is better than nested.

If we want to read only the first 40 characters, we can provide an argument to `read`.

```
with open('mydata.txt', 'r', encoding='utf-8') as f:

    print(f.read(40))
```

Output-

Beautiful is better than ugly.

명시적인 것이 암

Now only the first 40 characters are read.

You can read the whole file in a single string, but if the file size is large, then it will consume a lot of space in memory. For example, if you have a 2GB file, then it will take 2 GB space in memory. If you do not want to consume so much space in memory at a time, then you can read the whole file in chunks. Let us see this for our small file `data.txt`, we will read it in chunks of 10 characters.

```
with open('mydata.txt', 'r', encoding='utf-8') as f:

    while True:
```



```
part = f.read(10)
if part == '':
    break
print(part, end='')
```

Output-

Beautiful is better than ugly.

명시적인 것이 암시적인 것보다 낫습니다.

단순한 것이 복잡한 것보다 낫습니다.

Flat is better than nested.

The `read` method returns an empty string at the end of the file, so we have put a `break` statement to terminate this loop when end of file occurs.

In binary mode, the argument to `read` denotes bytes, so you can read the file in chunks of bytes. For example, you could read your 2 GB file in chunks of 10 megabytes. In binary mode, a `bytes` string is returned so the empty string would be denoted as `b''`.

13.12 Line oriented reading

The next few methods that we will see for reading a file are line-oriented methods. These methods will work in binary mode as well, but they are meaningless in that case because binary data is not line oriented. Line based file processing is meaningful only in text files as they contain text organized in lines.

To read the file a line at a time, we can use the `readline` method which reads the next line into a string and returns that string. It reads the contents of the file until it finds a newline character, it then returns the content read so far and the newline character in a string. At the end of the file, it returns an empty string. We will take an example file and read it line by line using the `readline` method.

-----zenpython.txt-----

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

```
with open('zenpython.txt', 'r') as f:
    print(f.readline())
```

Output-

Beautiful is better than ugly.

The first line is read, now this time we will read the second and third lines also.

```
with open('zenpython.txt', 'r') as f:
    print(f.readline())
    print(f.readline())
    print(f.readline())
```

Output-

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

We can see some extra empty lines are printed in the output which were not present in the file. Let us see why we have these extra lines. Each line in the file ends with a newline character and this newline character is read and included in the string that represents the line. So, the strings that are returned by the `readline` method have the newline character included at the end, and the `print` function adds its own newline character while printing. This is why two newline characters are printed, one that is read from the file and the other one from `print`, so we get extra empty lines in the output. If we do not want these lines then we have to tell the `print` function to suppress the newline character while printing.

```
with open('zenpython.txt', 'r') as f:
```

```
print(f.readline(), end='')
print(f.readline(), end='')
print(f.readline(), end='')
```

Output-

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Now the extra lines are not printed. Another way to avoid the extra lines could be to call the `rstrip` method to remove any whitespace from the end of the returned string.

```
with open('zenpython.txt', 'r') as f:
    print(f.readline().rstrip())
    print(f.readline().rstrip())
    print(f.readline().rstrip())
```

We can put the `readline` method inside a loop to read the entire file line by line.

```
with open('zenpython.txt', 'r') as f:
    while True:
        s = f.readline()
        if s == '':
            break
        print(s, end='')
```

`readline` returns an empty string when the end of file is reached so we have put a `break` to terminate the loop when the end of file is reached.

Blank lines inside the file are represented as strings containing a single newline character. They are not returned as empty strings. Empty string is returned only when the end of file is reached.

If you send an integer argument to the `readline` method, then it will read that much characters from the next line. The call `f.readline(n)` will read `n` characters from the next line into a string and return that string.

If you want to read all the lines of a file in a list of strings, you can use the `readlines()` method. This method returns all the remaining lines of the file as a list of strings. The newline character is retained in the strings.

```
with open('zenpython.txt', 'r') as f:
    lines = f.readlines()
    print(lines)
```

Output-

```
['Beautiful is better than ugly.\n', 'Explicit is
better than implicit.\n', 'Simple is better than
complex.\n', 'Complex is better than
complicated.\n']
```

This method loads the entire file into the memory at once so it can prove to be costly for big files. You could also get a list of all the lines in file by sending the file object to the `list` function, so `list(f)` is another way to get lines of the file in a list.

The best way to read a text file line by line is by treating the file object as an iterator and using it in a for loop. A file object that is opened in text mode for reading, is an iterator whose items are the lines of the file.

```
with open('zenpython.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

This for loop iterates over the file object, and in each successive iteration we get a string that contains the next line from the file including the newline character. The string that we get in each iteration is assigned to the loop iteration variable. The loop is automatically terminated when there is no more data left in the file to read. When you use this for loop, there is no need of using any read method. This is more efficient in terms of space than using `for line in f.readlines()` since it does not fetch the whole file in

memory at a time. So, this is an efficient and simple way to read the file line by line.

To avoid the blank lines in the output, we can either suppress the newline in the `print` function as we have done in our code, or we can call the `rstrip` method to remove any whitespace from the end of the returned string.

If we are not at the beginning of the file, but in some other location, then `readline` method will return remainder of the current line of the file, `readlines` method will return all remaining lines of the files as a list of strings, and `for line in f:` will iterate remaining lines of the file.

13.13 Writing to a file

To write data to a file you have to open the file in one of these modes.

'w' 'a' 'x' 'w+' 'a+' 'x+' 'r+' 'wb' 'ab'
'xb' 'w+b' 'a+b' 'x+b' 'r+b'

In the case of 'w', 'w+', 'wb', 'w+b' modes, you need to be careful because if the file already exists then the data that is present in the file will be erased and you will get a blank file for writing. In case of 'r' and 'r+b' modes, you need to move the cursor to avoid overwriting the existing contents. If you want to append data to an existing file then it should be opened in append mode.

We have already seen the `write` method that is used to write data to the file. It writes a string of characters (or bytes in binary mode) into a file and returns number of characters (or bytes) written. In text mode, you need to provide it a string of type `str`, and in binary mode you need to give it a `bytes` string. Other types of objects have to be converted to a string (in text mode) or a bytes string (in binary mode) before writing them using the `write` method. Let us see some examples:

```
with open('learn.txt', 'w') as f:
    f.write('Data Science\n')
    f.write('Machine Learning\n')
```

```
f.write('Artificial intelligence\n')
```

We have opened the file `learn.txt` in write mode and called the `write` method three times. After running this program, the three strings will be written to the file on separate lines. Unlike `print` function, the `write` method does not add a trailing newline character at the end of the string that is written to the file. The newline character will be added only if it is a part of the string being written. So, you have to add the newline explicitly in the string that you are writing to the file otherwise the next `write` call will write the data at the same line.

We know that when we read data in binary mode, it is returned in the form of a bytes string. Similarly when we have to write data in binary mode, we supply the data in the form of bytes string or bytearray object. You can use the `encode` and `decode` methods while reading and writing to binary files.

```
with open('myfile.bin', 'wb') as f:
    data = '👉 Explicit is better than implicit
👉 '
    f.write(data.encode('utf-8'))
with open('myfile.bin', 'rb') as f:
    s = f.read()
    data = s.decode('utf-8')
    print(data)
```

If the strings that we want to write are present in an iterable, like a list or a tuple, then we can use the `writelines` method. This method writes all the strings present in an iterable into a file and it does not return any value. This method also works in both binary and text modes. Let us open our `learn.txt` file in append mode and use the `writelines` method to write strings from a list.

```
L = ['Python\n', 'Java\n', 'Swift\n', 'Perl\n']
with open('learn.txt', 'a') as f:
    f.writelines(L)
```

This method just writes the strings as such, if we want the strings to be on separate lines, newlines have to be present at the end of each string. You could also write a list of strings by calling the `write` method repeatedly inside a for loop, or by joining the strings using the `join` method and then calling the `write` method once for that joined string. But using this `writelines` method is faster than both of them.

These methods write only string type objects to the file, if you want to write any other type of Python object then it has to be converted to `str` in text mode or `bytes` in binary mode or you will have to use pickling which we will see later in this chapter.

13.14 Redirecting output of print to a file

The `print` function can also be used to write to a file. The output of `print`, that is by default, sent to the screen can be redirected to an open file. For this, you have to supply the file object as an argument for the named parameter `file`. Here is an example:

```
x = 3567

with open('data2.txt', 'w') as f:
    print('Ultimate Python', x, file=f)
```

The output produced by `print` will be written to `data2.txt` file. The value of variable `x` will be stored as sequence of 4 characters not as an integer, since we are working in text mode. When we write to a file using the `print` function, the newline will automatically be written since that is the default behaviour of `print`. If we want to suppress it, we can supply an argument for the named parameter `end`. This redirection of output to a file using the `print` function will work only in text mode.

13.15 Example Programs

In this section, we will write some programs for file processing in text mode.

1. Write a program to display a file in reverse order, line by line.

```
with open('names.txt', 'r') as f:
    for line in reversed(f.readlines()):
        print(line, end='')
```

The `readlines` method gives us a list of all the lines in the file, and we sent this list to the `reversed` function which returns an iterator that accesses the list in reverse order. This iterator is used in a for loop and so the lines of the file are printed in reverse order.

2. Write a program to count the number of lines in a file.

```
with open('names.txt', 'r') as f:
    print(len(f.readlines()))
```

The list returned by `readlines` method contains all the lines of the file and finding its length gives us the number of lines in the file. If the file is too large and you do not want to use the `readlines` method, you can use the file iterator.

```
with open('names.txt', 'r') as f:
    count = 0
    for line in f:
        count += 1
print(count)
```

3. Write a program to display the contents of the file with line number displayed before each line.

```
with open('names.txt', 'r') as f:
    count = 0
    for line in f:
        count += 1
        print(count, line, end='')
```

In the loop that we had written for counting number of lines, we will first print the value of `count` variable, followed by printing the line itself. This

will display all the lines of the file with line number.

4. Write a program to search for a string in a text file. Display all the lines in the file that contain the search string.

```
search_string = input('Enter the text to be
searched : ')

with open('learn.txt', 'r') as f:
    for line in f:
        p = line.find(search_string)
        if p >= 0:
            print(line, end='')
```

We have used the `find` method of `str` type; it returns -1 if the search string is not found. A line of the file is printed only if the return value of this method is greater than or equal to 0 for that line. This way only those lines will be printed that contain the search string.

5. Write a program to copy the contents of a file to another file.

```
with open('new.txt', 'w') as f1, open('names.txt',
'r') as f2:

    f1.write(f2.read())
```

The file `names.txt` is opened in read mode and the file `new.txt` is opened in write mode. First, we read the whole file `names.txt` in a string and then we write this string to the file `new.txt`. This copies the whole content of the file `names.txt` to `new.txt`. You can read and write line by line if you do not want to read the whole file in one string.

```
with open('new.txt', 'w') as f1, open('names.txt',
'r') as f2:

    for line in f2:

        f1.write(line)
```

If we open the file `new.txt` in append mode, then the contents of the file `names.txt` will be added at the end of the file `new.txt` without erasing its existing content.

```
with open('new.txt', 'a') as f1, open('names.txt',
'r') as f2:
    for line in f2:
        f1.write(line)
```

6. Write a program to append the contents of variable number of files to a file.

```
def append(file1, *args):
    with open(file1, 'a') as f1:
        for file in args:
            with open(file, 'r') as f2:
                for line in f2:
                    f1.write(line)

append('school.txt', 'class1.txt', 'class2.txt',
'class3.txt')

append('people.txt', 'students.txt',
'employees.txt')
```

Since the number of files that are to be appended is variable, we have made a function that takes variable number of arguments. The first parameter to this function is the name of the file to which we want to append, and after that it can accept variable number of arguments. These arguments will be the names of the files which we want to append to this first file.

The first file `file1` is opened in append mode. After this, we have written a for loop to iterate over the `args` tuple that contains variable number of arguments. Inside this loop we have opened each file in read mode and written its data to `file1`.

We have called this function two times, in the first call the data of the files `class1.txt`, `class2.txt` and `class3.txt` will be appended to `school.txt` and in the second call the data of files `students.txt` and `employees.txt` will be appended to the file `people.txt`.

7. Write a program to append the contents of a file to variable number of files.

```
def append(file1, *args):
    with open(file1, 'r') as f1:
        for file in args:
            with open(file, 'a') as f2:
                for line in f1:
                    f2.write(line)

    f1.seek(0)

append('copyright.txt', 'document1.txt',
'document2.txt')

append('companyinfo.txt', 'doc1.txt', 'doc2.txt',
'doc3.txt', 'doc4.txt')
```

In this function, `file1` is the file whose content is to be appended and rest of the files after that are the files which get the content added. So, this time we have opened `file1` in read mode and other files in append mode. Note that we have used the `seek` method to go to the beginning of `file1`, every time after reading its content. Here is another way of writing this function:

```
def append(file1, *args):
    with open(file1, 'r') as f1:
        text = f1.read()

    for file in args:
        with open(file, 'a') as f2:
            f2.write(text)
```

In this function we are not reading `file1` line by line every time we need to append its contents. We just read its once, store its contents in variable named `text` and then append this text to all the files.

8. Write a list comprehension to get a list of all those lines in the file that start with a digit. The strings in the list should not contain the ending

newline characters read from the file.

```
with open('info.txt', 'r') as f:
    lines = [line.rstrip() for line in f if
line[0].isdigit()]
    print(lines)
```

We iterate over each line and by using the `if` clause we select only those lines that start with a digit and to remove the ending newline character, we have used the method `rstrip`.

9. Write a program to count the number of blank lines in a file. Any line that contains only newline character, tabs or spaces should be considered a blank line.

```
blank_lines = 0
with open('info.txt', 'r') as f:
    for line in f:
        if line.strip() == '':
            blank_lines += 1
print(blank_lines)
```

After removing the whitespace characters from a line by using the `strip` method if the line is empty, it means that the line had only whitespace characters and so is counted as a blank line.

10. The following file named `employees_info.txt` contains information of employees on separate lines. The fields of information are employee ID, name, email ID, phone number and salary and these fields are separated by colon.

-----employees_info.txt-----

```
A231 : Raman : raman@xyz.com: 9988008898 : 25000
F632 : Anita S : anita@abc.com : 8987708838 : 30000
A513 : Sam : sam@xyz.com: 987775577 : 10000
X673 : Tom : tom@pqr.com : 887675577 : 15000
```

X673 : Ambica : ambica@pqr.com : 887674474 : 45000

Read this file and calculate the bonus for each employee. The bonus should be 50% of the salary if the salary is less than 20,000 otherwise it should be 30%. Display the name, phone number and bonus amount for each employee.

```
with open('employees_info.txt', 'r') as f:
    for line in f:
        _, name, _, phone, salary = line.split(':')
        salary = float(salary.strip())
        bonus = 0.5 * salary if salary < 20000 else
0.3 * salary
        print(name, phone, bonus)
```

Each line is split into individual fields by using the `split` method. Since we do not need the employee ID and email ID, we have ignored them by using underscores. Any spaces from the `salary` string are removed using the `strip` method and then it is converted to float value. After this the bonus is calculated with the help of if else operator.

11. Read the file `employees_info.txt` given in the previous problem and write its information in another file `employees1.txt` in formatted way.

-----employees1.txt-----

A231 : Raman : raman@xyz.com : 9988008898 : 25000

F632 : Anita S : anita@abc.com : 8987708838 : 30000

A513 : Sam : sam@xyz.com : 987775577 : 10000

X673 : Tom : tom@pqr.com : 887675577 : 15000

X673 : Ambica : ambica@pqr.com : 887674474 : 45000

```
with open('employees_info.txt', 'r') as f,
open('employees1.txt', 'w') as f1:
    for line in f:
```

```
        empid, name, email, phone, salary =  
line.rstrip().split(':')  
  
        print(f'{empid:5} : {name:10} : {email:18}:  
{phone:15}:{salary:>9}', file=f1)
```

We have used the `print` function to write the formatted f string to the file.

12. The following file `information.txt` contains questions, options for the questions and correct answers on separate lines.

-----information.txt-----

Which of these is the exponentiation operator in Python

A. % B. ^ C. * D. **

D

What is returned from a function that does not have a return statement

A. 0 B. None C. Nothing

B

Python is a case sensitive language.

A.True B.False

A

Write a quiz like program that shows the questions to the user one by one, and checks the answer entered by the user.

```
with open('information.txt','r') as f:  
    while True:  
        question = f.readline().strip()  
        if question == '':  
            break
```

```

options = f.readline().strip()
answer = f.readline().strip()
print(question)
print(options)
response = input('Enter your answer : ')
if response.strip().upper() == answer:
    print('Your answer is correct\n')
else:
    print('Correct answer is ', answer)
print(f.readline())

```

13. The following file named `students_info.txt` contains student records on separate lines. Each record contains roll number, name, subject-marks pairs and email ID. Colon is used to separate these fields and '-' is used to separate subject and marks in the subject-marks pairs. The number of subject and marks pair can be different for every student.

-----students_info.txt-----

23412 : Deep : Bio-55: Maths-97: Chem-78: Eng-98: deep@yahoo.com

23413 : Dev : Comp-45:Maths-97: Bio-78: dev@yahoo.com

23413 : Anand: Maths-62:Eng-45:Comp-45: anand@yahoo.com

48135 : Kiran : Maths-39:Bio-67: Comp-78: Eng-98: Science-23: French-45:
kiran@yahoo.com

23412 : Harsh : Bio-45: Maths-67: Chem-78: Eng-98: harsh@yahoo.com

23413 : Sheetal : Comp-95:Maths-87: Bio-88: Chem-38: French-45:
sheetal@yahoo.com

23413 : Sukhi : French-32:Eng-35:Comp-45: sukhi@yahoo.com

48136 : Khushi : Maths-99:Bio-97: Comp-78: Eng-98: Science-67:
khushi@yahoo.com

Read this file and calculate the percentage marks for each student. Create another file named `results.txt` and write the roll number, name and percentage of each student in this file.

```
with open('students_info.txt', 'r') as f1,
open('results.txt', 'w') as f2:
    for line in f1:
        rollno, name, *pairs, _ = line.split(':')
        total = 0
        for pair in pairs:
            _, marks = pair.split('-')
            marks = int(marks.strip())
            total += marks
        percentage = total / len(pairs)
        print(f'{rollno} {name:12}
{percentage:8.2f}% ', file=f2)
```

The individual fields in a line are separated using `split` and the subject marks pairs are stored in variable named `pairs`. This `pairs` tuple is iterated over to get the marks which are then added to get the total and percentage.

14. In the previous program, we wrote the result in a single file. Change the program so that the result is now written in three separate files.

If the percentage is greater than or equal to 80 write the result of the student in `highperformers.txt`.

If the percentage is less than 80 but greater than or equal to 50, write the result of the student in `potentialperformers.txt`.

If the percentage is less than 50, write the result of the student in `lowperformers.txt`.

```
with open('students_info.txt', 'r') as f1, \
        open('highperformers.txt', 'w') as f2, \
```



```

        open('potentialperformers.txt', 'w') as f3,
\
        open('lowperformers.txt', 'w') as f4:
    for line in f1:
        rollno, name, *pairs, _ = line.split(':')
        total = 0
        for pair in pairs:
            _, marks = pair.split('-')
            marks = int(marks.strip())
            total += marks
        percentage = total / len(pairs)
        if percentage >= 80:
            print(f'{rollno} {name:12}'
{percentage:8.2f}% ', file=f2)
        elif percentage >= 50:
            print(f'{rollno} {name:12}'
{percentage:8.2f}% ', file=f3)
        else:
            print(f'{rollno} {name:12}'
{percentage:8.2f}% ', file=f4)

```

13.16 File Related Modules

There are some files related built in modules in Python that you can use in your programs to manipulate files and directories on your file system. You can use Python documentation to explore different features of these modules when you need them. Here are a few functions from some of these modules.

The `os` module can be used for performing various file processing operations like renaming or deleting files.

<code>os.rename(current, new)</code>	Renames file or directory
<code>os.remove(filename)</code>	Removes the file
<code>os.getcwd()</code> current working directory	Returns the path of the
<code>os.chdir(path)</code> directory	Changes our current working
<code>os.rmdir(directory_name)</code> our current directory, to remove from any other place you have to specify the full path	Removes the directory from
<code>os.listdir(path)</code> given directory	Returns a list of all entries in the
<code>os.scandir(path)</code> given directory, available from version 3.5	Returns iterator of all entries in the
<code>os.makedirs(path)</code> exist) in the path specified	Creates all directories (if they do not
<code>os.mkdir(path)</code> in the path	Creates only the rightmost directory

There are many other functions also in this module related to directories.

Here are some functions available in the `os.path` module.

<code>os.path.basename(path)</code>	Returns the base filename from the path
<code>os.path.dirname(path)</code> path	Returns the directory name from the
<code>os.path.exists(path)</code> to an existing file or directory.	Returns True if the pathname refers
<code>os.path.isdir(path)</code> to an existing directory	Returns True if the pathname refers
<code>os.path.isfile(path)</code> to an existing regular file	Returns True if the pathname refers

<code>os.path.getatime(path)</code>	Returns the last access time of a file
<code>os.path.getmtime(path)</code> a file	Returns the last modification time of
<code>os.path.getsize(path)</code>	Returns the size of a file, in bytes
<code>os.path.split(path)</code> the base filename	Splits a path into the directory and
<code>os.path.abspath(path)</code> absolute path	Converts a relative path to an
<code>os.path.isabs(path)</code> absolute path	Returns True if the path is an
<code>os.path.join(string1, string2, ...)</code> a path using a path separator ('\' or '/') appropriate for the platform	Joins components of

Here are some functions from the `shutil` module. You can use it to move or copy a file, and delete directory trees.

<code>shutil.move(source, destination)</code> source to destination	Moves a file or directory
<code>shutil.copy(source, destination)</code> destination	Copies source file to
<code>shutil.copytree(source, destination)</code> tree rooted at source to destination directory	Copies the directory
<code>shutil.rmtree(path)</code> the path	Recursively deletes the directory tree rooted at

The module `glob` provides wildcard handling for filenames. To create and extract ZIP files, you can use the `zipfile` module.

13.17 Command Line Arguments

While running your program from the system command line (shell prompt or command prompt window), you can send arguments which can be accessed inside your program. These are called command line arguments and they are

a way to provide additional information to the program at the start-up. Command line arguments are analogous to function arguments. Function arguments can be different each time the function is run. Similarly, command line arguments can be different each time the program is run. These command line arguments make your program more general and flexible because they are not hardcoded inside the program. Without changing the code, you can control how your program is run by providing a different input each time it is run.

For accessing and using command line arguments, you need to import the `sys` module. All the command-line arguments are stored in the list `sys.argv`. To get a count of the total number of command-line arguments, you can write `len(sys.argv)`. The first argument `sys.argv[0]` is always the name of the program that is being executed.

Here is a simple 2 line code that is contained in the file named `sample.py`.

-----sample.py-----

```
import sys
print(sys.argv)
```

To run this program from the system command line, open your command line window, navigate to the directory where this program is saved and run your Python program.

```
C:\Users\Deepali\Programs>python sample.py
['sample.py']
```

In our program we have printed the list `sys.argv`, so we get a list which has only one element that is the name of the program file. Now let us run this again with some more command line arguments.

```
C:\Users\Deepali\Programs>python sample.py first
second 33 44
['sample.py', 'first', 'second', '33', '44']
```

Now the list `sys.argv` contains five strings, the first element is always the name of the file and then there are other arguments that we have written at the command line. We can see that all the command line arguments are stored as strings in the list `sys.argv`. Most of the time, you would want to

access only those arguments that follow the script name; for that, you can use the slicing operator in your program.

-----sample.py-----

```
import sys
print(sys.argv[1:])
```

By using the slice, we are printing all the elements of the list except the first element which is the name of the file. When we run this on the command line, we will not get the script name in the list.

```
C:\Users\Deepali\Programs>python sample.py first
second 33 44
```

```
['first', 'second', '33', '44']
```

All the command line arguments are stored as strings in the list `sys.argv`. If we want any argument to be of another type, we have to use an appropriate conversion function, such as `int()` or `float()`.

Now let us make use of these command line arguments in the file programs that we have seen earlier.

We have seen the following program ([Section 13.15, Question 2](#)) that counts the number of lines in the file `names.txt`. Now, we will change this program so that we can send the name of the file at the command line.

```
with open('names.txt', 'r') as f:
    count = 0
    for line in f:
        count += 1
print(count)
```

-----countlines.py-----

```
import sys
with open(sys.argv[1], 'r') as f:
    count = 0
    for line in f:
```

```
        count += 1
print(count)
```

We have to import the `sys` module and instead of `names.txt`, we have written `sys.argv[1]`.

Now, let us run this program from the command line.

```
C:\Users\Deepali\Programs>python countlines.py
data.txt
```

```
10
```

```
C:\Users\Deepali\Programs>python countlines.py
names.txt
```

```
20
```

So now we can get the count of lines in any text file by providing its name at the command line while executing `countlines.py`. This program will give error when you try to execute it from IDLE Run menu or when no filename is provided while executing at the command line. To avoid this, you can put a check in the beginning and ask the user to provide the filename.

```
import sys
if len(sys.argv) == 1:
    filename = input('Enter filename : ')
else:
    filename = sys.argv[1]
with open(filename, 'r') as f:
    count = 0
    for line in f:
        count += 1
print(count)
```

To provide the command line arguments while running the program in IDLE, you can use the Run...Customized option from the Run menu.

In [Section 13.15, Question 5](#), we saw this program that copies the contents from `names.txt` to `new.txt`.

```
with open('new.txt', 'w') as f1, open('names.txt',
    'r') as f2:
    for line in f2:
        f1.write(line)
```

Now we will change the program such that it accepts the names of the files as command line arguments.

-----copy.py-----

```
import sys
with open(sys.argv[1], 'w') as f1,
    open(sys.argv[2], 'r') as f2:
    for line in f2:
        f1.write(line)
```

This program copies the file in `sys.argv[2]` to file in `sys.argv[1]`.

We can run this program on the command line for different file names.

```
C:\Users\Deepali\Programs>python copy.py new.txt
names.txt
```

```
C:\Users\Deepali\Programs>python copy.py
students1.txt students.txt
```

In [Section 13.15, Question 6](#), we wrote a program that appends data from multiple files at the end of a file. Here is the modified program that accepts the filenames as command-line arguments:

-----append.py-----

```
import sys
with open(sys.argv[1], 'a') as f1:
```

```
for file in sys.argv[2:]:
    with open(file, 'r') as f2:
        for line in f2:
            f1.write(line)
```

Here `sys.argv[1]` is the name of the file to which we want to append multiple files. All the command line arguments after this will be the names of those files which are to be appended to this file.

```
C:\Users\Deepali\Programs>python append.py
names.txt names1.txt names2.txt
```

```
C:\Users\Deepali\Programs>python append.py
names.txt names1.txt names2.txt names3.txt
```

If you want, you can place a check on the number of command line arguments by using the expression `len(sys.argv)`.

For simple cases, iterating over the `argv` list and accessing the arguments is fine. Python can accept command line options also. For more advanced parsing of the command line options and arguments, you can use the module `argparse`.

13.18 Storing and Retrieving Python objects using pickle

We have seen how to read and write string data in files. We may want to save different types of Python objects like lists and dictionaries also to a file so that they exist even when we close the program, and we can reload them whenever we want. To write any Python object to a file using the `write` method, it has to be converted to a string first. In the following code we are trying to write an integer, a float and a list into a text file using the `write` method, but it fails.

```
with open('data.txt', 'w') as f:
    f.write(23)
    f.write(2.5)
```



```
f.write([10, 20, 30])
```

Output-

TypeError: write() argument must be str, not int

We cannot write these Python objects using the `write` method, they have to be converted to `str` type before writing.

```
with open('data.txt', 'w') as f:
    f.write(str(23))
    f.write(str(2.5))
    f.write(str([10, 20, 30]))
```

Now these objects are written to the file, let us read the data back from the file.

```
with open('data.txt', 'r') as f:
    print(f.read())
```

Output-

```
232.5[10, 20, 30]
```

We get back a string and the type information of the objects is lost. All reading functions give back the data in the form of strings, so getting back the Python objects from the file, requires another conversion which may not be possible always. Also, things get more complicated when you want to write bigger and complex Python objects like nested dictionaries or may be class instances (that we will study later on). There is a simpler way of doing this which does not require us to convert Python objects back and forth to string form.

The built-in module named `pickle` automates the process of reading and writing Python objects into files. This module allows us to store any Python object in the file without converting it to a string, and thus the type information of the object will not be lost. It is called pickling as it preserves your Python objects so that you can use them later on. The pickled objects can be stored in a file or they can be sent over a network.

Pickling is also called serialization as your Python object is turned into a stream of bytes and this serialized byte stream is written to the file. While reading from the file, the reverse operation is done which is called unpickling or deserialization; the stream of bytes is converted back to Python object. This `pickle` module knows how to convert any Python object into a byte stream and how to reconstruct the object back from that byte stream.

While pickling you need to open the file in binary mode, because the Python object is converted to a stream of bytes and that byte stream is written to the file. The objects are stored in a binary format, so you need to work in binary mode while pickling.

The function `dump` of the `pickle` module is used to write a Python object to the file. To read the Python object back into your program, you can use the function `load` from the `pickle` module.

Pickling Unpickling

```
import pickle    import pickle
file = open('data.pck', wb)    file =
open('data.pck', rb)
pickle.dump(p, file)    o = pickle.load(file)
```

For pickling a Python object, you need to first import the `pickle` module and then open a file in binary mode for writing and then dump the Python object. For unpickling, you need to open a file in binary mode for reading and then call the `load` function. The name of the object is not saved, only the object is saved in the file. The `load` function returns us that object and we can assign it to a name in our program. Here is an example of storing a list object in a file using the `pickle` module:

```
import pickle
numbers = [10, 20, 30]
with open('data.pickle', 'wb') as file:
    pickle.dump(numbers, file)
```

The function `dump()` writes the pickled representation of the list object to the file. Now, let us read this file in another program.

```
import pickle
with open('data.pickle', 'rb') as file:
    a = pickle.load(file)
print(type(a))
print(a)
```

Output-

```
<class 'list'>
[10, 20, 30]
```

The function `pickle.load` will reconstruct the list object from the byte stream inside the file, and it will return that object. We have assigned that returned object to name `a`. When we print the type of `a`, we can see that it is a list and printing `a` gives us the list that we had stored in the file using the dump function.

If we want to pickle multiple objects, we can pack them into another object and pickle them. The following code stores two lists and a dictionary in the file by packing them in a tuple:

```
import pickle
names = ['John', 'Bob', 'Tom']
data = {'a':1, 'b':2, 'c':3}
numbers = [10, 20, 30]
with open('data.pickle', 'wb') as file:
    pickle.dump((numbers, names, data), file)
```

While reading this file, we can separate the data in our program by unpacking the tuple read from the file.

```
import pickle
with open('data.pickle', 'rb') as file:
    a, b, c = pickle.load(file)
print(a, b, c)
```

Output-

```
[10, 20, 30] ['John', 'Bob', 'Tom'] {'a': 1, 'b': 2, 'c': 3}
```

The `load` function returned a tuple object which we unpacked into variables `a`, `b`, and `c`. This way we got our two lists and the dictionary back. So, this is how we can dump multiple objects and load them back.

You can also pickle multiple objects by dumping them one after the other, and the objects will be loaded in the order they were pickled. For example, we could pickle our list objects and dictionary object one by one.

```
import pickle

names = ['John', 'Bob', 'Tom']
data = {'a': 1, 'b': 2, 'c': 3}
numbers = [10, 20, 30]

with open('data.pickle', 'wb') as file:
    pickle.dump(numbers, file)
    pickle.dump(names, file)
    pickle.dump(data, file)
```

We can read them back by the following code:

```
import pickle

with open('data.pickle', 'rb') as file:
    a = pickle.load(file)
    b = pickle.load(file)
    c = pickle.load(file)

print(type(a), type(b), type(c))
print(a, b, c)
```

Output-

```
<class 'list'> <class 'list'> <class 'dict'>
```

```
[10, 20, 30] ['John', 'Bob', 'Tom'] {'a': 1, 'b': 2, 'c': 3}
```

Each time we call `load`, we get a value from the file with its original type information intact. So, we can pickle multiple objects in this way also, but pickling them by packing them together in a tuple is better way as you have to load just one object from the file. In our next example, we have pickled a nested dictionary.

```
import pickle

students = {105416: {'name': 'John',
                    'gender': 'M',
                    'city': 'Paris',
                    'age': 21,
                    'marks': {'Maths': 89,
                              'Physics': 78,
                              'Chemistry': 91},
                    'is_sporty': True},
            144547: {'name': 'Dev',
                    'gender': 'M',
                    'city': 'London',
                    'age': 23,
                    'marks': {'Maths': 88,
                              'Physics': 77,
                              'Chemistry': 98},
                    'is_sporty': False},
            132399: {'name': 'Mary',
                    'gender': 'F',
                    'city': 'Paris',
```

```
        'age': 22,  
        'marks': {'Maths': 99,  
                  'Physics': 87,  
                  'Chemistry': 88},  
        'is_sporty': True}  
    }  
  
with open('students.pickle', 'wb') as file:  
    pickle.dump(students, file)
```

We can easily read the dictionary back from the file into our program by using the load function.

```
import pickle  
  
with open('students.pickle', 'rb') as file:  
    d = pickle.load(file)  
    print(d)
```

We cannot search inside an object stored in a pickled file; we have to read the whole object in memory to access it. For example, if we want to search something in this dictionary stored in the pickle file, we will have to read the dictionary in our program and then perform the search.

The process of pickling is done using the pickle protocol which is Python specific, you can read and reconstruct a pickled object only through a Python program. There are different protocol versions, and an object pickled using a newer protocol version may not be unpickled with an older version. The dump (and dumps) functions use the latest version of the pickle protocol, if you want to use another protocol, you can send it as an argument to these functions.

Pickling should not be used to unpickle data from untrusted data sources as it may contain malicious content which can be used to execute harmful code while unpickling. Unpickling untrusted data can be a security risk.

If instead of sending serialized data to a file you want to store it in a bytes object in memory, then you can use the functions dumps and loads from

the `pickle` module. The `dumps` function serializes the object like the `dump` function but instead of writing it to a file, it returns the pickled representation of the object as a `bytes` string. The `loads` function performs deserialization like the `load` function, but instead of reading the serialized data from a file, it reads serialized data from a `bytes` object (like the one returned by `pickle.dumps` function) and returns the reconstituted Python object. Here is an example that uses the `dumps` and `loads` functions:

```
import pickle
numbers = [10, 20, 30]
s = pickle.dumps(numbers)
print(type(s))
x = pickle.loads(s)
print(type(x))
print(x)
```

Output-

```
<class 'bytes'>
<class 'list'>
[10, 20, 30]
```

Exercise

1. What will the following code print?

```
with open('data.txt', 'a+') as f:
    print(f.read())
```

(A) Empty string (B) Contents of the file

2. When you need to add some information to a logfile, which mode will you use to open your file?

(A) 'w'

(B) 'a'

(C) 'r'

3. If you use 'a+' mode for opening a file, then you are working in

(A) Text mode (B) Binary mode

4. 'r+' mode works only on existing files.

(A) True (B) False

5. To empty the buffer without closing the file, which method will you use:

(A) empty (B) flush

(C) clear

6. `fseek(0, 2)` takes the cursor:

(A) to the beginning of the file

(B) to the end of the file

(C) 2 bytes away from the beginning of the file

7. `f.seek(-5, 1)` takes the cursor:

(A) 5 bytes forwards from the current position

(B) 5 bytes backwards from the current position

(C) 5 bytes backwards from the end

8. In binary mode, the `read` method returns a string of type:

(A) `str` (B) `bytes`

9. Which mode should be used for pickling objects.

(A) Text mode (B) Binary mode

10. In text mode, the `write` method adds a newline character at the end of the string that it writes.

(A) True (B) False

11. Which expression will give you a list of all command line arguments except the program name?

(A) `sys.argv[-1:]`

(B) `sys.argv[1:]`

(C) `sys.argv[:1]`

12. Which loop is more efficient?

(A)

```
for line in f:
    print(line)
```

(B)

```
for line in f.readlines():
    print(line)
```

13. Write a program to display only those lines from a file that do not start with #.

14. Write a program to display only the first 5 lines of a file.

15. Write a program to display only the last 5 lines of a file.

16. Write a program to copy the contents of one file to another file such that each space in first file is replaced with a dash in the copied file.

17. Write a function `copy_file` that takes source and destination file names and copies the file by copying 100 characters at a time.

18. Write a program to compare two files line by line and report the line number where they first differ.

19. Write a program to compare two files line by line and display all the lines which are different.

20. In the exercise on loops, we wrote a program to count the frequency of each word in a string. Now, write a program to count the frequency of each word in a file.

21. What will be the problem, if you use the expression `line[:-1]` instead of `line.rstrip()` in the following code?

```
with open('data1.txt', 'r') as f:
    lines =[line.rstrip() for line in f]
    print(lines)
```

22. Write a program to add an empty line after each line in the file.

23. Write a program to search for a string in all the files of a directory.
24. Write a program to delete lines that start with #.
25. What will be the output of the following code?

```
with open('data.txt', 'r') as f:
    print(f.read())
    print(f.read().lower())
```

26. From the following file `students_info.txt`, create another file named `sorted_student.txt` that contains the student records in sorted order.

----- students_info.txt-----

Khushi : Female : khushi@yahoo.com : 9877898998

Deepak : Male : deep@yahoo.com : 988898995

Zeba : Female : zeba@yahoo.com : 988894598

Dev : Male : dev@yahoo.com : 988898228

Anand : Male : anand@yahoo.com : 988845998

Kiran : Female : kiran@yahoo.com : 988678998

Harsh : Male : harsh@yahoo.com : 988897898

Sheetal : Female : sheetal@yahoo.com : 988008998

Sukhi : Male : sukhi@yahoo.com : 988898228

Harsh : Male : harshk@yahoo.com : 987897898

27. Write a program to search for a name in the `students_info.txt` file given in the previous question. Display the whole record of the student if the name is found. If there is more than one record with that name, display all of them.
28. From the file `students_info.txt` given in *Question 26*, create two separate files for records of male and female students. Name the files `boys.txt` and `girls.txt`
29. Write a program that behaves like the mail merge feature of MS Word. Use the files `invitation.txt` given below and

students_info.txt from question 26 to generate different files that serve as personalized invitation letters for different students.

-----invitaton.txt-----

We are delighted to invite you to our upcoming Student Orientation Program at XYZ University. This event will take place on 4th September 2023 at the XYZ University campus.

Warm Regards

XYZ University

Here are the types of files that will be generated by the program.

-----DeepakInvitation.txt-----

Dear Mr Deepak,

We are delighted to invite you to our upcoming Student Orientation Program at XYZ University. This event will take place on 4th September 2023 at XYZ University campus.

Warm Regards

XYZ University

-----KhushiInvitation.txt-----

Dear Ms Khushi,

We are delighted to invite you to our upcoming Student Orientation Program at XYZ University. This event will take place on 4th September 2023 at XYZ University campus.

Warm Regards

XYZ University

30. Write a program to add a copyright text at the end of each .py file in your current directory.

Project : Hangman Game

In this project, we will implement the game of Hangman. First, let us see how this game is played. It is a word guessing game played on paper by

generally two players. One player thinks of a secret word and the other player tries to guess that word by guessing individual letters.

When the game begins, player1 who has the secret word draws a row of dashes where each dash represents a letter of the secret word. So, if the length of the secret word is 8, then he draws a row of 8 dashes. He, also, draws a frame on which he is going to draw the hangman.

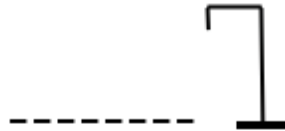


Figure 13.4: Initial hangman drawing

Player 2 starts guessing letters one at a time. For each correct guess player1 places the letter in these empty dashes where the letter appears in the word. For an incorrect guess, which means that the letter is not in the word, he draws a body part of the hangman.

Player 2 wins if the word is fully guessed before the hangman figure is complete, and he loses if the hangman is completely drawn before the full word is guessed. So, player2 can keep guessing letters only till the hangman diagram is not complete.

Player2 can also attempt to guess the full word at any time in the game.

The body of hangman consists of a head, a chest, two arms, a tummy, and two legs i.e., in total seven body parts. So, the guessing player can make maximum seven incorrect guesses. There can be different variations in the figure of the hangman; if you want to give more chances to the guessing player, then you can draw more parts in the figure. We will stick to seven parts, so the guessing player can make only seven incorrect guesses. Let us see some examples of how it is played.

Player1 thinks of a nine lettered secret word and he draws nine dashes and the frame. Player2 starts guessing the letters:

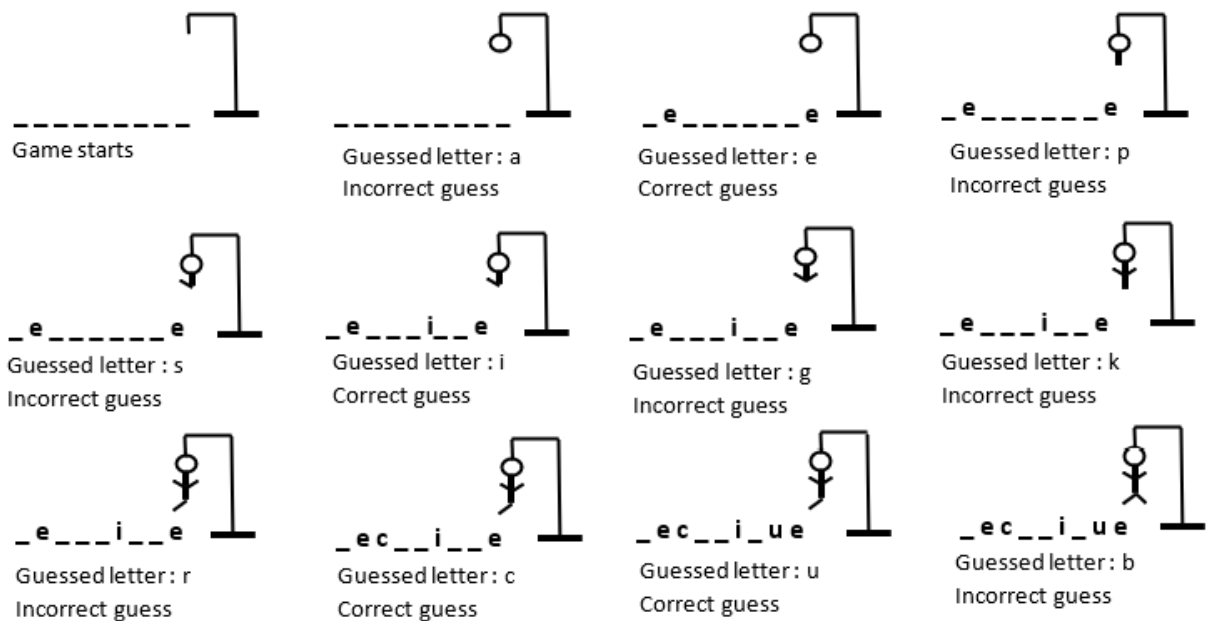


Figure 13.5: Player 2 loses the hangman game

Player 2 loses the game as the hangman figure is complete and he was not able to guess the word. The secret word was technique. In the next example, player1 thinks of an eight lettered word and draws eight dashes and a frame.

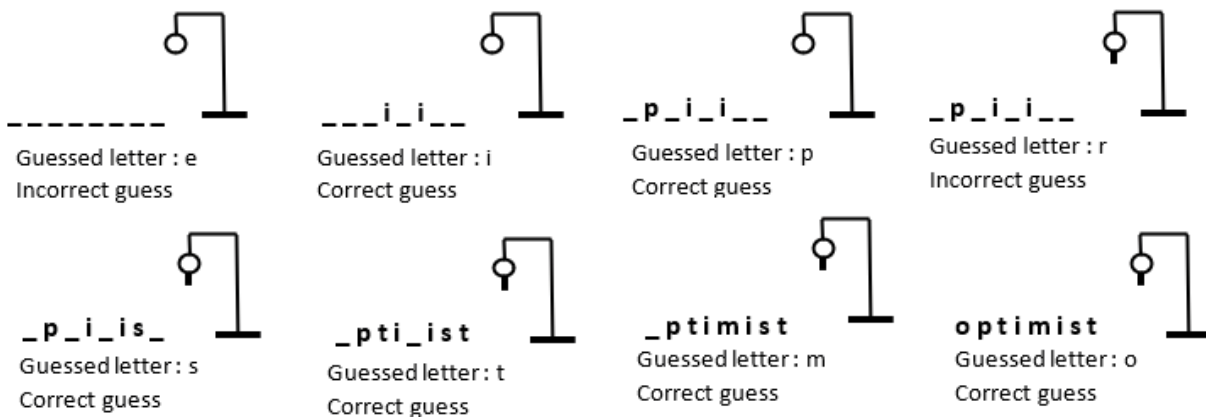


Figure 13.6: Player 2 wins the hangman game

Player 2 wins the game as the word has been completed before the completion of hangman figure.

This is how the game is played, you can start writing the code on your own. If you do not have any idea about how to start, you can look at the implementation given next.

In our implementation, computer will be the player who gets the secret word and the user who runs the program will be the guessing player. We will set the secret word to 'circumference'. After making the program work, we will see how to get a word randomly from a file.

```
print('.' * 50, 'Welcome to HANGMAN', '.' * 50)
secret_word = 'circumference'
play_game(secret_word)
```

The function `play_game` is called with `secret_word` as argument. Inside the definition of this function, we will write the whole logic of playing the game. So, now let us write the code for this function:

```
def play_game(secret_word):
    correct_guesses = ''
    incorrect_guesses = ''
    partial_word = '_' * len(secret_word)
    print(f'Your word is {len(secret_word)}
letters long')
    print('You can make maximum 7 incorrect
guesses\n')

    while len(incorrect_guesses) < 7:
        pass
    else:
        pass
```

We have defined two empty strings named `correct_guesses` and `incorrect_guesses`. In the `correct_guesses` string, we will keep on adding those letters that are guessed by the user and are there in the secret word, and in `incorrect_guesses` string, we will add those letters that are guessed by the user but are not in the secret word.

Next, we have taken a string named `partial_word` for the partially guessed word. Initially this string contains only underscores, and the number of underscores is equal to the number of letters in `secret_word`.

After this, we print a message telling the user the length of the secret word, and that he can make maximum seven incorrect guesses.

Now, we have a `while` loop that will execute until the number of incorrect guesses is less than seven. When the number of incorrect guesses becomes equal to 7, the loop will terminate.

Before writing the body of the loop, let us write the `else` part of the loop. We know that the code in the `else` part executes, only when the loop terminates normally and not due to `break`.

```
while len(incorrect_guesses) < 7:
    pass
else:
    print('You made 7 incorrect guesses')
    print('Now no more attempts left, you have
lost the game')
    print('The word was', secret_word)
```

This `else` part will execute only when the loop terminates normally, that is when the loop condition becomes `False`, and this loop condition will become `False` when incorrect guesses will be equal to 7. So, control will come to the `else` part only when the user has made 7 incorrect guesses, we tell him that he has lost the game and we will also reveal the secret word.

Now let us see what goes in the main body of the loop.

```
while len(incorrect_guesses) < 7:
    guessed_letter = get_guess() # will send some
arguments here
    if guessed_letter in secret_word:
        pass
    else:
        pass
else:
```

.....

In each iteration of this loop, we will get the guessed letter from the user. We will write the `get_guess` function for it. We need to send some arguments to this function; we will see that in a short while. If the guessed letter appears in the `secret_word` we will execute a certain action, otherwise we will execute a different action. First, let us see what we will do when the letter appears in the secret word.

```
    if guessed_letter in secret_word:
        print('Good, you made a correct guess')
        correct_guesses += guessed_letter
        partial_word = get_partial_word() # will send
some arguments here
        print(partial_word)

        if partial_word == secret_word:
            print('Congratulations, you won the
game')

            print('You guessed the word in
{len(correct_guesses)} correct guesses ', end = '
')

            print(f'and {len(incorrect_guesses)}
incorrect guesses')

            break
```

We will tell the user that he made a correct guess. Then we will add the guessed letter to the string `correct_guesses`. The partially guessed word will change, because now this letter will appear in this partial word. So we get the new partial word using the function `get_partial_word()`. We need to send some arguments to this function; we will see that in a short while. After this we will show this partial word to the user.

Now, if the partially guessed word becomes equal to the secret word, it means that all the letters have been filled, then we will tell the user that he has won the game, and we also show the user how many correct and

incorrect guesses he had made. After this, we will put the `break` statement because now we do not want this loop to continue, the game has ended.

Now, let us come to the `else` part of this `if` statement. Control will come here when the guessed word is not in the secret word.

```
    if guessed_letter in secret_word:
        .....
    else:
        print('Sorry, incorrect guess')
        incorrect_guesses += guessed_letter
        print(partial_word)
```

We tell the user that this is an incorrect guess. Then we add the guessed letter to the string `incorrect_guesses`. Now, we show the partial word to the user. The partial word will not change in this case, it will be the same what it was in the last iteration.

So, the `while` loop that we have written can terminate in two cases, one when the user has guessed all the letters, in that case the `break` statement executes and user wins, and the other when the incorrect guesses become equal to 7 in which case the user loses the game.

Now let us write the definition for the function `get_guess` which is responsible for getting the guessed letter from the user. Instead of writing a simple `input` statement to get the letter, we have made this function because we want to validate the user input. If the user enters a string that is not a single letter or is a letter that he has guessed before, we will not accept that input and will ask the user to enter a letter again.

```
def get_guess(letters_guessed):
    if letters_guessed:
        print('Letters guessed already : ', end = '
')
        for letter in letters_guessed:
            print(letter, end=' ')
```

```

        print()
    while True:
        letter = input('Guess a letter : ').lower()
        if len(letter)!=1 or letter not in
'abcdefghijklmnopqrstuvwxyz':
            print('Please enter a single letter')
        elif letter in letters_guessed:
            print('You already guessed this letter
before, enter another letter\n')
        else:
            break
    return letter

```

We need to know the letters that have already been guessed, so here we have the parameter named `letters_guessed`. When we call this function, we will send `correct_guesses + incorrect_guesses` as argument because these two, when joined, give all the letters that have been guessed till now.

```

guessed_letter = get_guess(correct_guesses +
incorrect_guesses)

```

In the function definition, before asking the user for a letter we will show him all the letters that he has already guessed. So, if the parameter string is not empty, we will show all the guessed letters.

Then, in the `while` loop we are asking the user to guess a letter. We convert the letter to lower case, if the length of entered string is not 1 or if the letter is not an alphabetical character, then we ask the user to enter the letter again. The control will go to the `elif` part if the user enters a single letter. In the `elif` part, we check if the entered letter is already there in the guessed letters, if it is present, we print a message and ask the user to enter a letter again.

If both the conditions are False, control will be shifted to the `else` part. It means that we got a valid letter, so then we break out of the loop. This loop

will keep on executing till the user does not enter a valid single letter that has not been guessed before. At the end, we return the letter from the function.

Now, let us write the code for the function `get_partial_word` which is executed when the guessed letter is in the secret word, and we have to get the new partial word.

```
partial_word = get_partial_word(secret_word,
correct_guesses)
```

We will need to send two arguments while calling this function, the secret word and the string that contains the correctly guessed letters.

```
def get_partial_word(secret_word, correct_guesses):
    partial_word = ''
    for letter in secret_word:
        if letter in correct_guesses:
            partial_word += letter
        else:
            partial_word += '_'
    return partial_word
```

We have a variable named `partial_word` and initially we take this to be an empty string. Then we iterate over the string `secret_word`. If the letter in the `secret_word` is present in `correct_guesses` string, then we place the letter in `partial_word`, otherwise we place an underscore. At the end we return `partial_word`.

In the function `play_game`, we are printing `partial_word` in two places. The `partial_word` includes underscores and when we print two underscores together there is no space visible between them so we will print partial word with spaces in between. Instead of putting the code at two places, let us create a function:

```
def print_with_spaces(string):
    print()
```

```

for ch in string:
    print(ch, end=' ')
print('\n\n')

```

This function prints a string with a space after each character of the string. Now, instead of `print(partial_word)`, we will call `print_with_spaces(partial_word)`

Now, we have a basic implementation of the hangman game. Before adding more features to it, we can execute it and see if it is working fine.

It is possible that the user is able to guess the whole word after filling in some letters only, so in that case he does not need to fill in all the correct letters. He can just guess the whole word and win. Let us see how we can do this in our code.

Every time the user guesses a correct letter, we will ask him whether he has guessed the whole word. For this we will call another function in this if statement:

```

if partial_word == secret_word or
ask_if_guessed(secret_word) == True:
.....

```

The function `ask_if_guessed` will ask the user whether he has guessed the whole word. If the whole word that he has guessed is correct then it returns `True`, otherwise if he has not guessed the whole word or guessed a wrong word, it will return `False`.

So now the user can win in two cases: when he has correctly guessed all the letters of the word, in which case `partial_word` will be equal to `secret_word`, and the user can also win when after guessing a few letters, he guesses the whole word.

So now let us see the code of the function `ask_if_guessed`

```

def ask_if_guessed(secret_word):
    response = input('If you have guessed the word,
enter it otherwise press Enter : ')
    if response == '':

```

```

        return False
    elif response == secret_word:
        return True
    else:
        print('No this is not the word ....')
        return False

```

First a prompt is displayed. If the user has not guessed the word then he needs to press Enter otherwise he has to enter the word that he has guessed. If user presses Enter, then `response` will be an empty string, and in this case we will return False. If the entered word is equal to `secret_word` then we return True. Otherwise, whatever the user enters will not be equal to the `secret_word` so in that case we return False.

You can execute the modified program and see if the new feature works.

Now, we will add one more feature to this game to make it a little easier for the user. After the user has made 5 incorrect guesses, we will tell him that now he can make only 2 more incorrect guesses and give him a hint about the word.

For this we take a variable named `hint` and send it to the `play_game` function along with `secret_word`.

```

print('.' * 50, 'Welcome to HANGMAN', '.' * 50)
secret_word = 'circumference'
hint = 'enclosing boundary'
play_game(secret_word, hint)

```

Now in the definition of the `play_game` function, we have to add one more parameter.

```

def play_game(secret_word, hint):
    .....

```

Inside the function, after we have checked whether the guessed letter is correct or incorrect, we will check if the number of incorrect guesses has

become equal to 5.

```
while len(incorrect_guesses) < 7:
    .....
    if partial_word == secret_word or
ask_if_guessed(secret_word) == True:
        .....
    else:
        .....
        if len(incorrect_guesses) == 5:
            print('You can make only 2 more
mistakes, here is a hint for you')
            print(f'Meaning of the secret word is -
{hint}\n')
        else:
            .....
```

This will work but there is a problem in this, which we will see in the following sample. Suppose on executing, we enter the letters a, e, t, b, c, o, k. Now we have made 5 incorrect guesses, so the warning message will be displayed and the hint is also shown.

```
You can make only 2 more mistakes, here is a hint
for you
```

```
Meaning of the secret word is - enclosing boundary
```

Now, we make another guess (letter n), which is a correct one. Again, the warning message and hint will be shown. It will keep on showing the hint till we make another incorrect guess. It is because the value of `incorrect_guesses` will remain 5 till we make another incorrect guess. When we make an incorrect guess, the value of `incorrect_guesses` becomes 6 and so the warning and hint are not displayed.

Now, let us see what changes we can make in our code to solve this problem.

```

def play_game(secret_word, hint):
    .....
    hint_shown = False
    while len(incorrect_guesses) < 7:
        .....
        if partial_word == secret_word or
ask_if_guessed(secret_word) == True:
            .....
            else:
                .....
                if len(incorrect_guesses) == 5 and hint_shown
== False:
                    print('You can make only 2 more
mistakes, here is a hint for you')
                    print(f'Meaning of the secret word is -
{hint}\n')
                    hint_shown = True
            else:
                .....

```

We take a Boolean variable and initialize it to False, and once we have displayed the hint we make it True. We have added another condition in the if statement, so now the hint is shown only when the variable `hint_shown` is False.

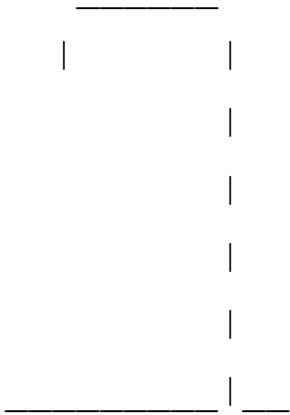
Now, let us make this game more like the one that is played with pen and paper. So, we will draw a body part of the hangman each time an incorrect guess is made. For that we will take a tuple, each item of which is a string that shows a picture of hangman. These strings contain vertical bars, forward and backslashes and capital letter O.

```

hangman_drawings = (

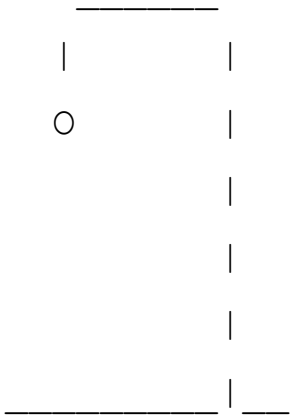
```

'''



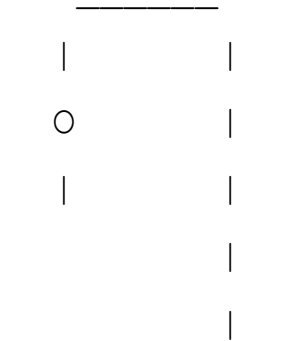
''',

'''



''',

'''



[illegible]

The first string has only the frame, then each subsequent string has a body part added. The last string has the full picture of hangman. Now in the `play_game` method, we will print these strings.

```
def play_game(secret_word, hint):
```

```

.....

print('You can make maximum 7 incorrect
guesses\n')

print(hangman_drawings[0])

while len(incorrect_guesses) < 7:

    guessed_letter = get_guess(correct_guesses
+ incorrect_guesses)

    if guessed_letter in secret_word:

        .....

    else:

        print('Sorry, incorrect guess')

        incorrect_guesses += guessed_letter

print(hangman_drawings[len(incorrect_guesses)])

    print_with_spaces(partial_word)

```

Initially we print the first string (hangman_drawings[0]) that contains only the frame. When the user makes an incorrect guess, we display a string from the tuple.

Now we will put the main code inside a loop so that the user can choose to play the game again.

```

print('.' * 50, 'Welcome to HANGMAN', '.' * 50)

while True:

    secret_word = 'circumference'

    hint = 'enclosing boundary'

```

```

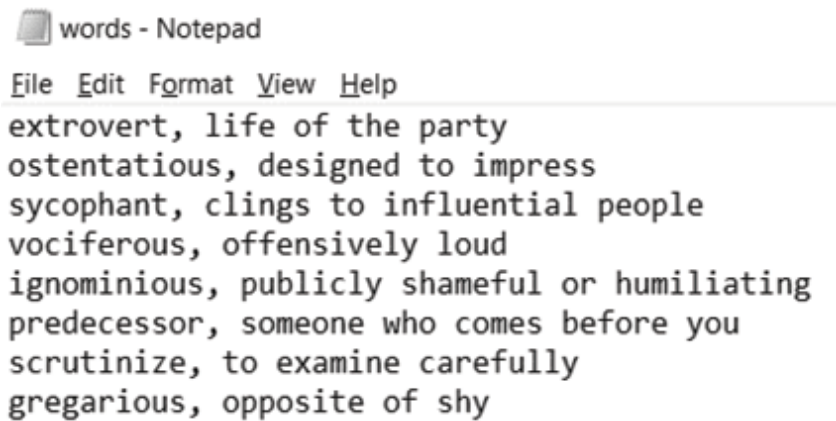
play_game(secret_word, hint)
response = input('\nWant to play again (y/n) :
')

if response == 'n':
    break

```

Now the user can play the game again if he wants. But we have been playing the game with only one word('circumference'). Let us see how we can get a different word each time. We need to store words and their hints somewhere. We can either store them in a list or dictionary inside the program or if we have many words, we can store them in a file and read a word from that file.

We will store the words in a file and will get a random word from the file. We can create a text file in which each line contains a word and the hint separated by a comma. Now, let us see how we can get a random word from these words.



words - Notepad

File Edit Format View Help

extrovert, life of the party
 ostentatious, designed to impress
 sycophant, clings to influential people
 vociferous, offensively loud
 ignominious, publicly shameful or humiliating
 predecessor, someone who comes before you
 scrutinize, to examine carefully
 gregarious, opposite of shy

Figure 13.7: File containing words and hints

We open this file and read all the lines of this file into a list.

```

with open('words.txt', 'r') as file:
    words = file.readlines()

```

To select a random string from this list named `words` we will use the `choice` function from the `random` module.

```

while True:

```

```

    secret_word, hint =
random.choice(words).split(',')
    play_game(secret_word, hint)
    response = input('\nWant to play again (y/n) :
')
    if response == 'n':
        break

```

We have used the `split` function on the result of `choice` function, as each string of the list contains word and its hint separated by a comma. The return value of `split` function is assigned to `secret_word` and `hint`.

So now instead writing a single secret word and its hint in our program, we are getting different secret words and their hints from a file. Now since the user can play the game many times, there are chances that the same word is shown to the user again. We need to make sure that the user never gets the same word again. For this, we will take a list name `used_words` and will append `secret_word` to this list.

```

used_words = []
with open('words.txt', 'r') as file:
    words = file.readlines()
while True:
    secret_word, hint =
random.choice(words).split(',')
    while secret_word in used_words:
        secret_word, hint =
random.choice(words).split(',')
    used_words.append(secret_word)
    play_game(secret_word, hint)
    response = input('\nWant to play again (y/n) :
')

```

```
    if response == 'n':  
        break
```

Now we have made sure that we get an unused word from the list each time. So now we have our full implementation of the hangman game.

If your file `words.txt` file is too long and you do not want to read the whole file in a list, then you can count the number of lines in the file and then select a random line from the file.

```
def get_a_word(number_of_words):  
    with open('words.txt', 'r') as file:  
        x = random.randint(1, number_of_words)  
        for i in range(x):  
            line = file.readline()  
        return line.split(',')  
  
used_words = []  
  
with open('words.txt', 'r') as file:  
    number_of_words = 0  
    for line in file:  
        number_of_words += 1  
  
while True:  
    secret_word, hint = get_a_word(number_of_words)  
    while secret_word in used_words:  
        secret_word, hint =  
get_a_word(number_of_words)  
    used_words.append(secret_word)  
    play_game(secret_word, hint)  
    response = input('\nWant to play again (y/n) :  
' )
```

```
if response == 'n':  
    break
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Object Oriented Programming

14

14.1 Programming Paradigms

Programming paradigm is an approach to organize and structure your code; you can think of it as a way or style of programming. Each paradigm prescribes some design principles and features that define how a program is structured. The three common programming paradigms are procedural programming paradigm, object-oriented programming paradigm (OOP) and functional programming paradigm.

Programming languages are designed such that they provide features to support one or more programming paradigms. Python is a multi-paradigm programming language which means that it supports multiple programming paradigms. A Python programmer has the flexibility to write the program in procedural, functional or object-oriented style. It is up to the programmers to choose the suitable style according to their problem. They can also mix all the approaches to accomplish a specific task, if the need arises.

In the procedural paradigm, there is a step-by-step procedure that is sequentially followed for solving a specific problem. It is implemented through code blocks called functions. The program is organised in a way such that the functions process the data of the program. In object-oriented programming, real world entities or concepts are modelled using objects. An object has both state and behaviour which means it contains both data and code to manipulate that data. In procedural programming, you model your program in terms of functions, while in object-oriented programming you model your program in terms of objects. Functional programming paradigm

is a style of programming that uses built-in higher-order functions. A higher-order function is function that takes another function as an argument or returns it as a result. Functional programming focuses more on ‘what to solve’ rather than ‘how to solve’.

So far, we have been mostly using procedural approach in our programs. Now we will see the object-oriented approach. In this chapter and the next two chapters, we will explore the object-oriented features of Python. Python also supports functional programming with the help of tools like comprehensions, function objects, lambdas, generators, decorators, map, filter etc. Some of these functional features, have already been covered and we will explore the rest of them later in the book.

14.2 Introduction to object-oriented programming

Before learning how to implement object-oriented programming in Python, let us see the common terms used in object-oriented programming. Classes and objects are the two main components of object-oriented programming.

We have been talking about objects right from the introductory chapter. We have seen and used different types of objects like integer object, list object, string object, file object and function object. Each object has a specific type and objects of each type have certain characterises and behaviours which are all predefined. We do not have any control over the structure or behaviour of these objects, these are objects of either built in types or come from libraries. We just use these objects according to our requirement, so we are clients of these built-in types. While writing large and complex programs we will realise that the predefined types do not serve our purpose. For example, if you are creating a graphical game, you would want to have objects representing circles, triangles, players etc; for a mathematical project you might want to have objects representing vectors, matrices etc; for a grocery store application you might want to have objects representing different products, shopping carts and customers.

Object oriented programming allows us to create our own type of objects that would behave the way we want them to behave. We can create our domain specific objects while solving a problem. When we have to create

custom objects that our program requires, we have to define our own types. These new types are called user defined types and are created by defining classes. A class is a blueprint or template for creating objects. A class definition introduces a new type, and it describes the state and behaviour that the objects of this new type will have. Each object that is created from a class will have the data and behaviour specified in the class.

In object-oriented programming, we model our program in terms of objects. So, first we identify the kind of objects that our system will have and then we write class definitions that represent these types of objects. For example, if we are writing a program for a library management system, we might want to have objects that represent different users and books. Each user object will have a name, an ID, and each user can borrow a book or return a book. Each book object contains title of the book, an ISBN number, author name, and a book can be issued or deposited.

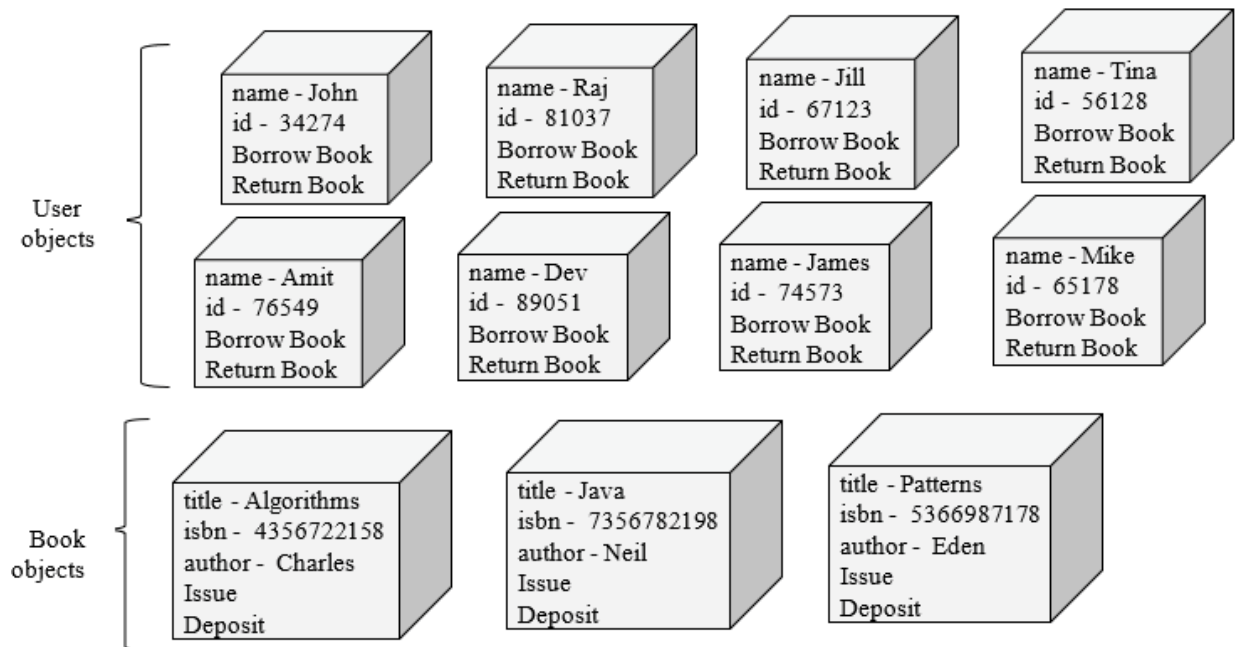


Figure 14.1: Objects

We generally need to model real-world things that have similar behaviour but differ in their internal state, which means that their data is different. For example, all the user objects can borrow a book or return a book, but each one has its own data. Similarly, all the Book objects have similar behaviour but different data.

To create the objects that represent users, we can define a class called `User` and to create objects that represent books, we can define a class called `Book`.

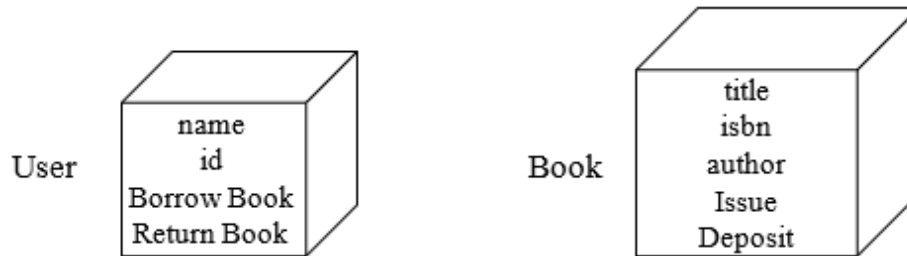


Figure 14.2: User class and Book class

Each class definition introduces a new type and it specifies the data and behaviour that objects created from it will have. Classes encapsulate state and behaviour together - state refers to the internal data stored in the object and behaviour refers to the actions that can be performed by the object. These actions generally act on the state of the object in some way. A class, on its own does not do anything, it is just a template for creating objects, the real work is done by the concrete objects created from the class. You can think of a class as a cookie cutter while the objects created from it are the cookies.

The objects created from a class are called instances or instance objects. Creating a new instance of the class is called instantiation. We can use `User` and `Book` classes to instantiate different user objects and book objects that we saw earlier.

This binding of data and code that acts on that data is called encapsulation. State is maintained through variables which are also called data members and behaviour is implemented through methods. Methods are like functions but they are defined within a class. This concept of encapsulation helps isolate the members of a class. The members of a class are separate from the members of another class and so we can have members with same name in different classes. For example, we can have a data member named `id` in both the `Book` class and the `User` class.

So, a class defines what data and methods should the object have, and the objects contain the actual data. Instantiation means creating an object using a class as the blueprint. The behaviour defined inside the class is shared by all

the objects but data is not. Each object of a specific type behaves in the same way but has its own data. This means that the methods defined inside the class are shared by all the objects, so there is only one copy of each method which is used by all the objects. Each instance object maintains its own copy of data. So, you can think of class as a template that is used to create objects that behave in the same way but have their own data.

Now let us look at some of the benefits of the object-oriented programming approach.

One of the advantages of object-oriented programming is code reusability. Classes that you define can be used multiple times by different applications. You can inherit from these classes to make new classes. This reduces development time and effort and hence lowers the development cost. There are many libraries available that provide classes that can be used by different client programs.

With object-oriented programming, it is easier to represent the real world in code. This modelling of real-world entities and concepts as objects helps in overall understanding of the program code. As your programs get longer, it becomes important to write code that is easier to understand. Better understanding of the code helps in easier debugging, modification and maintenance. In object-oriented programming, we identify the objects that will help in solving our problem and each object is given some responsibility. This structuring of program is more natural to work with and helps to break our program into smaller manageable pieces. So, whenever there is need to fix a bug or add a new feature, the programmer knows exactly where to go, he does not need to go through the entire program. Different parts of the system can be developed and updated independently without affecting the other parts. This also facilitates collaborative development where different teams work on a single project. This is why object-oriented program is well suited for programs that are large and complex and have to be regularly updated.

There is a sort of data security as the data is encapsulated inside the object and hence there are less chances of it being misused by other parts of the program. In procedural programming, your data passes through functions, but in object-oriented approach the data is safely placed inside the object.

Encapsulation also leads to abstraction. While studying functions, we saw that they provided abstraction which means hiding the internal details from the user. Object oriented programming offers a higher level of abstraction. You can hide all the inner working of the class from the user of the class. The user of the class needs to know about only the interface (functionality) of the class, which specifies *what* the class does, not *how* it works. For example, we have been using built in classes like `int`, `list`, `dict` and `str` without knowing their internal implementation details. As users (clients) we just need to be aware of the interface of the class. This information hiding also allows the creator of the class to change the implementation without breaking the client code that uses the class.

Polymorphism, which means one thing many forms, can also be implemented in object-oriented programming. Do not worry if some of the terms do not make sense now, things will become clearer once we start coding.

In the next section we will see how to create classes and objects in Python. Before that, let us clarify the terminology used in Python so that you do not get confused between objects and classes.

We know that everything in Python is an object. Integers, strings, functions, and modules are all objects in Python. When you define a function, a function object is created, similarly when you define a class, a class object is created. The objects that are created by instantiating the class are called instance objects or instances or sometimes simply objects.

14.3 Defining Classes and Creating Instance Objects

In this section, we will see the syntax for defining classes and creating instance objects. A new class is created by writing the `class` statement:

```
class Person:
    pass
```

The keyword `class` is written, followed by the class name and a colon. Conventionally, the class names begin with a capital letter and are generally singular nouns. If there are multiple words in the class name, then they are

joined using the CapWords convention, where the first letter of each word is capitalized.

The header line is followed by an indented block of statements that form the class body. Right now, we do not want to add any data or code to our class, so we have written a pass statement. This makes an empty class. When we execute this class definition, Python creates a class object and assigns it to the name `Person`. This is somewhat similar to what happens when a `def` statement is executed.

We can see the id and type of the class object that is created:

```
>>> id(Person)
2769602751456
>>> type(Person)
<class 'type'>
```

Like everything else, classes are also objects in Python; they are called class objects and their type is `type`. Now, let us see how to create instance objects from this class. A class object is callable, we can instantiate a class object by calling it like a function, i.e., by putting a pair of parentheses around it. The call to class object returns an object which is called the instance of the class.

```
>>> p1 = Person()
```

When this line is executed, an instance object is created whose type is `Person`. That object will be assigned to name `p1`. So, the name `p1` refers to an instance object whose type is `Person`. Let us create one more instance object:

```
>>> p2 = Person()
```

When we execute this statement, another object of type `Person` will be created which will be assigned to name `p2`. We can see types of `p1` and `p2` by using the built-in `type` function.

```
>>> type(p1)
<class '__main__.Person'>
>>> type(p2)
```

```
<class '__main__.Person'>
```

They are objects of type `Person`, let us see their ids:

```
>>> id(p1)
2769564298320
>>> id(p2)
2769601607888
```

We can see that these are 2 different objects in memory. Let us print these objects:

```
>>> p1
<__main__.Person object at 0x00000284D6E56C50>
>>> p2
<__main__.Person object at 0x00000284D91EB8D0>
```

The values shown here are in hexadecimal; in the `id` function, the same numbers were printed in decimal.

14.4 Adding methods to the class

We have seen how to define a class and how to instantiate it, but the class that we have created is useless as it does not have any data or methods. Let us first add behaviour to our class with the help of methods. For that, we will write two `def` statements inside the class:

```
class Person:
    def display(self):
        print('I am a person')
    def greet(self):
        print('Hello, how are you doing?')
```

These definitions of methods look like ordinary function definitions except that there is parameter named `self`. We will talk about this parameter in a short while. You can think of methods as functions inside a class.

To call a method, we will write the instance name, followed by a dot and the method name.

```
p1 = Person()
p2 = Person()
p1.display()
p1.greet()
p2.display()
p2.greet()
```

Output-

```
I am a person
Hello, how are you doing?
I am a person
Hello, how are you doing
```

This is how we can execute the methods using instance objects. You must be wondering how this code executed without any error because both the methods that we defined inside the class have one parameter each, but while calling the methods, we did not send any argument corresponding to the parameter named `self`. This worked because when a class method qualified with an instance is called, Python automatically sends the argument for the parameter `self`.

To see what value Python sends for this parameter, let us print the `self` parameter inside these methods.

```
class Person:
    def display(self):
        print('I am a person', self)
    def greet(self):
        print('Hi, how are you doing ? ', self)

p1 = Person()
p2 = Person()
p1.display()
p1.greet()
p2.display()
p2.greet()
```


Output-

```
I am a person <__main__.Person object at  
0x00000242BD0A7190>
```

```
Hi, how are you doing ? <__main__.Person object at  
0x00000242BD0A7190>
```

```
I am a person <__main__.Person object at  
0x00000242BD0A71D0>
```

```
Hi, how are you doing ? <__main__.Person object at  
0x00000242BD0A71D0>
```

On executing this code, we get objects `p1` and `p2` printed in place of `self`. This means that the instance object that called the method, is printed in the place of `self`. In the first two calls, `self` refers to object `p1`, and in the last two calls, `self` refers to object `p2`. So, now we know that Python provides the instance that calls the method as the argument for the parameter `self`. Although you specify this parameter `self` in the method definition, you do not have to provide a value for it while calling the method.

Generally, all methods inside a class should have this first parameter named `self`. There are some exceptions that we will see later on. Python uses this parameter to identify the instance object that calls the method. You can use any other name instead of `self`, but `self` is a convention widely adopted within the programming community. It is a very strong convention, so it is generally better to adhere to it.

So, in this section, we saw how to add methods to our class. The difference between methods and functions is that methods are always defined inside a class, they are invoked using the dot syntax and in a method definition the first parameter is generally always `self`. Apart from this, whatever features we have seen in [Chapter 10](#), like default values, returning values, variable arguments, etc., hold true for methods, also.

In OOP terminology, sometimes methods are referred to as messages that can be sent to objects. By calling a method, the user (client code) of the object sends a message to the object for performing a task. For example, when we write `list1.sort()`, we were sending a message to the list object to sort its data. Similarly, the code that uses an object of our `Person`

class can send messages to the object by calling the methods `display` or `greet`.

14.5 Adding instance variables

We have added behaviour to our `Person` class in the form of the two methods `display()` and `greet()`. These methods are shared by all the instance objects. Now, we will add data to our instance objects in the form of instance variables. Each instance object will maintain its own data which means that instance variables are not shared, each instance object will have its own copy of instance variables.

An instance variable is created like you create any other variable in Python, by assigning a value to it. But since an instance variable is associated with an instance object, you have to use the dot syntax.

```
>>> p1.name = 'Tom'
```

This statement attaches the instance variable `name` to the instance object `p1`.

```
>>> p1.name  
'Tom'
```

This is called an instance variable because it is attached to an instance. The instance variable `name` has been attached only to `p1`, not `p2`.

```
>>> p2.name
```

```
AttributeError: 'Person' object has no attribute  
'name'
```

We would generally want all the instance objects of a class to have the same variables. So, we will not attach the instance variables dynamically like this outside the class; we will attach them inside the methods. That way, all instance objects created from the same class will have the same set of instance variables.

We know that inside any method, we can access the instance object by writing `self`. So, inside a method if a variable name is prefixed with `self`, then that variable will be an instance variable. We will create a new method

`set_details()`, and inside this method, we will create two instance variables `name` and `age`.

```
class Person:
    def set_details(self):
        self.name = 'John'
        self.age = 20
    def display(self):
        print('I am a person', self)
    def greet(self):
        print('Hi, how are you doing ? ', self)

p1 = Person()
p1.set_details()
p2 = Person()
p2.set_details()
```

After creating the instance objects, we called the `set_details` method for each one. After executing this program, the objects referred to by `p1` and `p2` will have two instance variables each. Let us see their values:

```
>>> p1.name
'John'
>>> p1.age
20
>>> p2.name
'John'
>>> p2.age
20
```

So, whenever an instance object of `Person` class will call the method `set_details`, it will get these two instance variables attached to it.

Instance variables are specific to an instance of the class, every instance has its own copy of instance variables.

Changing the value for one instance does not affect the value in another instance. Let us change `p2.name` to `'Jack'` and `p2.age` to `30`.

```
>>> p2.name = 'Jack'
>>> p2.age = 30
>>> p2.name
'Jack'
>>> p2.age
30
```

Let us check the instance variables of object `p1`.

```
>>> p1.name
'John'
>>> p1.age
20
```

The instance variables of `p1` were not changed. So, each instance object has its own copy of instance variables and these variables define the state of that instance object.

The method `set_details` always sets the name to `'John'` and age to `20`. We would generally want to assign different values to different instance objects. So, to make this method more flexible, we will add two parameters in the definition, `name` and `age`.

```
def set_details(self, name, age):
    self.name = name
    self.age = age
```

We have assigned `name` to `self.name` and `age` to `self.age`. Do not get confused in the two sets of names. `self.name` and `self.age` are instance variables while `name` and `age` are parameters of this method, so they are just local variables inside the method. You can use `name` and `age` only inside this method, but you can use the instance variables in any method inside the class. This is because instance variables are attached to the instance object, and they will live as long as the object lives. They will not be destroyed when the method terminates, as is the case with local variables.

The dot notation makes sure that there is no conflict between the two sets of names. You can use any other name for the parameters, but it is a convention to use the same names as instance variables. Now, when we call `set_details`, we will send two arguments.

```
p1 = Person()
p1.set_details('Bob', 20)
p2 = Person()
p2.set_details('Ted', 90)
```

Now, we are able to give different values for name and age of different instance objects. After executing our modified program, we will see that `p1.name` and `p2.name` are different, and similarly, the age instance variable also has different values for objects `p1` and `p2`.

```
>>> p1.name
'Bob'
>>> p2.name
'Ted'
>>> p1.age
20
>>> p2.age
90
```

So, now each instance object can start with a different state.

After these instance variables have been created, they are available inside the methods of the class (because of `self`) and so any method of the class can use them. Let us use the two instance variables in the methods `display` and `greet`.

```
class Person:
    def set_details(self, name, age):
        self.name = name
        self.age = age
```

```
def display(self):
    print('I am', self.name)
def greet(self):
    if self.age < 80:
        print('Hi, how are you doing?')
    else:
        print('Hello, how do you do?')

p1 = Person()
p1.set_details('Bob', 20)
p1.display()
p1.greet()
p2 = Person()
p2.set_details('Ted', 90)
p2.display()
p2.greet()
```

Output-

I am Bob

Hi, how are you doing?

I am Ted

Hello, how do you do?

In the method `display`, we have used the instance variable `name`, and in the method `greet`, we have used the instance variable `age`. The instance variables `name` and `age` are created in `set_details()` method and referenced in the methods `display` and `greet`.

When you reference an instance variable outside a class, it has to be prefixed with the instance name and a dot (for example, `p1.name` or `p2.age`). Inside the methods, `self` refers to the current instance object (the object that called the method), so the instance variable name is prefixed with `self` and a dot. The `self` parameter helps you access or change the instance variables from within the methods, and this is why `self` is the first parameter in all the methods.

If you have worked in Java or C++, you must have noticed the difference in how the instance variables are defined. In these languages, these instance variables, which are also called data members, are statically declared; they are a formal part of the class definition. They are defined inside the class, outside of any method. In Python, instance variables are defined inside methods and it is possible to even dynamically attach instance variables. The variables that we create outside the methods at the class level are class variables that we will see shortly.

14.6 Calling a method inside another method

Suppose we want to call the method `display` inside the method `greet`. We have seen that, when an instance object calls the `display` method outside the class, it is called like `p1.display()` or `p2.display()`. Inside a method, the current instance is accessed by using `self`, so here we will call it as `self.display()`.

```
def greet(self):
    if self.age < 80:
        print('Hi, how are you doing?')
    else:
        print('Hello, how do you do?')
    self.display()
```

We have called the method `display` with `self`. From outside the class, we will call the method `greet` like `p1.greet()` or `p2.greet()` and inside the method `greet`, the method `display` will be called.

```
p1 = Person()
p1.set_details('Bob', 20)
p1.greet()
p2 = Person()
p2.set_details('Ted', 90)
p2.greet()
```

Output-

Hi, how are you doing?

I am Bob

Hello, how do you do?

I am Ted

So, outside the class, the instance variables and methods will be accessed by preceding them with instance object name. Inside the class methods, they will be accessed by preceding them with the name `self`.

We have seen how to define classes and how to create instance objects. There was a lot of new syntax involved, so let us summarise in a few points, whatever we have studied till now:

- We define a new class by using the `class` statement. When a `class` statement executes, it creates a new class object and binds it to the class name.
- Instantiation of the class creates a new instance object. To instantiate the class, we have to call the class object with a pair of parentheses.
- The instance object is like any other object of Python. It can be used as an element of a list, tuple, dictionary, or set. It can be passed to a function as an argument or can be returned from a function. It is a first-class object in Python.
- Even class objects are first-class objects in Python. They can also be passed as arguments or returned from a function, bound to variables, used as an element in a container, or even an attribute of an object.
- Methods are defined inside the class using the `def` statement, and they follow all the rules that we have studied in functions.
- Inside the method definition, the first parameter should be `self`. You do not have to provide any argument for `self` while calling the method. Python will automatically assign the instance object that calls the method to this parameter `self`. This parameter is always required so that we can access the instance variables and methods of an instance object from within the class.

- Instance variables can be created inside any method by assigning to a variable name prefixed with `self`.

```
self.variablename = value
```

- To reference an instance variable inside any method, you must prefix the variable name with `self`.

```
print(self.variablename)
```

- To call a method inside another method, you must prefix the method name with `self`.

```
self.methodname()
```

- Outside the class, we must use an instance object name before methods and instance variables. Inside the class, we must use `self` in front of the methods and instance variables.

14.7 Common pitfalls

Many programmers who are used to other languages like Java or C++, generally forget to include `self` as the first parameter in the methods. If you forget to do this, the interpreter will complain. Let us remove the `self` parameter from the `set_details` method of our class `Person`.

```
class Person:
    def set_details(name, age):
        self.name = name
        self.age = age
    def display(self):
        print('I am', self.name)
    def greet(self):
        if self.age < 80:
            print('Hi, how are you doing?')
```

```

        else:
            print('Hello, how do you do?')
        self.display()

p1 = Person()
p1.set_details('Bob', 20)
p1.greet()

```

Output-

File "E:\Programs\14_ObjectOriented\P14_8.py", line 18, in <module>

```
    p1.set_details('Bob', 20)
```

TypeError: Person.set_details() takes 2 positional arguments but 3 were given

When we execute the program with `self` parameter removed from the definition `set_details` method, Python shows an error.

We have sent two arguments in the call `p1.set_details('Bob', 20)` but the error message is saying that 3 were given. This shows that Python automatically sends an argument, and so we always need to specify the first parameter as `self`, and after that we can have our regular parameter list.

Another mistake that beginners in Python make, is forgetting to add `self` as a prefix for the instance variables and methods. Let us remove the `self` from the call to `display` that we made in the `greet` method.

```

def greet(self):
    if self.age < 80:
        print('Hi, how are you doing?')
    else:
        print('Hello, how do you do?')
    display()

```

Now, on executing the program we will get the following output:

```
Hi, how are you doing?
```

```
Traceback (most recent call last):
```

```
File "E:\Programs\14_ObjectOriented\P14_9.py",  
line 19, in <module>
```

```
    p1.greet()
```

```
File "E:\Programs\14_ObjectOriented\P14_9.py",  
line 14, in greet
```

```
    display()
```

```
NameError: name 'display' is not defined
```

display is a method of the class so it should be called with an instance of the class. We know that `self` refers to the current instance inside the class, so you need to call it as `self.display()` inside the class.

So, if we forget to use `self` before the method name, we get `NameError`. Similarly, if you forget to use `self` before an instance variable inside a method, then also you will get an error. For example, suppose we forget to write `self` in front of the `age` instance variable:

```
def greet(self):  
    if age < 80:  
        print('Hi, how are you doing?')  
    else:  
        print('Hello, how do you do?')  
    self.display()
```

We will get the following error on executing the program:

```
Traceback (most recent call last):
```

```
File "E:\Programs\14_ObjectOriented\P14_10.py",  
line 19, in <module>
```

```
    p1.greet()
```

```
File "E:\Programs\14_ObjectOriented\P14_10.py",  
line 10, in greet
```

```
    if age < 80:
```

```
NameError: name 'age' is not defined
```

Now, suppose we define another method `get_old` inside our `Person` class.

```
def get_old(self):  
    age = 75
```

Inside this method, we want to change instance variable `age` to 75, but we forget to put `self` before the instance variable `age`. Let us see what happens when we call this method for object `p1`:

```
>>> p1.get_old()
```

When we execute this, we expect a `NameError` but the statement will execute without any error. After execution of the program if we check age instance variable of `p1`, it is still 20. It was not changed to 75.

```
>>> p1.age
```

```
20
```

Let us see what happened here. An assignment was made to name `age` and we know that in Python, a variable is created when it is first assigned, so here the interpreter created a local variable named `age` with value 75. This is why we did not get any error.

If we add `self` before `age`, then the instance variable `age` will be changed.

```
def get_old(self):  
    self.age = 75
```

After executing the modified program, if we call `get_old()` for `p1` and then check the instance variable `age` of `p1`, then we can see the changes.

```
>>> p1.get_old()
```

```
>>> p1.age
```

```
75
```

So, you need to remember to write `self` whenever you have to use an instance variable or a method inside the class. Qualifying every instance variable and method with `self` involves more typing as compared to some other languages but it makes things clear. It helps you distinguish between a local variable and an instance variable and between a method call and a function call. There is no ambiguity; by looking at the code you can tell whether you are referring to an instance variable or a local variable and whether you are calling a function or a method.

14.8 Initializer

We have seen the following class in the previous sections:

```
class Person:
    def set_details(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print('I am', self.name)
    def greet(self):
        if self.age < 80:
            print('Hi, how are you doing?')
        else:
            print('Hello, how do you do?')

p1 = Person()
p1.set_details('Bob', 20)
p1.display()
p1.greet()
p2 = Person()
p2.set_details('Ted', 90)
p2.display()
p2.greet()
```

Whenever we create a new instance object for this class, immediately we have to call the method `set_details`, because when this method will be called, then only the instance object will have its instance variables created. Now suppose that we forget to call `set_details` for the instance object `p2`, and we call the methods `display` and `greet` for it.

```
p1 = Person()
p1.set_details('Bob', 20)
p1.display()
```

```
p1.greet()
p2 = Person()
p2.display()
p2.greet()
```

Output-

I am Bob

Hi, how are you doing?

Traceback (most recent call last):

```
File "E:\Programs\14_ObjectOriented\P14_13.py",
line 23, in <module>
```

```
    p2.display()
```

```
File "E:\Programs\14_ObjectOriented\P14_13.py",
line 7, in display
```

```
    print('I am', self.name)
```

AttributeError: 'Person' object has no attribute 'name'

For object `p1`, we have called the method `set_details` so its instance variables will be created, but for object `p2` we forgot to call this method so its instance variables will not be created. We have called the methods `display()` and `greet()` on object `p2`. The method `display()` wants to access instance variable `name` of `p2`, and the method `greet()` wants to access the instance variable `age` but these instance variables were not created for object `p2`, so we get `AttributeError`.

Therefore, if you forget to call the `set_details` method and call any of the two methods, `greet` or `display`, you will get an error. You must always remember to call the `set_details` method immediately after creating any `Person` object so that the instance variables are created for that particular object and can be used in other methods. Calling this method every time we create an object is cumbersome.

Python has a solution for this. It lets you automate this object initialization task. You can define a method named `__init__` in your class. This method will automatically be called right after the instance has been created.

So, if you have any code you think should be executed just after the object creation, put that code inside this method.

For our `Person` class, we want the code inside the `set_details` method to be executed after we have created the instance object so, let us change the name of the `set_details` method to `__init__`.

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age
```

This name `__init__` is not a convention like naming of `self`, this is a special name and you cannot choose any other name for this method. The two leading and trailing underscores are also important. Because of these underscores, this method is generally called dunder init, where dunder is shortform of double underscore.

Now we can delete the calls to `set_details()` from the program. We do not have to call this method `__init__` explicitly, the interpreter will call it implicitly. Now you must be thinking, when we do not have to call this method, how will we send the arguments for the parameters `name` and `age`. These arguments will be sent when the object is instantiated.

```
p1 = Person('Bob', 20)  
p1.display()  
p1.greet()  
p2 = Person('Ted', 90)  
p2.display()  
p2.greet()
```

Output-

```
I am Bob  
Hi, how are you doing?  
I am Ted  
Hello, how do you do?
```

When you create instance objects, any arguments that you pass to the class are passed to the `__init__` method.

So, we have seen that the initialization work is automatically done by the interpreter if you define the `__init__` method. You can create and initialize all your instance variables in this method. Although instance variables can be created in any other method also, it is more readable and clearer if you create all instance variables in the `__init__` method. Also, there is no risk of the instance variables being accessed before they are defined.

You can also perform any other startup task that you want, in this initializer method. For example, opening a file or setting up a network connection, or connecting to a database.

Like other methods, the first parameter to `__init__` is always `self`. After `self`, other parameters that are coded in `__init__` are generally used to give initial values to instance variables. These parameters can be given default values if required. The instance variables can also be initialized with values that are independent of parameters.

These methods, which have special names and double underscores before and after their names, are called magic methods in Python. The magic is that they are not called directly; they are called automatically in certain contexts. We will learn more about these dunder methods in a separate chapter.

If you have worked in other object-oriented languages, you must be thinking about constructors right from the start of the section. The `__init__` method definitely looks like a Java or C++ constructor, but it would be technically incorrect to call it a constructor of the class because by the time this method is called, the object is already constructed. To construct an instance, the magic method `__new__` is invoked. This method is responsible for creating the object by allocating memory for it. So technically, this is the actual constructor. As a beginner, you won't need to use this method much; it is used while coding metaclasses, which is an advanced topic. The default `__new__` is automatically invoked if we do not provide our version, and in most cases, the default version serves the purpose.

The `__init__` (dunder init) is the initializer method. It is called immediately after the instance is created. It is the first method that is called on the newly created instance object. The `self` parameter passed to this method refers to the newly created object. So, the method `__init__` does not construct the instance object. It initializes an already constructed instance object. In languages like C++ and Java, construction and initialization are a one-step process, but in Python, these two steps are separated.

You can have only one initializer method in a class, as there is no concept of function overloading in Python. However, it is possible to create instance objects with different types of data using class methods, which we will see later in this chapter. Also, you can give default values to parameters to create the illusion of having multiple initializers.

14.9 Data Hiding

Some languages implement data hiding in a class by declaring the data and methods as private or public. Private data and methods can be used only by the methods inside the class while public data and methods can be accessed from outside the class as well. The part of the class that is designated as private can be used inside the class only. In Python, there is no such concept of private or public; it does not enforce any sort of privacy. The access specifiers, public and private, are not available in Python.

First, let us see why there is a need for this distinction between private and public when working with a class. The users of a class are generally called clients and the client code uses the class by instantiating it, i.e., by creating its objects. Suppose we have three clients `Client1`, `Client2`, `Client3` that use a class named `Product` in their applications.

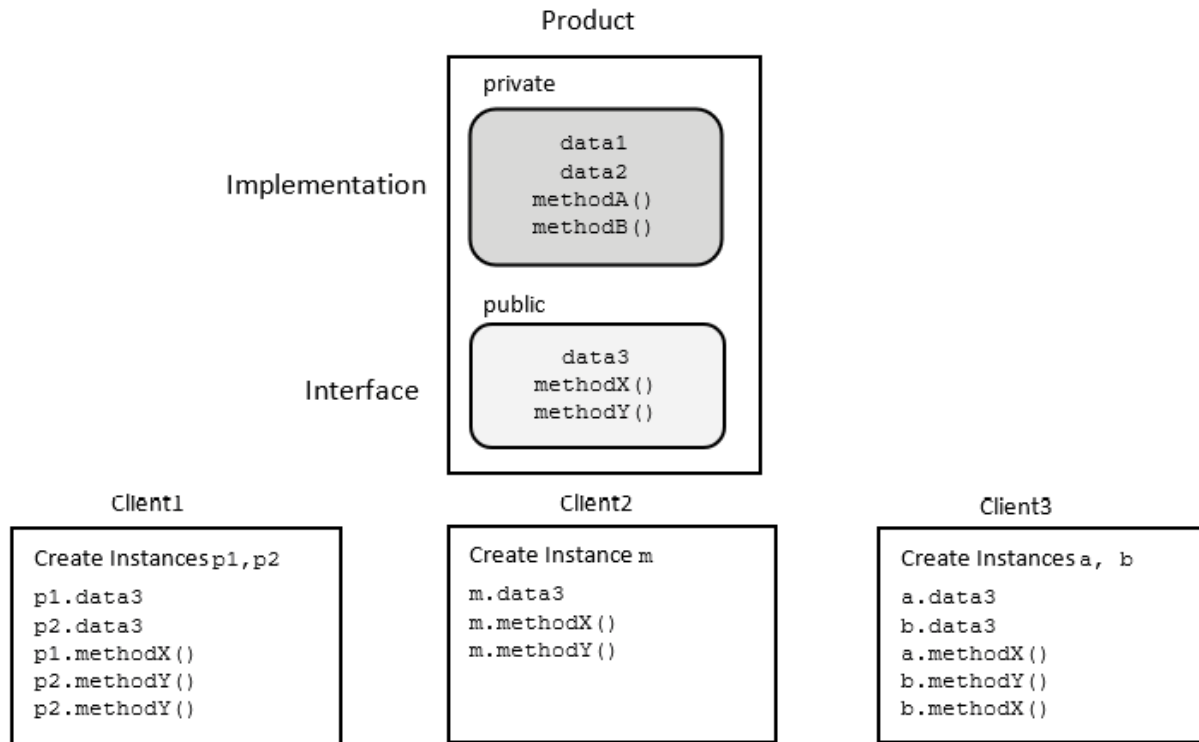


Figure 14.3: Data hiding

These clients can access the variable `data3` and can call the two methods `methodX` and `methodY`, through the instances of the class, but they have no access to variables `data1`, `data2` and methods `methodA` and `methodB`. The creator of the class has chosen to hide them from the user because these things are used in internal working of the class, they are not required by the user. For example, in a car you have access to steering, accelerator and brakes, you do not need access to all the internal parts of the car that make your car move or stop. Those internal details are best left to the creator of the car. If you get access, you might inadvertently damage something and the car will stop working. Moreover, nobody would want to drive a car with all the inner circuitry exposed. It would be very difficult to use such a car.

Similarly, in a class, only those parts are exposed to the user which are required, other internal details are hidden in the form of private variables and private methods. This avoids any confusion and also protects sensitive data that can be inadvertently or maliciously modified by the user. Your objects can be modified in a way that they do not work properly or go into an invalid state. This is why the internal details are not revealed to the user. The part

that is visible to the user is the interface of the class and the part that is hidden is the implementation. Interface of a class allows the programmer to use the class without understanding its internal details.

The interface of a class is well-defined and generally comes with a guarantee that it will not change with time, but the implementation may be changed without any notice. So, a private method or variable may be deleted, or its behavior can be changed without notice. These changes may be done to fix some bugs, change some functionality, or maybe to improve efficiency. Any change in implementation does not affect the client code because the client is not using the implementation part; it is not concerned about how the class is doing its work; it just gets its work done by calling the public methods.

If we again take the car analogy, the internal parts of the car may be changed or can be made to work in some other way but your steering and brakes will work in the same way. So, the interface is generally not changed.

Now, let us come to Python. In Python, everything inside the class is public, clients can access any data or method written inside the class. So, if the clients use the `Product` class written in Python, they are free to call any of the 4 methods and they can access any of the 3 data variables. Python does not enforce any access restrictions on data and methods like Java or C++ do. However, there is a naming convention that is used to indicate that a certain attribute is meant to be used inside the class only and it should not be used directly by the client. The word attribute in Python is used for any name following a dot. So, instance variables and methods are collectively called attributes.

The convention is that you can use a leading underscore on a variable or a method name to suggest that it is private and should not be used outside the class. For example, the names `_phone`, `_age`, `_change()`, `_increase()` indicate that these instance variables and methods are non-public. Variables or methods with a leading underscore should be accessed and modified only inside the methods of the class. They are not meant to be accessed from outside the class. This protects the internal data of the class from intentional or accidental modification.

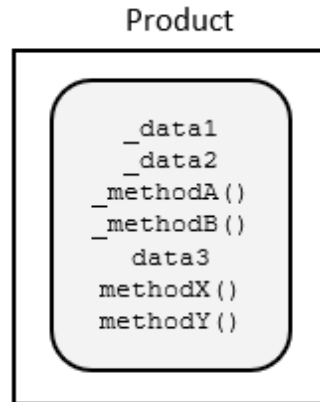


Figure 14.4: Leading underscore indicates privacy

These variables and methods with a leading underscore mean nothing special to the interpreter, they are technically just like any other variable or method, it is possible to access them outside the class also. The leading underscore is there to indicate privacy. This way you can discourage clients from using the private things of a class. However, you cannot stop them from doing so. If you remember, we had seen a similar data hiding convention in modules chapter.

Python works on the policy that we are all consenting and responsible adults and know how to use the code. Its philosophy is based on the trust, that users of the class will respect the convention and documentation and use the methods and variables appropriately.

One of the reasons for making everything accessible outside the class is debugging; when you need to fix a bug, you have to sometimes access the private attributes of the class.

So, if you prefix a variable or a method with a single underscore, then it indicates that this name is non-public, it is only for the internal use of the class and should not be accessed outside it.

If you prefix a name with double underscores, Python will do some name mangling and that attribute will not be directly visible from outside the class. For example, `__value` is internally replaced with `__MyClass__value`, where `MyClass` is the name of the class in which this attribute `__value` is defined. These names are mangled by prefixing with a single underscore and the class name. If you want to use the name `__value`, you will have to write `__MyClass__value`. These names are not directly accessible from

outside the class, but they can be indirectly accessed by using the mangled name.

So, if you use a name that starts with at least two leading underscores and has at most one trailing underscore, that name is mangled by Python, and it cannot be directly used by the user.

This naming can be used for your non-public members of the class to make it difficult for the user to access those members. But this name mangling mechanism is not there in the language for this purpose, its purpose is to make the name specific to the class so that there is no name clash with subclasses (inherited classes). This type of naming should be only used to avoid name clashes with attributes in subclasses. To indicate privacy, you should use single leading underscore. Names with double leading underscores are used to reduce the risk of duplicating the name in subclasses.

There are names that start and end with two underscores, we have seen one such name `__init__` (dunder init), and we will see many more. These types of names are used by Python for its internal use and we should not write our own names that have double leading and double trailing underscores.

A single trailing underscore is used to avoid name clashes with Python keywords and built in names. For example, if you want to use the name `class` or `range` in your program, you can use it as `class_` or `range_`. It is best not to use these names in your programs but if you ever need to do so, the convention is to use a trailing underscore.

Now let us see an example program. We have a class named `Product` in which we have 2 instance variables and 2 methods out of which one variable and one method are prefixed with an underscore which indicates that they are not supposed to be used outside the class.

```
class Product:
    def __init__(self):
        self.data1 = 10
        self._data2 = 20
    def method1(self):
        print('Executing method1')
```

```

        def _method2(self):
            print('Executing method2')
p = Product()
print(p.data1, p._data2)
p.method1()
p._method2()

```

Output-

```

10 20
Executing method1
Executing method2

```

When we execute this program, we do not get any error which means that we can access both variables and call both methods from outside the class. Although the names `_data2` and `_method2` are prefixed with an underscore, it is possible to access them outside the class. For the interpreter, this leading underscore does not make any difference, it is just a convention. Programmers should respect this convention and not access these attributes like this outside the class, unless there is some need for debugging or something similar.

In the above program if we change the single underscores to double underscores, and then execute the program we will get `AttributeError`.

```

class Product:
    def __init__(self):
        self.data1 = 10
        self.__data2 = 20
    def method1(self):
        print('Executing method1')
    def __method2(self):
        print('Executing method2')
p = Product()
print(p.data1, p.__data2)
p.method1()

```

```
p.__method2()
```

Output-

```
AttributeError: 'Product' object has no attribute  
'__data2'
```

If you execute the `dir` function for the instance object `p`, you will be able to see the mangled names. They have been prefixed with the class name and an underscore.

```
>>>dir(p)  
['_Product__data2', '_Product__method2',  
'__class__', 'data1', 'method1']
```

If we want to access them, we can do so with these mangled names.

```
p = Product()  
print(p.data1, p._Product__data2)  
p.method1()  
p._Product__method2()
```

Output-

```
10 20  
Executing method1  
Executing method2
```

We cannot directly access these attributes from outside the class, but we can access them indirectly. Inside the class methods, these variables can be accessed directly. As mentioned before, this naming should be used to avoid name clashes with attributes in subclasses. For making attributes non-public, we should use a single underscore.

Here is an example program that illustrates data hiding:

```
----- student.py -----  
class Student:  
    def __init__(self, name, phone, marks):  
        self.name = name  
        self.phone = phone
```

```

        self._marks = marks
    def _calculate_total(self):
        return sum(self._marks)
    def _calculate_percentage(self):
        return self._calculate_total() / 4
    def display(self):
        print(self.name, self.phone)
    def show_result(self):
        self.display()
        percentage = self._calculate_percentage()
        print(f'Percentage : {percentage : .1f}')
        print('Pass' if percentage > 40 else
'Fail')

```

In this class Student, the instance variable `_marks` and the methods `_calculate_total` and `_calculate_percentage` are not supposed to be used outside the class. The implementation of these three can be changed or they can even be deleted so the clients should not use them in their code. The instance variables `name` and `phone` and the methods `display` and `show_result` can be used by the client.

Generally, classes are written in separate modules and that module is imported in the application program. We have placed our Student class in the `student.py` file and this module will be imported by different applications or clients. We have two client programs that import this Student class.

-----client1.py-----

```

from student import Student
s = Student('Dev', 986754361, [50, 85, 70, 90])
s.show_result()

```

-----client2.py-----

```

from student import Student
s = Student('Raj', 987654535, [73, 89, 78, 88])
s.display()

```



```

if s._calculate_total() > 160:
    print('Pass')
else:
    print('Fail')

```

The code in client1.py instantiates the Student class and then calls the public method show_result. The code in client2.py also instantiates the class and it calls the methods display and _calculate_total. The method _calculate_total was supposed to be private; it shouldn't have been used by the client but the program will work because Python does not enforce any data hiding.

Now, let us consider a scenario where changes are made in the implementation of the Student class after some time. The results are now calculated based on the 'best of 3' approach. _calculate_total will now calculate the total of best 3 subjects and _calculate_percentage will calculate the percentage of these three 3 subjects.

----- student.py -----

```

class Student:
    def __init__(self, name, phone, marks):
        self.name = name
        self.phone = phone
        self._marks = marks
    def _calculate_total(self):
        total_best3 = sum(sorted(self._marks)[1:])
        return total_best3
    def _calculate_percentage(self):
        return self._calculate_total() / 3
    def display(self):
        print(self.name, self.phone)
    def show_result(self):
        self.display()
        percentage = self._calculate_percentage()

```

```
        print(f'Percentage : {percentage : .1f}')
```

```
    print('Pass' if percentage > 40 else
```

```
'Fail')
```

For client1 there will be no problem, he does not need to change his code because he never used any of the private things of the class. The code of client2 will still work but now it has a logical error in it. The `_calculate_total` is now returning the total in 3 subjects and so `if s._calculate_total() > 160:` does not make sense now and has to be changed. The number 160 should be changed to 120.

Now, suppose after some time, grading system is introduced and students are assigned CGPA instead of percentage. So, in the Student class, the methods `_calculate_total` and `_calculate_percentage` are deleted and a new method `_calculate_cgpa` is introduced.

----- student.py -----

```
class Student:
```

```
    def __init__(self, name, phone, marks):
```

```
        self.name = name
```

```
        self.phone = phone
```

```
        self._marks = marks
```

```
    def _calculate_cgpa(self):
```

```
        credit_hours = [3, 3, 4, 2]
```

```
        total_grade_points = 0
```

```
        for i, score in enumerate(self._marks):
```

```
            if score > 90:
```

```
                grade_points = 10
```

```
            elif score > 70:
```

```
                grade_points = 8
```

```
            elif score > 50:
```

```
                grade_points = 6
```

```
            elif score > 30:
```

```
                grade_points = 4
```

```

        else:
            grade_points = 0
            total_grade_points += grade_points *
credit_hours[i]
            cgpa = total_grade_points /
sum(credit_hours)
            return cgpa
def display(self):
    print(self.name, self.phone)
def show_result(self):
    self.display()
    cgpa = self._calculate_cgpa()
    print(f'cgpa : {cgpa : .1f}')
    print('Pass' if cgpa > 4 else 'Fail')

```

Again, client1 has no problem, but the code of client2 now will give an error as now there is no method named `_calculate_total`.

Therefore, it is advisable to avoid using the private attributes of a class. If users choose to use them, they do so at their own risk, the given example illustrates this concern. In real life code, the code of the class and the client code will not be so small, and so the issues resulting from using private attributes can be extensive and challenging to identify and rectify.

14.10 Class Variables

While studying the object-oriented concepts we had seen that the behaviour of all instance object is same while their data is generally different. This is why methods are stored in the class object and shared by all instances while the instance variables are stored in different instance objects.

While modelling our objects, we might find that there is some data that does not vary for each instance, it is the same for every instance created from a particular class. Storing this piece of data in every instance object would be an unnecessary waste of memory, it would be good if we could have just one copy of that data and let each instance object access it. We can do this by

defining variables at the class level; these variables are called class variables or class attributes. Let us define a class variable for the `Person` class that we had written earlier.

```
class Person:
    species = 'Homo sapien'
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(f'{self.name} is {self.age} years
old')
p1 = Person('John', 20)
p2 = Person('Jack', 34)
p1.display()
p2.display()
```

The variable named `species` is defined inside the class but outside any method, so it is a class variable. Class variables are generally placed at the top of the class definition, just below the class header. There is only a single copy of a class variable and it is shared by all the instances of the class. It belongs to the class, not to individual instance objects. The data of a class variable is stored in the class object itself, while the data of instance variables is stored in individual instance objects.

The value of `species` will be the same for all `Person` objects, there is no need to have a unique copy for each instance, so we have defined it as a class variable. Class variables are created in the class definition while the instance variables are created inside the methods, usually inside `__init__()`.

A class variable can be accessed using the class name or the instance name.

```
>>> Person.species
'Homo sapiens'
>>> p1.species
'Homo sapiens'
```

```
>>> p2.species  
'Homo sapiens'
```

We can see that the class variable is the same whether you access it with a class name or an instance name. Let us use the `id` function to verify that all these three references refer to the same variable stored in the class object.

```
>>> id(Person.species)  
2605159076592  
>>> id(p1.species)  
2605159076592  
>>> id(p2.species)  
2605159076592
```

We cannot access an instance variable like this with the class name, for example we cannot write `Person.name`.

```
>>> Person.name  
AttributeError: type object 'Person' has no  
attribute 'name'
```

This is because each `Person` instance object has a different name, there is no name attribute attached with the `Person` class itself, but you can write `Person.species` since `species` is the same for all `Person` instance objects.

In fact, the class variables can be accessed even before any instance object is created. Inside the class methods, you can access a class variable by preceding it with class name or `self`. Let us use this inside the `display` method.

```
def display(self):  
    print(f'{self.name} is {self.age} years old  
{Person.species}')
```

We could have written `self.species` but using the class name clearly shows that it is a class variable.

So, if there is any value that needs to be shared by all instances of a class, then there is no need to waste memory by storing it in each instance object, we can make it a class variable and only one copy will be stored in the class

object and all the instance objects can use the same copy. Class variables are created for storing data that does not vary for each instance while instance variables are created for data that can be different for each instance.

Let us see one more example. We have the following class named `BankAccount` that has instance variables for the representing the account number, owner name and balance.

```
class BankAccount:
    rate = 5
    min_balance = 1000
    min_balance_fees = 10
    def __init__(self, account_number, owner_name,
balance):
        self.account_number = account_number
        self.owner_name = owner_name
        self.balance = balance
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
account1 = BankAccount('7348', 'Tom', 50)
account2 = BankAccount('6378', 'Bob', 400)
```

The rate of interest would be the same for each instance of the account, so you can make it a class variable.

A bank can charge some fees if the balance becomes less than a minimum amount. So, you can make class variables for minimum balance and for minimum balance fees. The values of these variables can change but they will not vary for different accounts, which means that they will be the same for all the instances, so we have defined them at the class level. You can use these class variables in different methods that you define for this class. For example, you can check for minimum balance after a withdrawal in the `withdraw` method.

Class variables are often used to store class specific constants. For example, when you are creating a bounded data structure, you need a to specify a maximum limit for the size of the structure. It is better if we use a named constant instead of embedding a literal value in our code.

```
class Stack:
    MAX_LIMIT = 10
    def __init__(self):
        self.items = []
    def push(self, item):
        if len(self.items) >= Stack.MAX_LIMIT:
            raise Exception('Stack is full')
        self.items.append(item)
    def pop(self):
        if self.items == []:
            raise RuntimeError('Stack is empty')
        return self.items.pop()
    def display(self):
        print(self.items)
```

In this class, MAX_LIMIT is a class level constant, it is in all upper case as that is the convention for naming constants in Python.

We can use a class variable to count the number of instance objects created from a particular class. Let us add one more class variable in our Person class, this variable will store the number of Person instance objects created.

```
class Person:
    species = 'Homo sapien'
    count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.count += 1
```

```
def display(self):  
    print(f'{self.name} is {self.age} years old  
{Person.species}')
```

When the class definition executes, count variable is created and stored in the class object and it is initialized to zero. In the initializer, we have incremented count by 1. So, whenever a new instance object will be created the value of this variable count will be incremented.

```
>>> p1 = Person('Devanshi', 18)  
>>> p2 = Person('Devank', 10)  
>>> Person.count  
2
```

The value of class variable count is 2, since we have created two instance objects.

Class variables can be used to track data across all instances of a class. In the following program, we have a list as the class attribute and every time we create a new instance object, we add the account owner's name to the list.

```
class BankAccount:  
    account_holders = []  
    def __init__(self, account_number, owner_name,  
balance):  
        self.account_number = account_number  
        self.owner_name = owner_name  
        self.balance = balance
```

```
BankAccount.account_holders.append(self.owner_name)  
account1 = BankAccount('7348', 'Tom', 50)  
account2 = BankAccount('6378', 'Bob', 400)  
account3 = BankAccount('8348', 'Ron', 500)  
print(BankAccount.account_holders)
```

Output-

```
['Tom', 'Bob', 'Ron']
```


14.11 Class and object namespaces

We have learnt about namespaces; they are mapping from names to objects. In Python, classes and instance objects have their own distinct namespaces, generally implemented through dictionaries.

There is a namespace created for each class that is defined. When a class definition is executed, a new namespace is created for it. Anything defined at the top level of the class lives in this namespace, so all class variables and methods are part of this namespace. Basically, this namespace manages all names that are to be shared by all the instances of the class. When an instance object is created it gets its own namespace. Instance variables are part of this namespace. An instance gets access to all the names defined in the class namespace and the names defined in its own instance namespace.

These namespaces are represented by the `__dict__` attribute of the class or the instance. After executing the previous program, we can see the `__dict__` attribute of the class and the instance objects.

```
>>> account1.__dict__
{'account_number': '7348', 'owner_name': 'Tom',
'balance': 50}
>>> account2.__dict__
{'account_number': '6378', 'owner_name': 'Bob',
'balance': 400}
>>> BankAccount.__dict__
mappingproxy({'__module__': '__main__',
'account_holders': ['Tom', 'Bob', 'Ron'],
'__init__': <function BankAccount.__init__ at
0x000001A1B8A32340>, '__dict__': <attribute
'__dict__' of 'BankAccount' objects>,
'__weakref__': <attribute '__weakref__' of
'BankAccount' objects>, '__doc__': None})
```

When an attribute is accessed using an instance name, first the instance namespace is searched. If the attribute is found there then the value is returned, otherwise the attribute is searched in the class namespace. If found there, then the value is returned, otherwise `AttributeError` is raised. If

there is an attribute with same name in both instance namespace and class namespace, then the attribute in the instance namespace will be returned, because it is looked up before the class namespace.

In other words, if there is an instance variable that has the same name as the class variable, then the instance variable hides the class variable if you access the name through an instance.

In the following example, we have a class variable named `rate` and we have an instance variable which is also named `rate`.

```
class Account():
    rate = 5
    def __init__(self):
        self.rate = 10
    def display(self):
        print(Account.rate, self.rate)

a = Account()
a.display()
```

Output-

```
5 10
```

`Account.rate` gives us the value of class variable while `a.rate` gives us the value of instance variable. When we access a variable through an instance, Python first checks whether the instance contains that variable, if the instance does not contain that variable, then it checks the class to see if there is any class variable.

14.12 Changing a class variable through an instance

We have seen that we can access a class variable using either the class name or the instance name, however things are different when we change the value of a class variable. If you change the value of a class variable using the class name, it gets changed but if you try to change the value of a class variable by

using an instance, then something unexpected occurs. Let us understand this with the help of a simple example.

```
class Account():  
    rate = 5  
a1 = Account()  
a2 = Account()
```

In this class, we have a class variable named `rate` and we have created two instances of this class named `a1` and `a2`. As we know, we can access this variable `rate` using the class name or any of the two instances.

```
>>> Account.rate  
5  
>>> a1.rate  
5  
>>> a2.rate  
5
```

Let us change the value of this variable `rate` using the class name.

```
>>> Account.rate = 6  
>>> Account.rate  
6  
>>> a1.rate  
6  
>>> a2.rate  
6
```

The value of class variable was changed successfully. Now let us change the value of this class variable using the instance variable `a1`.

```
>>> a1.rate = 7  
>>> Account.rate  
6  
>>> a1.rate  
7
```

```
>>> a2.rate
6
```

We observe that only the expression `a1.rate` is showing the new value 7, while `Account.rate` and `a2.rate` are showing 6. The assignment `a1.rate = 7` did not change the class variable, it actually created a new instance variable named `rate` for the instance `a1` and the expression `a1.rate` accessed this instance variable. To verify this, let us check the ids and the `__dict__` attribute.

```
>>> id(Account.rate)
140713805079496
>>> id(a1.rate)
140713805079528
>>> id(a2.rate)
140713805079496
>>> a1.__dict__
{'rate': 7}
>>> a2.__dict__
{}
>>> Account.__dict__
mappingproxy({'__module__': '__main__', 'rate': 6,
 '__dict__': <attribute '__dict__' of 'Account'
 objects>, '__weakref__': <attribute '__weakref__'
 of 'Account' objects>, '__doc__': None})
```

This confirms that the instance variable `a1` got a new instance variable named `rate`.

In the previous section, we saw that if there is an instance variable and a class variable with same name, then the class variable gets hidden if we access that name using the instance. In our example, when we write `a1.rate`, first the instance namespace is searched and the variable is found there and so its value is returned. When we write `a2.rate`, the instance namespace is searched, this name `rate` is not found so the class namespace is searched, the name is found there and its value is returned. Only instance

object `a1` gets an instance variable named `rate`, other instances will continue to use the class variable whenever attribute `rate` is accessed through them.

So, if you want to change a class variable, you should do it through the class name, otherwise a new instance variable with the same name will be created for that particular instance object, and this instance variable will shadow the class variable.

Similar thing will happen if you change the value of a class variable inside a method using `self`. Let us see this with the help of the example that we have seen earlier. In the `Person` class, we had added a class variable to count the number of instance objects. In the `__init__` method, we incremented this variable by writing the statement `Person.count += 1`, if we change this to `self.count += 1` then the program will not work correctly.

```
class Person:
    species = 'Homo sapien'
    count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.count += 1
    def display(self):
        print(f'{self.name} is {self.age} years old
{Person.species}')
p1 = Person('Devanshi', 18)
p2 = Person('Devank', 10)
print(Person.count, p1.count, p2.count)
```

Output-

```
0 1 1
```

The statement `self.count += 1` does not change the class variable. We know that this statement is equivalent to `self.count = self.count + 1`. When this statement executes, the interpreter accesses the value of

class variable `count`, adds 1 to this value and then creates a new instance variable named `count` with the new value. This is why the value of class variable `count` always remains 0, and whenever a new instance object is created it gets a new instance variable named `count` with value 1.

When you assign to a class variable via the class, the attribute in the class namespace is changed. When you assign to a class variable via an instance, a new instance variable with the same name is created in the instance namespace.

If your class attribute is a mutable object, then it is possible to mutate it through the instance objects. Since all objects access the same class attribute, anyone of them can make in-place changes in the class attribute. This can give unexpected results if users are not aware of this.

14.13 Class Methods

In the `BankAccount` class that we have seen earlier, suppose we need to create a method that shows the class related details. We have three class variables in our class, so in our method we will print the values of these variables.

```
class BankAccount:
    rate = 5
    min_balance = 100
    min_balance_fees = 10
    def __init__(self, account_number, owner_name,
balance):
        self.account_number = account_number
        self.owner_name = owner_name
        self.balance = balance
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
```

```

def details(self):
    print(f'Rate : {BankAccount.rate}')
    print(f'Minimum Balance :
{BankAccount.min_balance}')
    print(f'Minimum Balance fees :
{BankAccount.min_balance_fees}')
account1 = BankAccount('7348', 'Tom', 50)
account1.details()

```

Output-

```

Rate : 5
Minimum Balance : 100
Minimum Balance fees : 10

```

The `details` method displays all the class variables and as usual we have called it with an instance. The `self` parameter was not used inside the method since it needed to access only the class variables. This method does not need to access any instance specific information. It would be better if we could call this method using the class name instead of any instance name. We can do so if we make this method a class method. A class method is a method that is associated with the class itself not with any particular instance of the class. To make this method a class method we need to precede the method definition with the line `@classmethod`.

```

@classmethod
def details(cls):
    print(f'Rate : {cls.rate}')
    print(f'Minimum Balance :
{cls.min_balance}')
    print(f'Minimum Balance fees :
{cls.min_balance_fees}')

```

The line `@classmethod` is a function decorator about which we will study later on, for now you can understand that adding this line turns a normal method into a class method. The other change that we can see in this method is that the parameter is now named `cls`. This is because when a class method is called, the interpreter automatically sends the class object and not

any instance object. The parameter is conventionally named `cls` because it is referring to the class object. You could write any other name here instead of `cls`, but just like `self`, this name `cls` is also a strong convention. This word is a short form of class; since `class` is a reserved word, this word `cls` is used.

Now, we can call this method with the class name. While calling, there is no need to provide any argument for the `cls` parameter, interpreter will automatically send the class as the argument for this parameter.

```
>>> BankAccount.details()
```

```
Rate : 5
```

```
Minimum Balance : 100
```

```
Minimum Balance fees : 10
```

We do not need any instance to call this method. We know that all the class variables are created even before any instance object is created, so we can call this method even if we do not have any instance of this class. When the call `BankAccount.details()` will execute, interpreter will automatically send the class as the first argument. So, the parameter `cls` refers to the class object inside the method definition, and that is why inside the method we have accessed the class variables through `cls` instead of hardcoding the class name.

A class method can also be invoked using an instance, but it makes more sense to invoke it using the class name. Class methods can work only with the class variables, they cannot access instance variables as they do not have a `self` parameter, and thus they have no access to the state of the instance.

So, if we have to implement a method that needs to use only the class variables, we can make that method a class method.

The normal methods that we have been defining till now, have `self` as the first parameter and when they are called, they automatically receive the current instance as the first argument. These methods are more precisely called instance methods, to distinguish them from the class methods and static methods. So, when we simply say methods of a class, we generally mean instance methods, because the other two are not as frequently used.

Now let us add a class method to the Person class that we have written earlier:

```
class Person:
    species = 'Homo sapien'
    count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.count += 1
    def display(self):
        print(f'{self.name} is {self.age} years old
{Person.species}')
    @classmethod
    def show_count(cls):
        print(f'There are {cls.count}
{cls.species}s')
Person.show_count()
p1 = Person('Devanshi', 18)
p2 = Person('Devank', 10)
Person.show_count()
```

Output-

There are 0 Homo sapiens

There are 2 Homo sapiens

Inside the class method `show_count` we have used two class variables `species` and `count`. We could call this method using any of the two instances.

```
>>> p1.count
2
>>> p2.count
2
```

We will get the same output, but calling with class name is more natural. So, when you have to process some information that is associated with the class itself not with any instance object, you can turn your method into a class method by writing the decorator `@classmethod` and specifying `cls` as the first parameter.

Class methods can be used to create alternative initializers in a class and to break static methods, we will see both of these approaches in detail.

14.14 Creating alternative initializers using class Methods

Class methods allow us to define alternative initializers (also known as factory methods) in a class. These methods help us create instance objects from different types of input data. Let us understand this with the help of an example. Again, we take the same `Person` class. We have deleted the class variables to keep it short and simple.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(f'{self.name} is {self.age} years
old')
p1 = Person('Devanshi', 18)
p2 = Person('Devank', 10)
```

We can initialize a new instance object of this `Person` class in only one way, by providing values of name and age. There may be situations when we want to create instance objects of type `Person` from different types of data. For example, we may have a string that contains name and age separated by a comma, or we may have a dictionary that contains name and age.

```
s = 'Jack, 23'
d = {'name': 'Jane', 'age': 34}
```

You might read this type of data from a file or from any other place. Now you want to be able to create an instance of type `Person` from these types of strings and dictionaries.

Python does not support function overloading, so there can be only one type of initializer. We cannot have more than one definition for `__init__` method inside a class. To initialize our objects in different ways, we can use class methods. In our `Person` class we will add two class methods named `from_str` and `from_dict`.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def from_str(cls, s):
        name, age = s.split(',')
        return cls(name, int(age))
    @classmethod
    def from_dict(cls, d):
        return cls(d['name'], d['age'])
    def display(self):
        print(f'{self.name} is {self.age} years
old')
s = 'Jack, 23'
d = {'name': 'Jane', 'age': 34}
p3 = Person.from_str(s)
p3.display()
p4 = Person.from_dict(d)
p4.display()
```

Output-

```
Jack is 23 years old
Jane is 34 years old
```

The method `from_str` takes a string as argument and creates and returns a `Person` object, and the method `from_dict` creates a `Person` object from a dictionary.

In the `from_str` method, `cls` as usual is the first parameter and the next parameter `s` is for accepting a string. We split the string `s` to get the values of `name` and `age`. After that we create a new instance object of type `Person` by using these values of `name` and `age`. We know that inside the class methods, the `cls` parameter refers to the class object. So, `cls` here refers to the `Person` class and writing `cls(name, int(age))` is equivalent to writing `Person(name, int(age))` and it will create a new `Person` instance object. It will call `__init__` to initialize the newly created object.

Similarly, our class method `from_dict` creates and returns a `Person` object from a dictionary with `'name'` and `'age'` as keys. We have sent the values of the dictionary as argument to the `Person` class initializer.

The methods `from_str` and `from_dict` are called with the class name. The instance objects that are returned by these methods are assigned to names `p3` and `p4`.

We can see that both the factory methods internally use the `__init__` method to create and return the instance objects. Instead of hardcoding the class name in these methods we have used the `cls` parameter to create the objects. This is good, if in future we have to rename the class or inherit a new class from this class. These factory methods would work for any class inherited from the `Person` class.

So, if we want to create factory methods that support inheritance, we should use class methods.

Instead of using these class methods, you might be tempted to change your `__init__` so that it works with different types of input data. You might think of using default arguments or variable number of arguments and then use checks inside the method to process data differently in each case. This approach can sometimes work, but it makes the code difficult to understand and maintain. The `if...elif...else` construct could be confusing if there are many cases to consider. The class methods approach is simpler to understand and also increases the readability of the calling code. The

`__init__` method is generally a simple method that initializes the instance variables from the arguments. The alternative initializers can do additional pre-processing of data to create the instance. Rather than cluttering our `__init__` method with all the code, we create separate initializers. Here is another example of a class method used to create an alternative initializer:

```
from datetime import datetime

class Employee:
    def __init__(self, first_name, last_name,
birth_year, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.birth_year = birth_year
        self.salary = salary
    def show(self):
        print(f'I am {self.first_name}
{self.last_name} born in {self.birth_year}')

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def from_employee(cls, emp):
        name = emp.first_name + ' ' + emp.last_name
        age = datetime.today().year -
emp.birth_year
        return cls(name, age)
    def display(self):
        print('I am', self.name, self.age, 'years
old')

e1 = Employee('James', 'Smith', 1990, 5000)
p1 = Person.from_employee(e1)
```

```
p1.display()
```

We want to create a Person object from an Employee object. For this, we will create a class method named `from_employee` in the Person class.

An Employee object has `first_name`, `last_name`, `birth_year` and `salary` as instance variables and in Person class we only need name and age. To get the name we will add the first name and last name. To get age we will subtract birth year from the current year. To get the current year we have to import `datetime` class from the `datetime` module. After getting the name and age of Employee, we create a new Person object and return it.

14.15 Static Methods

Sometimes we have to write methods that are related to the class but do not need any access to instance or class data for performing their work. These methods could be some helper or utility methods that are used inside the class but they can perform their task independently. There is no need to define these methods as instance methods or class methods as they do not need access to the instance object or the class object. We can define these methods as static methods by preceding them with the `@staticmethod` decorator. Unlike instance methods and class methods, static methods do not have any special first parameter. They can have regular parameters, but the first parameter has no special significance. So, when a static method is called, Python does not send the class object or the instance object as the first argument. This is why these methods cannot access or modify the instance state or the class state.

In the `BankAccount` class we saw earlier, we can add a static method named `about` that can be used to display general information about the class.

```
class BankAccount:
```

```
    .....  
    .....
```

```
    @staticmethod
```

```

def about():
    print('Information about BankAccount class
.....')

    print('.....')
    print('.....')
BankAccount.about()

```

A static method can be invoked using either the class name or an instance name.

In the following Date class, you can write a static method `is_leap` that can be used as helper method in other methods of the class.

```

class Date:
    def __init__(self, d, m, y):
        self.d = d
        self.m = m
        self.y = y
    def method1(self, year):
        .....
        if Date.isleap(year):
            .....
            .....
    def method2(self, days):
        .....
        if Date.isleap(self.y):
            .....
            .....
    @staticmethod
    def is_leap(year):
        if year % 4 == 0 and year % 100 != 0 or
year % 400 == 0:
            return True

```

```
else:
    return False
```

So, when you have to create a helper or utility method, that contains some logic related to the class, turn it into a static method. For example, if you are creating a `Fraction` class, you can create a static method for finding `hcf` of two numbers. This method can be used to reduce the fraction to lowest terms.

We have learnt about instance methods, class methods and static methods. If you have to make a method that needs to access instance variables, make it an instance method. An instance method has special first parameter named `self` that refers to the current instance object. If you have to make a method that needs to use only class variables and not instance variables, make it a class method. A class method has a special first parameter named `cls` that refers to the class object. When you need to create a general utility method, that needs to use neither instance variables nor class variables, make it a static method. Such a method depends only on its own argument values. It does not have any special first parameter.

A static method is just like a regular function, but it belongs to the class namespace. We know that the definition of a class defines a separate namespace and when you want to group functionalities under the class namespace, you can create static methods.

Static methods are like normal functions so instead of defining a static method, you could define a module level function that is defined near the class. If you have a single class per module or only closely related classes in a module, then you can make a module level function instead of writing a static method.

In the previous section, we saw that class methods could be used to create alternative initializers. Class methods can also be useful while splitting static methods. Suppose we have to write a static method that is very long and we decide to split it into several static methods. So, now, our static method will call other static methods. For this, we have to hardcode the class name, which can be a problem if we have inherited classes. We can avoid the hardcoding of the class name if we use a class method instead of a static method, because class method can use the parameter `cls` instead of the class name. Let us understand this with the help of an example:


```

class MyClass:
    @staticmethod
    def method1():
        print('method1 doing work')
        MyClass.method2()
        MyClass.method3()
    @staticmethod
    def method2():
        print('method2 doing work')
    @staticmethod
    def method3():
        print('method3 doing work')

```

Inside method1, we must hardcode the class name to call the other two static methods. We can avoid this if we make method1 a class method.

```

class MyClass:
    @classmethod
    def method1(cls):
        print('method1 doing work')
        cls.method2()
        cls.method3()
    @staticmethod
    def method2():
        print('method2 doing work')
    @staticmethod
    def method3():
        print('method3 doing work')

```

So, when you have a static method calling other static methods, convert it to a class method to avoid hardcoding the class name.

14.16 Creating Managed Attributes using properties

Properties can be used to create data attributes with special functionality. If you want some extra functionality (like type checking, data validation or transformation) while getting or setting a data attribute, you can define a property which creates a managed attribute. The user can access and modify this managed attribute with regular syntax (e.g. `print(MyClass.x)` or `MyClass.x = 3`), but behind the scene some method will be automatically executed while setting or getting the attribute. Property allows us to access data like a variable, but the accessing is handled internally by methods. This way, we can control attribute access by attaching custom behavior. Before seeing the syntax of creating a property, first, we will see with the help of a simple example why we need properties.

Suppose we have developed this class `Person`, with two instance variables `name` and `age`, and the method `display`.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.name, self.age)
if __name__ == '__main__':
    p = Person('Raj', 30)
    p.display()
```

Let us assume that this is a big class that is being used by many clients. After some time, we as the implementors of the class want to restrain the value of `age`. We want to ensure that whenever `age` is assigned a value, that value should be within the range 20 - 80.

A solution to this could be to make `age` a private variable and use getter and setter methods to access and update this private variable. Setters (also know

as mutators) and getters (also known as accessors) are generally used in object-oriented languages to restrict access to private variables and they allow you to control how these variables are accessed and updated.

We modify the class and make age a private variable by prefixing it with an underscore, so now client is not supposed to access it directly. We define a method `set_age` that will be used to assign a value to the private variable `_age`, and we define another method `get_age` that will be used to access the value of variable `_age`. In the `set_age` method we can put the validation code.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age
    def display(self):
        print(self.name, self._age)
    def set_age(self, new_age):
        if 20 <= new_age <= 80:
            self._age = new_age
        else:
            raise ValueError('Age must be between
20 and 80')
    def get_age(self):
        return self._age
if __name__ == '__main__':
    p = Person('Raj', 30)
    p.display()
```

Now, whenever the user wants to change the age, he will do it through the `set_age` method, and the data validation will be done.

```
>>> p.set_age(100)
ValueError: Age must be between 20 and 80
>>> p.set_age(12)
```

```
ValueError: Age must be between 20 and 80
```

```
>>> p.set_age(25)
```

```
>>> p.display()
```

```
Raj 25
```

So, by defining the setter and getter methods, we could successfully implement the new restriction on age.

Earlier when there was no restriction, and age was a public variable, if the user had to increase the current age by 1, he would simply write:

```
p.age +=1
```

Now in the modified class, we have setter and getter methods so to increase the value of age, user has to write this:

```
p.set_age(p.get_age() + 1)
```

These types of expressions are confusing and decrease readability. There is still a problem in our modified class. When the user creates a new object, he can send any value for the age because there is no data validation done in the initializer.

```
p1 = Person('Dev', 2000)
```

So, we need to perform the data validation in the initializer also by calling the set_age method.

```
def __init__(self, name, age):  
    self.name = name  
    self.set_age(age)
```

Now the data validation will be done at the time of creation of a new object also. It seems that we have solved the problem of restricting the value of age. Now users of our class will not be able to enter any value of age outside the range 20-80. But remember our Person class is being used by several clients, and there is lot of existing code that accesses age directly, for example `p.age = 30` or `print(p.age)`. The new changes in your class will break this client code and it will have to be rewritten with statements like `p.set_age(30)` and `print(p.get_age())`. You have changed the user interface and so your new update is not backward compatible. This refactoring can cause problems in your client code.

To avoid this problem, in other object-oriented languages, programmers would start their class design with private attributes along with getters and setters that do nothing except getting and setting the value of the private variable. These setters and getters do not perform any extra processing and they are not needed at the outset but they have to be added because they might be needed later, when you need some processing to be done while setting and getting an attribute. This design makes sure that if in future you have to add any data validation, then the existing client code will not break. The clients will already be accessing data through setters and getters, so you can change the implementation without changing the interface and breaking your client's code.

The getter and setter methods can also be used to make an attribute read only or write only. If you define only the getter method for a private variable and don't define the setter method for it then the variable becomes read only, users will be able to read that variable but cannot update it. As we have seen, setters and getters also allow data validation, i.e., the setter method can control what value can be assigned to the variable and getter method can change the way the variable is represented when it is accessed. In most other languages, getter and setter methods are common and they are used to protect and validate your private data.

This setter and getter methods approach is not preferred in Python, the Pythonic way of going about this whole thing would be to create a property. Properties allow us to write our class in a way that does not require the user of the class to call setter and getter methods.

The syntax of calling a property is same as the syntax for accessing a data attribute, although it is actually a method. The client code that uses a property does not look like a method call, instead it looks like a direct data attribute access. Let us see how we would use create a property for age in our Person class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.name, self.age)
```

This was our initial `Person` class in which we had to make changes to include data validation for age. Here is the modified class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, new_age):
        if 20 <= new_age <= 80:
            self._age = new_age
        else:
            raise ValueError('Age must be between
20 and 80')
    def display(self):
        print(self.name, self._age)
```

We have added two special methods, and both are named `age`. Before the header line of these methods, we have added a line starting with ‘@’ symbol. The line `@property` makes the first method a getter method, and the line `@age.setter` makes the second method a setter method.

Now after this modification, the name `age` has become a property, we can access it like we access an instance variable. There is no need to call it like a method by using parentheses. The actual value of age is stored in the private variable named `_age`. The `age` attribute is a property which provides an interface to this private variable. The name of the property should be different from the attribute where we store our data.

Whenever we reference the attribute named `age`, the method with the line `@property` will be executed and whenever we assign something to it, the method with the line `@age.setter` will be executed. The method with `@property` is the getter method and the method with `@age.setter` is

the setter method for the property. The setter method accepts an argument which is used for setting the property. Note that the name of both methods is the same; they are different because they are prefixed with different `@` lines. These lines are decorators, they decorate these methods. We have seen similar decorator syntax when we learnt about class methods and static methods. We will learn about the details of decorators later in a separate chapter. The getter method is always preceded with `@property` decorator and the setter method is preceded with the decorator that contains the property name followed by a dot and the word `setter`. If the name of your property is `salary` then the decorator for its setter would be `@salary.setter`.

The user of the class can now access `age` as if it were an instance variable.

```
>>> p = Person('Raj', 30)
>>> p.age + 1
31
>>> p.age = 40
>>> p.age = 200
ValueError: Age must be between 20 and 80
```

So, now we can easily access `age` as an instance variable and the data validation is also done. This is much more concise and cleaner than it was using the `set_age` and `get_age` methods approach. There is no need of calling the methods explicitly; whenever we access or update the attribute `age`, these methods will be automatically called behind the scenes. So, you can reference or assign to the property using the syntax of an instance variable, but under the covers, the method code is getting executed. By defining this property, we have added a new attribute that can be accessed like an instance variable.

In fact, if you put the parentheses, it will show error.

```
>>> p.age()
TypeError: 'int' object is not callable
```

Note that we have not changed the initializer, we have not written `self._age = age`. The statement in the initializer is `self.age = age`. Since `age` is a property now, we are setting the property `age` here and

so the setter method will be automatically called and the data validation will be done here also.

```
>>> p = Person('Raj', 300)
ValueError: Age must be between 20 and 80
```

The private variable `_age` is created in the setter method of the property. The initializer is indirectly calling this setter method to make sure that the data validation is done. If in the initializer, we write `self._age = age` then the data validation will not be done when a new object is initialized.

So, when you need to perform some data validation on an existing instance attribute, you can turn it into a property. The client can execute the property without using the parentheses after the property name, so the client gets a cleaner syntax, which is more like accessing a data attribute rather than a method call. The syntax is much better than the `set_age` and `get_age` approach, and the existing client code will continue to work smoothly even after these changes. No changes need to be done in the existing client code, so the changes made to your class are backward compatible.

All this makes sense only when you respect the convention of using an underscore for private attributes. The client code could use `_age` for referencing and assigning directly. Python does not enforce any strict restriction, so programmers are supposed to follow the conventions.

14.16.1 Creating read only attributes using properties

Another use of property is that you can make an attribute read-only or write-only. If you provide only the getter method, not the setter method, the property becomes a read only property. This way we can protect our private attribute from any sort of modification by the client, while still giving the access to read the value of the attribute.

```
class Employee:
    def __init__(self, name, password, salary):
        self._name = name
        self.password = password
        self.salary = salary
```



```

@property
def name(self):
    return self._name
@property
def password(self):
    raise AttributeError('password not
readable')
@password.setter
def password(self, new_password):
    self._password = new_password
@property
def salary(self):
    return self._salary
@salary.setter
def salary(self, new_salary):
    self._password = new_salary

```

In this class `Employee`, we have defined three properties, `name`, `password` and `salary`. For the `name` property, we have defined only the getter method, so this property becomes a read only property. The attribute `name` is read only. This attribute can only be set when the instance is created and it can only be changed within the class methods. It cannot be modified from outside the class.

The attribute `password` is write-only because in its getter method we have raised `AttributeError`. Note that it is necessary to provide the getter method, you cannot make an attribute write only by providing just the setter method.

If you provide both the setter and getter methods, the property becomes read/write property. For example, the attribute `salary` can both be referenced and assigned to, it is a read/write property.

```

>>> e = Employee('Jill', 'feb31', 5000)
>>> e.name

```

```
'Jill'
```

The name attribute is read-only, if we try to assign something to it, we cannot.

```
>>> e.name = 'Jack'
```

```
AttributeError: property 'name' of 'Employee'
object has no setter
```

The password attribute is not readable.

```
>>> e.password
```

```
AttributeError: password not readable. Did you
mean: '_password'?
```

```
>>> e.password = 'feb29'
```

The salary attribute is both readable and writable.

```
>>> e.salary = 6000
```

```
>>> e.salary
```

```
6000
```

14.16.2 Creating Computed attributes using properties

A common use of property is to create dynamically computed attributes, the values of these attributes are not actually stored, they are computed on request. Let us see an example of this:

```
class Rectangle():
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
        self.diagonal = (self.length * self.length
+ self.breadth * self.breadth) ** 0.5
    def area(self):
        return self.length * self.breadth
    def perimeter(self):
```

```
        return 2 * (self.length + self.breadth)
```

In this Rectangle class we have three instance variables, length, breadth and diagonal, and two methods area and perimeter. The value of instance variable diagonal is computed from the values of instance variables length and breadth.

```
>>> r = Rectangle(2, 5)
```

```
>>> r.diagonal
```

```
5.385164807134504
```

```
>>> r.area()
```

```
10
```

```
>>> r.perimeter()
```

```
14
```

Now let us change length:

```
>>> r.length = 10
```

```
>>> r.diagonal
```

```
5.385164807134504
```

We changed length, but value of diagonal has not changed.

```
>>> r.area()
```

```
50
```

```
>>> r.perimeter()
```

```
30
```

Area and perimeter have changed because they are implemented as methods.

So, if you change the value of an instance variable, any other instance variable that is computed from it will not automatically update. Here in this class if we change length or breadth, then diagonal will not change accordingly. One solution could be to implement diagonal as a method. But then we will not be able to access it as an instance variable; whenever we want to access it, we have to put parentheses. This will also break any client code that has used diagonal as an instance variable. The solution is to turn it into a property.

```
@property
def diagonal(self):
    return (self.length * self.length +
self.breadth * self.breadth) ** 0.5
```

Now we can continue to access diagonal as an instance variable; whenever we will access diagonal, its value will be calculated and we will get the updated value. So, changes in length and breadth will be reflected in the diagonal.

```
>>> r = Rectangle(2, 5)
>>> r.diagonal
5.385164807134504
>>> r.length = 10
>>> r.diagonal
11.180339887498949
```

There is no need to define the setter method, because we do not expect the user to change the diagonal.

14.16.3 Deleter method of property

We can also define a deleter method for the property, this deleter method defines what happens when a property is deleted. To create the deleter method, you have to define a method with the same name as the property and add the decorator with the word `deleter` in it.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, new_age):
```

```

        if 20 <= new_age <= 80:
            self._age = new_age
        else:
            raise ValueError('Age must be between
20 and 80')
    @age.deleter
    def age(self):
        del self._age
        print('age deleted')
    def display(self):
        print(self.name, self._age)

```

The deleter method will be executed, when the attribute is deleted.

```

>>> p = Person('Jill', 25)
>>> print(p.age)
25
>>> del p.age
age deleted

```

Let us summarise what we have learnt about properties.

A property allows access to an instance variable through methods, even though the method syntax is not used. By using the property syntax, we can define methods that are automatically called when an instance variable is referenced, assigned or deleted. We can define three methods for a property:

Getter - executed when the attribute is accessed, preceded with decorator `@property`

Setter - executed when the value of attribute is set, preceded with decorator `@name.setter`

Deleter - Executed when the attribute is deleted, preceded with decorator `@name.deleter`

All three methods have same name which is the name of the property, they are distinguished because of the decorators. All of them take `self` as the

first argument and the setter method takes an additional argument for setting the value of the property. If you want to provide a docstring for the property, specify it in the getter method.

Properties can be used for attribute type checking and validation, for creating read-only or write-only attributes and for creating computed attributes. You can incorporate new behaviour in your instance variables, without any need to rewrite the existing client code. Thus, you can use a property to give new functionality to existing instance variables.

In the property getter and setter methods, do not perform actions that are surprising or take much time. Referencing or assigning to an attribute is something that the client will expect to run instantly so it is not advisable to run other helper methods in the property methods. If a task is very complex and time consuming and may have side effects, consider putting it in a separate normal method.

14.17 Designing a class

After learning about data hiding and properties, let us see how to decide which attributes should be private, public or turned into a property. If there is an attribute that should never be accessed by the user, it should be made private by prefixing it with an underscore. There is no need of defining any property for it, as it should not be accessed from outside the class. These internal attributes are part of the implementation and should not be exposed in the public interface in any way.

Then there are attributes that have to be accessed or modified by the user. You can start implementing your class by coding such instance variables as public and in future if you need more control over any instance variable, you can change it to a private attribute and write a property to access it. You should define a property only if it provides some extra functionality to the attribute. There is no point in defining a property that just gets and sets data without any extra logic. In other languages that do not have property mechanism, we need such setter and getters, but in Python we can always start with plain public attributes and promote them to properties whenever required without changing the interface. Public attributes that need no extra

functionality while being accessed or modified should remain plain public attributes in the class.

So, in Python, you can start with a very simple design and later introduce properties without changing the interface. There is no need to pollute your space with multiple setters and getters just to ensure that future changes are backward compatible.

Exercise

1. What will be the output for the following program?

```
class Test:
    pass

t1 = Test()
t2 = Test()

print(t1 == t2, end=' ')
print(type(t1) == type(t2), end=' ')
```

2. The class object is created when _____
(A) the class definition is executed (B) the class is instantiated
3. In a method definition, the parameter `self` refers to the:
(A) class object (B) instance object that invokes the method
4. If you create a method that doesn't need any arguments, you don't have to specify any parameters in the definition.
(A) True (B) False
5. Instantiation of the class creates a new _____ object.
(A) class (B) instance
6. While calling a method, it is optional to provide an argument for parameter `self`. If you don't provide an argument for `self`, it will be provided by the interpreter.
(A) True (B) False

7. Is there anything wrong in the following code?

```
class Test:
    def method1(self):
        print('Inside method1')
    def method2(self):
        print('Inside method2')
        method1()

t = Test()
t.method2()
```

8. class Test:

```
    def method1(self):
        x = 12
    def method2(self):
        self.y = 10
```

(A) x and y are instance variables (B) x is instance variable, y is local variable

(C) y is instance variable, x is a local variable

What will be the output for Questions 9 and 10.

9. class Test:

```
    def method1(self):
        self.x = 10
    def display(self):
        print(self.x)
```

```
t = Test()
t.display()
```

10. class Test:


```

def method1(self, x):
    self.x = x
def method2(self):
    x += 10
def display(self):
    print(self.x)

t = Test()
t.method1(5)
t.method2()
t.display()

```

11. _____ variables are different for each instance, _____ variables are shared by all instances of the class.
(A) Instance, class (B) Class, instance
12. Variables that are assigned a value inside _____ are class variables, and variables that are assigned values inside _____ are instance variables.
(A) class methods, class definition (B) class definition, class methods
13. If you call a class method using an instance argument, it receives the instance as the first argument.
(A) True (B) False
14. Conventionally, the first parameter of an instance method is named _____ and the first parameter of a class method is named _____.
(A) this cls
(B) self cls
(C) self class
15. The _____ decorator changes the method such that it receives the class as the first argument and _____ decorator changes the method such that it receives no special first argument.

(A) @classmethod @instancemethod

(B) @classmethod @staticmethod

(C) @staticmethod @classmethod

16. Will this code show error?

```
class Test:
    x = 7

    @classmethod
    def method1(self):
        print(self.x)

Test.method1()
```

17. Can you write an instance variable preceded with the class name, for example `MyClass.x` where `MyClass` is the name of the class and `x` is an instance variable?

18. Can you write a class variable preceded with an instance name, for example `p1.x` where `p1` is the name of an instance and `x` is a class variable?

19. Make a class that represents a bank account, name it `BankAccount`. Create four methods named `set_details`, `display`, `withdraw` and `deposit`.

In the `set_details` method, create two instance variables: `name` and `balance`. The default value for `balance` should be zero. In the `display` method, display the values of these two instance variables.

The methods `withdraw` and `deposit` should have a parameter named `amount`. Inside the method `withdraw`, subtract the `amount` from `balance` and inside the `deposit` method, add `amount` to `balance`.

Create two instances of this class and call the methods on those instances.

20. In the `BankAccount` class that you created in the previous exercise, delete the `set_details()` method and create a `__init__`

method.

21. Create a class named `Book` with a `__init__` method. Inside the `__init__` method, create the instance variables `isbn`, `title`, `author`, `publisher`, `pages`, `price`, `copies`.

Create these four instance objects from this class.

```
book1 = Book('957-4-36-547417-1', 'Learn  
Physics', 'Stephen', 'CBC', 350, 200, 10)
```

```
book2 = Book('652-6-86-748413-3', 'Learn  
Chemistry', 'Jack', 'CBC', 400, 220, 20)
```

```
book3 = Book('957-7-39-347216-2', 'Learn  
Maths', 'John', 'XYZ', 500, 300, 5)
```

```
book4 = Book('957-7-39-347216-2', 'Learn  
Biology', 'Jack', 'XYZ', 400, 200, 6)
```

Write a method named `display` that prints the ISBN, title, price and number of copies of the book.

22. For the `Book` class that you have created, write a method named `in_stock` that returns `True` if the number of copies is more than zero. Otherwise, it returns `False`.

Create another method named `sell` that decreases the number of copies by 1 if the book is in stock. Otherwise, it prints the message that the book is out of stock.

23. Create a list named `books` that contains the 4 `Book` instance objects that you have created in *Question 21*. Iterate over this list using a for loop and call the `display()` for each object in the list.

Write a list comprehension to create another list that contains the titles of books written by an author named Jack.

24. In the `Book` class, create a property named `price` such that the price of a book cannot be less than 10 or more than 500.
25. Make a class `Fraction` that contains two instance variables, `nr`, and `dr` (`nr` stands for numerator and `dr` for denominator). Define a

`__init__` method that provides values for these instance variables. Make the denominator optional by providing a default argument of 1.

In the `__init__` method, make the denominator positive if it is negative. For example, $-2/-3$ should be changed to $2/3$ and $2/-3$ to $-2/3$.

Write a method named `show` that prints numerator, then `'/'` and then the denominator.

Make sure that you write this class as we will be using it to learn magic methods in the next chapter.

26. In the `Fraction` class created in the previous question, define a method named `multiply` that multiplies two `Fraction` instance objects. For multiplying two fractions, you have to multiply the numerator with numerator and denominator with denominator.

Inside the method, create a new instance object that is the product of the two fractions and return it. Write your method in such a way that it supports multiplication of a `Fraction` by an integer also.

Similarly define a method named `add` to add two `Fraction` instance objects. Sum of two fractions $n1/d1$ and $n2/d2$ is $(n1*d2 + n2*d1) / (d1*d2)$. This method should also support addition of a `Fraction` by an integer.

Test your fraction class with this code:

```
f1 = Fraction(2, 3)
f1.show()
f2 = Fraction(3, 4)
f2.show()
f3 = f1.multiply(f2)
f3.show()
f3 = f1.add(f2)
f3.show()
f3 = f1.add(5)
```

```
f3.show()
f3 = f1.multiply(5)
f3.show()
```

The output that you should get is given below:

```
2/3
3/4
6/12
17/12
17/3
10/3
```

27. For the following class `Product`, create a read only property named `selling_price` that is calculated by deducting discount from the `marked_price`. The instance variable `discount` represents discount in percent.

```
class Product():
    def __init__(self, id, marked_price,
discount):
        self.id = id
        self.marked_price = marked_price
        self.discount = discount
    def display(self):
        print(self.id, self.marked_price,
self.discount)
p1 = Product('X879', 400, 6)
p2 = Product('A234', 100, 5)
p3 = Product('B987', 990, 4)
p4 = Product('H456', 800, 6)
```

28. Suppose after some time, you want to give an additional 2% discount on a product, if its price is above 500. To incorporate this change, implement `discount` as a property in your `Product` class created in the previous question.
29. Write a `Circle` class with an instance variable `radius` and a method named `area`. Create two more attributes named `diameter` and `circumference` and make them behave as read only attributes.

Perform data validation on `radius`, user should not be allowed to assign a negative value to it.

For a circle:

`diameter = 2 * radius`

`circumference = 2 * 3.14 * radius`

`area = 3.14 * radius * radius`

30. The following function finds the highest common factor of two numbers:

```
def hcf(x, y):  
    x = abs(x)  
    y = abs(y)  
    smaller = y if x > y else x  
    s = smaller  
    while s > 0:  
        if x % s == 0 and y % s == 0:  
            break  
        s -= 1  
    return s
```

Make it a static method in the `Fraction` class that you had written in *Question 26*.

31. In your `Fraction` class of the previous question, write a private instance method `_reduce` that reduces a fraction to its lowest terms. To reduce a `Fraction` to its lowest terms you have to divide the numerator and denominator by the highest common factor. Call the static method `hcf` in `__init__` and also call it on the resultant fraction in methods `multiply` and `add`.
32. In the following class named `SalesPerson`, add two class variables named `total_revenue` and `names`. The variable `names` should be a list that contains names of all salespersons and `total_revenue` should contain the total sales amount of all the salespersons.

```
class SalesPerson:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.sales_amount = 0
    def make_sale(self, money):
        self.sales_amount += money
    def show(self):
        print(self.name, self.age,
self.sales_amount)
s1 = SalesPerson('Bob', 25)
s2 = SalesPerson('Ted', 22)
s3 = SalesPerson('Jack', 27)
s1.make_sale(1000)
s1.make_sale(1200)
s2.make_sale(5000)
s3.make_sale(3000)
```

```
s3.make_sale(8000)
s1.show()
s2.show()
s3.show()
```

33. Add a class variable named `domains` to the following `Employee` class. This class variable should be of set type and it should store all domain names used by the employees.

```
class Employee:
    def __init__(self, name, email):
        self.name = name
        self.email = email
    def display(self):
        print(self.name, self.email)
e1 = Employee('John', 'john@gmail.com')
e2 = Employee('Jack', 'jack@yahoo.com')
e3 = Employee('Jill', 'jill@outlook.com')
e4 = Employee('Ted', 'ted@yahoo.com')
e5 = Employee('Tim', 'tim@gmail.com')
e6 = Employee('Mike', 'mike@yahoo.com')
```

34. In the following `Employee` class, add a class variable named `allowed_domains`.

```
allowed_domains = {'yahoo.com', 'gmail.com',
'outlook.com'}
```

Whenever an email is assigned, if the domain name is not in `allowed_domains`, raise a `RuntimeError`.

```
class Employee:
    def __init__(self, name, email):
```



```

        self.name = name
        self.email = email
    def display(self):
        print(self.name, self.email)
e1 = Employee('John', 'john@gmail.com')
e2 = Employee('Jack', 'jack@yahoo.com')
e3 = Employee('Jill', 'jill@outlook.com')
e4 = Employee('Ted', 'ted@yahoo.com')
e5 = Employee('Tim', 'tim@xmail.com')

```

35. The following program shows implementation of Stack Abstract data type using list. In a stack, elements are pushed and popped from one end of the stack which is called the top of the stack.

This implementation has no maximum limit on the size of the stack. You have to introduce a maximum limit by adding a class variable named `MAX_SIZE`. In the `push` method, before inserting a new element, check the size of the stack and raise a `RuntimeError` if the stack is full.

```

class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def size(self):
        return len(self.items)
    def push(self, item):
        self.items.append(item)
    def pop(self):

```

```

        if self.is_empty():
            raise RuntimeError("Stack is
empty")

        return self.items.pop()

    def display(self):
        print(self.items)

if __name__ == '__main__':
    st = Stack()
    while True:
        print('1.Push')
        print('2.Pop')
        print('3.Size')
        print('4.Display')
        print('5.Quit')
        choice = int(input('Enter your choice
: '))

        if choice == 1:
            x = int(input('Enter the element
to be pushed : '))
            st.push(x)
        elif choice == 2:
            x = st.pop()
            print('Popped element is : ', x)
        elif choice == 3:
            print('Size of stack ', st.size())
        elif choice == 4:

```

```

        st.display()
    elif choice == 5:
        break
    else:
        print('Wrong choice')
print()

```

36. Class variables with immutable values can be used as defaults for instance variables. In the following BankAccount class, add an instance variable named bank in the `__init__` method. Add a class variable `bank_name` that will be used as default argument in the `__init__` method for bank parameter.

```

class BankAccount:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance
    def display(self):
        print(self.name, self.balance)
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
a1 = BankAccount('Mike', 200)
a2 = BankAccount('Tom')
a1.display()
a2.display()

```

Project : Quiz creation

In this project we will implement a quiz. Making a program to implement a simple quiz is not that difficult but we will make this project interesting and challenging by implementing a quiz that is quite flexible. First let us see what are the features that we want our quiz to have.

The quiz should support multiple topics; user should be able to select a topic in which he wants to take the quiz. The number of questions in the quiz is not fixed. Whenever user takes the quiz, he should be able to select the number of questions that he wants to attempt. Here is a screenshot of the initial screen of the quiz:

```
Enter your name : John
Welcome John
You can take the quiz in any one of these 5 topics
1.Maths
2.Geography
3.Python Language
4.Chemistry
5.Data Structures
Choose your topic (1-5): 1
There are 12 questions available in Maths
How many questions do you want to attempt : 6
```

Figure 14.5: Initial screen of the quiz

The questions should not be presented to the user in some fixed order, they should appear randomly. The quiz should contain multiple choice questions and the number of choices need not be fixed. For each question, 2 or more choices can be shown to the user. For example, in the following screenshot, 3 choices are shown for the first question and 2 choices for the second question:

```

What is the area of circle
1. pi * r * r
2. pi * r
3. pi * pi * r
Enter your option : 1
Your answer is correct

What is the value of 2+3*4
1. 20
2. 14
Enter your choice : 1
Sorry, wrong answer. Option 2 is the correct answer

```

Figure 14.6: Displaying questions and getting answers

2 points are awarded for each correct answer and 1 point is deducted for each wrong answer. After taking the quiz, user is shown the result of the quiz and is also shown his previous scores in that topic. After this, the user is asked if he wants to take the quiz again, if he enters y then again you have to show him all the available topics and start the quiz all over again.

```

..... Quiz over .....
You gave 6 correct answers and 2 wrong answers
Your score is 10/16
.....
Press Enter to view the history of your scores in Maths
You scored 9/12 on Sun Oct 1 16:35:44 2023
You scored 12/12 on Thu Oct 5 17:43:39 2023
You scored 8/14 on Sun Oct 8 16:46:25 2023
You scored 10/16 on Mon Oct 9 14:52:45 2023
Want to take the quiz again(y/n) :

```

Figure 14.7: Text displayed when the quiz is over

So now you know what are the requirements and how your application should work. You can come up with your own design and start coding based on that design. It is up to you to design it using only functions or by using object-oriented approach. Here is an object-oriented approach to design it:

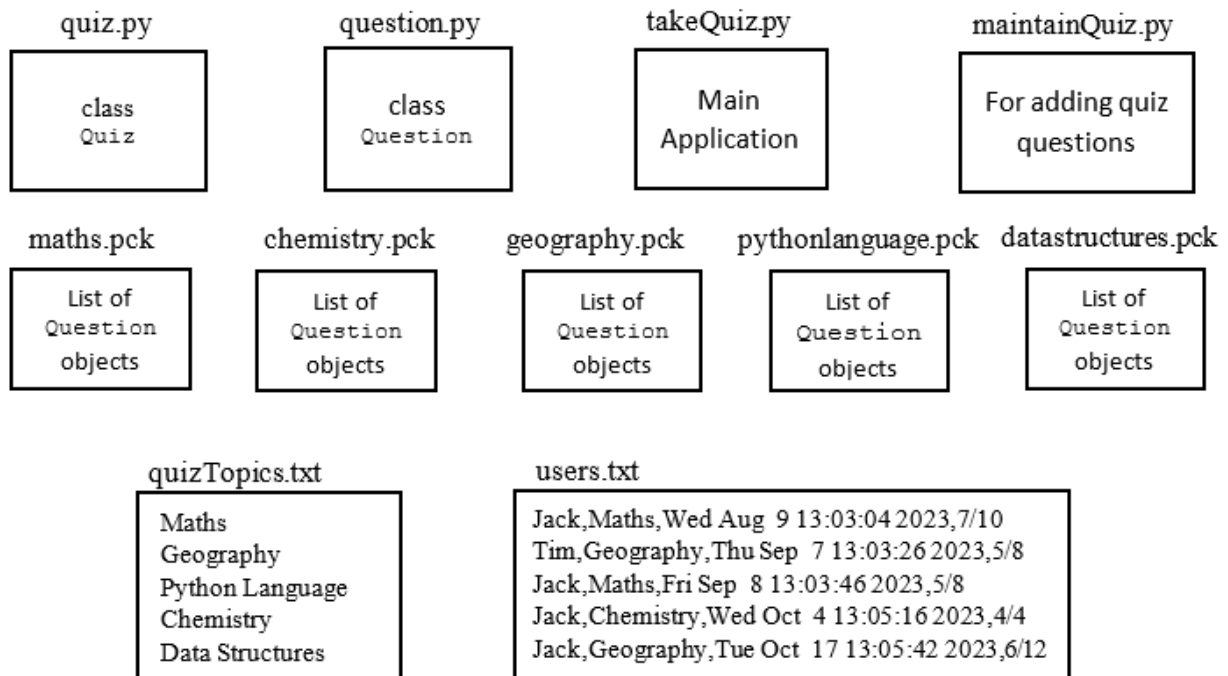


Figure 14.8: Files used in the program

In the module `quiz.py` we will define a class named `Quiz`, and in the module `question.py`, we will define a class named `Question`. The module `takeQuiz.py` will be the main module. It is the application that the user will run when he wants to take a quiz. The module `maintainQuiz.py` is for the creator of the quiz, it will be executed whenever the creator wants to add new quiz questions. The information of questions will be stored in pickled files, for each topic there will be a separate file. Each of these pickled files contains a list of `Question` objects.

The text file `quizTopics.txt` contains the names of the quiz topics. Right now, we have 5 topics in our quiz application. The creator of the quiz can add more topics, and for each newly added topic, a new pickled file will be created.

At the end of the quiz, we are showing the user his previous scores, so we need to save the results of each quiz in some file. We will store the information of all the previous scores of the users in the file `users.txt`. Each line in this file is a comma-separated list of items, where the first item is the name of the user, the second item is the topic, then there is the date and time when the quiz was taken, and then the score of the user.

We have to write the code in the 4 modules. Let us start by writing the Question class in the `question.py` module:

```
class Question:
    def __init__(self):
        self.text = ''
        self.options = []
        self.answer = 0
```

Each question will have some text that will be shown to the user when the question is asked. We create an instance variable `text` and initialize it to an empty string. We need multiple choice questions, so we will keep all the answer options in a list. The instance variable `answer` will store the option number of the correct answer. So, suppose there are 4 answer options given in the `options` list and the option 2 is the correct answer then value of the instance variable `answer` will be 2.

Now we will write a method `enter_details` in which we will enter the values for these instance variables.

```
def enter_details(self):
    self.text = input('Enter the text of the
question : ')
    n = int(input('How many options do you want to
give for the answer : '))
    for i in range(n):
        option = input(f'Enter option {i+1} : ')
        self.options.append(option)
    self.answer = int(input('Enter the option
number of the correct answer : '))
```

First, we get the text of the question. A question can have any number of options for the answer so next we ask for the number of options. We store this in variable `n` and then write a loop that iterates `n` times. In this loop, we will get the options entered and will append those options to the `options` list. Next, we need to store the correct answer, so after the loop we get the option number of the correct answer. All three instance variables will get the values after this method `enter_details()` is called.

The creator of the quiz will create `Question` instance objects in the `maintainQuiz.py` file and will call this method `enter_details` to fill in all the details of the question.

Now, we will create a method named `ask`, this method will be called when the question has to be presented to the user who is taking the quiz.

```
def ask(self):
    print(self.text)
    for i in range(len(self.options)):
        print(f'{i+1}. {self.options[i]}')
    response = int(input('Enter your option : '))
    return self._check(response)
```

The text of the question and all the options for the answer are shown to the user. After that we ask the user to enter an option. To check the user's response we will call another method named `_check`.

```
def _check(self, response):
    if response == self.answer:
        print('Your answer is correct\n')
        return True
    else:
        print('Sorry, wrong answer.', end = ' ')
        print(f'Option {self.answer} is the correct\n')
        return False
```

If `response` is equal to `self.answer` we will print a message and return `True` otherwise we will print another message, show the correct answer and return `False`. The return value of the method `_check` is also returned from the method `ask`.

Now, let us write some test code to see if the code that we have written is working properly. We will write this code inside `if __name__ == '__main__':` because we do not want this code to be executed when the module is imported; we want it to be executed only when the module is run as the main script.


```
if __name__ == '__main__':  
    question = Question()  
    question.enter_details()  
    question.ask()
```

We have created a `Question` instance object and then called the method `enter_details` to fill in all the details and then we have called the `ask` method. Here is a sample run of the test code:

```
Enter the text of the question : Which is the smallest planet  
How many options do you want to give for the answer : 3  
Enter option 1 : Mars  
Enter option 2 : Mercury  
Enter option 3 : Venus  
Enter the option number of the correct answer : 2  
Which is the smallest planet  
1. Mars  
2. Mercury  
3. Venus  
Enter your option : 1  
Sorry, wrong answer. Option 2 is the correct answer
```

Figure 14.9: Test code to test `Question` class

So now we have a working `Question` class. In the test code, we have created a `Question` instance object and called the two methods on it. The `Question` object was not saved anywhere for future use. The file `maintainQuiz.py` that we will write next will be executed to store the list of `Question` objects in different pickled files.

Next, create a file `quizTopics.txt` in your current folder and enter some topics in it on which you want to create the quiz. Each topic should be added on a separate line.

Now, we will write code in the file `maintainQuiz.py`. This module will be executed by the creator of the quiz to create a new quiz file or to add questions to an existing quiz file. We need to import the `pickle` module and the `Question` class.

```
import pickle  
from question import Question
```

```

while True:
    print('1. Create a new quiz file')
    print('2. Add questions to an existing quiz
file')
    print('3. Exit')

    choice = input('Enter your choice : ')
    if choice != '1' and choice != '2':
        break

    with open('quizTopics.txt', 'r') as file:
        topics = [topic.strip() for topic in
file.readlines()]

    # rest of the code

```

We write an infinite loop, inside this loop first we show the options. Option 1 is for creating a new quiz file and option 2 is for adding questions to an existing file. After this we ask the user his choice of option, if choice is not 1 or 2 then we will break out of this infinite loop.

After this we read all the topics from the file `quizTopics.txt` and create a list of all those topics. The newline is stripped from each topic name with the help of `strip` method.

Now, we will write an `if-else` statement to execute different pieces of code depending on whether the choice is 1 or 2.

```

if choice == '1':
    topic = input('Enter the new topic : ')
    if topic in topics:
        print('This topic is already present')
        print('Choose option2 to add questions
to the existing file')

```

```

        continue
    with open('quizTopics.txt', 'a') as file:
        file.write(f'{topic}\n')
    questions_list = []
else: # choice will be 2
    print(f'\nAvailable topics : {topics}')
    topic = input('Enter the topic in which you
want to add questions : ')
    if topic not in topics:
        print('This topic not included still)
        print('Choose option 1 to create a new
quiz file')
        continue
    filename = topic.lower().replace(' ', '') +
'.pck'
    with open(filename, 'rb') as file:
        questions_list = pickle.load(file)
    print(f'This topic has
{len(questions_list)} questions\n')
    for question in questions_list:
        print(f' - {question.text}\n')

```

If the choice is 1, which means that the user wants to create a new quiz file, we ask the user to enter a new topic. If this topic is already present in the topics list, then we will print a message and continue. The continue statement will take the control to the start of the while loop where the user will be asked to enter his choice again.

If the topic is not present in topics list, then we will open the quizTopics.txt file in append mode and add the new topic to it.

After this, we create an empty list named `questions_list`. Later, we will add `Question` objects to it and write to the file.

The control will come to the else part only when the choice is 2. In this case, the user wants to add questions to an existing file. We show all the available topics to the user and then ask the user to enter the topic name to which he wants to add the questions. If the topic name that the user enters is not in the topics list, then we print a message. The `continue` statement will take the control to the start of an infinite `while` loop.

If the topic is available, then first we will get the filename from the topic name. The names of the pickled files are generated from the topic names. First, the topic name is converted to lowercase, then the spaces are deleted from it, and then `.pck` is added at the end. The resultant string is the file name in which questions related to this topic are stored. For example, if the topic name is 'Data Structures', then the filename will be `datastructures.pck`. We will open that file in read mode and get the `questions_list` from the file.

So, in the case of choice 1, `questions_list` is an empty list, and in this case, `questions_list` is the list that we read from the file. After reading the list from the file, we will show the user all the existing questions. For this, we will display the `text` attribute of each `Question` object present in `questions_list`.

Next, we will append questions to the `questions_list`.

```
while True:
    question = Question()
    question.enter_details()
    questions_list.append(question)
    response = input('Want to enter another
question(y/n) :')
    if response == 'n':
        break
```

Inside this infinite loop, we create a `Question` object and enter all the details by calling the method `enter_details` and then append that `Question` object to the list. This loop keeps on executing till the user enters 'n' in response to the question.

After the user has entered all the questions, we will write `questions_list` to the file.

```
filename = topic.lower().replace(' ', '') +
'.pck'
with open(filename, 'wb') as file:
    pickle.dump(questions_list, file)
```

We get the filename from the topic and then open the file in write mode, so if the choice is 1 then a new file will be created and if the choice is 2 then the list will overwrite the existing list that is there in the file.

We can execute this program to enter questions in different topic files:

```
1. Create a new quiz file
2. Add questions to an existing quiz file
3. Exit
Enter your choice : 1
Enter the new topic : Physics
Enter the text of the question : Which material is the best conductor of electricity
How many options do you want to give for the answer : 3
Enter option 1 : wood
Enter option 2 : metal
Enter option 3: plastic
Enter the option number of the correct answer : 2
Want to enter another question(y/n) :y
Enter the text of the question : Which of the following is not a type of simple machine
How many options do you want to give for the answer : 2
Enter option 1 : Pulley
Enter option 2 : Battery
Enter the option number of the correct answer : 2
Want to enter another question :n
```

Figure 14.10: *Entering questions in different topic files*

Similarly, we can add questions to an existing topic.

Now let us write our `Quiz` class in the file `quiz.py`. We will need to import the `Question` class that we have written, and we will also need `pickle` and `random` modules.

```
from question import Question
```

```

import pickle
import random
class Quiz:
    points_correct_answer = 2
    points_wrong_answer = -1
    def __init__(self, topic):
        self.topic = topic
        self.filename = topic.lower().replace('
', '') + '.pck'
        self.number_of_questions = 0
        self.correct_answers = 0
        self.wrong_answers = 0
        self.score = 0
        self.max_score = 0

```

Let us look at the instance variables. First, we have the topic of the quiz and then we have the filename. This is the name of the file where the questions of this topic are stored. As we have seen, the name of the file is generated from the topic name itself. First, the topic name is converted to lowercase and then the spaces are deleted from it and then .pck is added at the end. The resultant string is the name of the file in which questions related to this topic are stored.

Next, we have the number of questions that the user wants to attempt, number of correct answers, number of wrong answers, score and maximum score.

We have defined two class variables `points_correct_answer` and `points_wrong_answer`. These are not instance specific values, they will be same for each `Quiz` object, so we have made them class variables.

Now we will write a method named `start`, which when called will start a quiz. For starting the quiz, first of all we have to read the file in which we have stored the questions. We know that we have separate files for each topic, and each file contains a pickled list of `Question` objects. So, we will

open the related file in read binary mode and will get a list of all the question objects by calling the load method from the pickle module.

```
def start(self):  
    with open(self.filename, 'rb') as file :  
        questions_list = pickle.load(file)
```

Now, we want to show the user how many questions are available in the topic of the quiz, so we will print the length of questions_list.

```
print(f'There are {len(questions_list)} questions  
available in {self.topic}')
```

Next, we will ask the user how many questions he wants to attempt. We want to ensure that the user enters the number of questions less than or equal to the number of questions in the list. So, we will write a loop to validate the input.

```
while True:  
    self.number_of_questions = int(input('How many  
questions do you want to attempt : '))  
    if self.number_of_questions <=  
len(questions_list):  
        break  
    print(f'Number of questions should be <=  
{len(questions_list)}')
```

This loop will keep on executing till the user enters a number that is less than or equal to the number of questions in the list.

Now we know the number of questions, so the maximum score for the quiz will be will be equal to number of questions multiplied by points for correct answer.

```
self.max_score = self.number_of_questions *  
Quiz.points_correct_answer
```

Suppose the number of questions is 10 and 2 marks are awarded for each correct answer then the maximum score is 20. Thus, whatever score the user gets, will be out of 20.

Now we will shuffle the list of questions that we have read from the file, we need to do this because we want the questions to appear in random order.

```
random.shuffle(questions_list)
```

Now we have this for loop.

```
for i in range(self.number_of_questions):
    question = questions_list[i]
    a = question.ask()
    if a:
        self.correct_answers += 1
    else:
        self.wrong_answers += 1
```

Suppose the number of questions that the user wants to attempt is 10, then this loop will iterate 10 times and will show the first 10 questions of the shuffled list. Each item of `questions_list` is a `Question` object, so we can call the `ask` method on that `Question` object. If the method returns `True`, it means that the answer was correct so we increment the instance variable `correct_answers` by 1, otherwise we increment `wrong_answers` by 1. We need to import the `Question` class, because we are using these `Question` objects here.

At the end of the method `start` we will print this message.

```
print('..... Quiz over
.....')
```

Now we will define a method `get_result` that will show and return the result of the quiz.

```
def get_result(self):
    print(f'You gave {self.correct_answers} correct
answers', end=' ')
    print(f' and {self.wrong_answers} wrong
answers')
    self.score = ( self.correct_answers *
Quiz.points_correct_answer
```



```

        + self.wrong_answers *
Quiz.points_wrong_answer )
    return f'{self.score}/{self.max_score}'

```

First, the number of correct answers and wrong answers are showed and then the score is calculated. The value of score is equal to number of correct answers into points for correct answer plus number of wrong answers into points for wrong answers. Then we return the result in string form.

We can add some test code at the end.

```

if __name__ == '__main__':
    quiz = Quiz('Maths')
    quiz.start()
    print(quiz.get_result())

```

We have created a Quiz instance object with Maths as the topic and then called the methods start and get_result. When the method start will execute, it will look for the file named maths.pck. So for this test code to run smoothly make sure that you have created this file and added questions to it by executing the maintainQuiz.py file.

Now we will see how to write the code for taking the quiz. This code goes in the file takeQuiz.py.

```

from quiz import Quiz
from time import ctime
with open('quizTopics.txt', 'r') as file:
    topics = file.readlines()
name = input('Enter your name : ')
print(f'\nWelcome {name}')

```

We imported the Quiz class from our quiz module and the ctime function from the time module. We open the quizTopics.txt file and read all the topics in a list named topics. Next, we ask the user to enter his or her name and then a welcome message is printed.

After this we will show all the available topics to the user.

```
print(f'You can take the quiz in any one of these  
{len(topics)} topics\n')  
for i in range(len(topics)):  
    print(f'{i+1}.{topics[i]}', end='')
```

We are iterating over the `topics` list and displaying each topic with a serial number. Next, we ask the user to enter a topic number and we get the topic name from the topic number. So now we have the topic name in variable `topic`.

```
topic_number = int(input(f'\nChoose your topic (1-  
{len(topics)}): '))  
topic = topics[topic_number-1].rstrip()
```

Now we will create a `Quiz` object and call `start` and `get_result` methods on it.

```
quiz = Quiz(topic)  
quiz.start()  
result = quiz.get_result()
```

The return value of `get_result` method is stored in the variable named `result`. We show the score to user and draw a line on the screen.

```
print(f'Your score is {result}')
```

```
print('.' * 50)
```

Next, we will write the user's score into the file `users.txt`.

```
with open('users.txt', 'a') as file:  
    file.write(f'{name},{topic},{ctime()},  
{result}\n')
```

We have opened the file `users.txt` in append mode and wrote the user name, the topic of the quiz, current time and the result. All these items will be separated by commas.

So, now user has taken the quiz, we have shown him the result and we have also written the result to the file. Now we will show the user his previous scores in this topic.

```

input(f'Press Enter to view the history of your
scores in {topic}\n')
with open('users.txt', 'r') as file:
    for line in file:
        data = line.split(',')
        if data[0] == name and data[1] == topic:
            print(f'You scored {data[3].rstrip()}
on {data[2]} ')

```

Here we are opening the file `users.txt` in read mode and then we are reading this file line by line. We split each line with comma as the separator, so `data` is a list of all the 4 items on a line. The first item is the name of the user, second item is the topic, third item is the date and time when the quiz was taken and fourth item is the score. Then we have written an `if` statement; if the name in the line read from the file is the name of the user, and topic is equal to current topic then we will show the third and fourth item to the user. This way we will be able to show the user all his previous scores in the topic in which he has taken the quiz just now.

We can put the whole code inside a loop, and ask the user if he wants to take the quiz again.

```

while True:
    print(f'You can take the quiz in any one of
these {len(topics)} topics\n')
    .....
    .....
    response = input('\nWant to take the quiz
again(y/n) : ')
    if response == 'n':
        break

```

So now the user can take the quiz again and again in the same topic or in a different topic.

Project : Snakes and Ladders Game

In this project we will implement the snakes and ladders game. There can be different variations of the game and its rules but the main objective remains the same. Let us have a look at the rules that we are going to follow while implementing the game.

It is a board game where the board is a 10x10 grid of 100 squares with numbers labelled on them. The numbers are printed in this way - 1 to 10, then 11 to 20, 21 to 30 and so on. So, in the first row we have to move left to right, then in second row right to left then left to right and so on. There are snakes and ladders drawn on the board.

100	99	98	97	96	95	94	93	92	91
81	82	83	84	85	86	87	88	89	90
80	79	78	77	76	75	74	73	72	71
61	62	63	64	65	66	67	68	69	70
60	59	58	57	56	55	54	53	52	51
41	42	43	44	45	46	47	48	49	50
40	39	38	37	36	35	34	33	32	31
21	22	23	24	25	26	27	28	29	30
20	19	18	17	16	15	14	13	12	11
1	2	3	4	5	6	7	8	9	10

Figure 14.11: Snakes and Ladders game board

The game can be played by 2,3 or 4 players and each player gets a different coloured pawn to move on the board. Each player starts at square 1 and moves forward. To move on the board, each player gets a turn to roll the dice. The number on the dice indicates how many squares the player should move his pawn forward. For example, if the pawn is on square 9 and the dice shows 4, then the pawn reaches square 13. If a player rolls a six, he gets an extra turn.

If the player's pawn lands up at the bottom of a ladder, he moves up all the way to the top square of the ladder. If the pawn lands up at the head of a snake, he has to move down to the tail of the snake.

If a pawn lands on a square on which there is already a pawn, then the pawn that is there already has to go to the starting point. For example, if the yellow

pawn is at square 25 and the red one also reaches there then the yellow pawn has to go back to starting point.

The player who reaches square 100 first is the winner. The player should land exactly on this square to win the game. For example, if the player is at 97, he has to roll 3 to win the game. If he rolls any other number greater than 3 then he cannot move. If he gets 1 or 2 he can move, but if he gets 4, 5 or 6 he cannot move.

So, these are the rules of the game, let us start implementing it in a file named `SnakesLadders.py`.

```
colours = ['BLUE', 'GREEN', 'RED', 'YELLOW']
while True:
    n = int(input('How many players : '))
    if n in {2, 3, 4}:
        break
    print('You can have only 2,3 or 4 players')
players = []
```

We take a list named `colours` in which we have four strings where each string is the name of a colour. Each player will be given a pawn of different colour from this list. Next, we ask how many players want to play the game. The number of users can be 2, 3 or 4 so we used a loop to validate the input. The program will proceed forward only if you enter 2, 3 or 4 otherwise this loop will keep on executing. Then we take an empty list named `players`, each element of this list will be a `Player` object so let us first write a `Player` class in another file named `player.py`.

```
class Player:
    def __init__(self, name, colour):
        self.name = name
        self.colour = colour
        self.place = 0
```

We have three instance variables; `name` is the name of the player, `colour` is the colour of pawn which is given to the player and `place` denotes the square number on the board where the player's pawn is placed at any time.

Initially we take this as zero and as the game proceeds this number will change. Next, we will write the method `_roll_dice`

```
def _roll_dice(self):  
    input(f'\n{self.name}({self.colour}), Press  
Enter to roll the dice')  
    roll = random.randint(1,6)  
    print(f'You rolled a {roll},', end=' ')  
    return roll
```

We display a message which shows the player's name and colour and asks him to press Enter to roll the dice. Then we call the `randint` function from the `random` module. This will give us a random number from 1 to 6. We need to import the `random` module for this. After this we display the number that the user has rolled and return this number from the method.

Next, we will write a method named `play`.

```
def play(self):  
    .....  
    return self.place
```

This method will be called when the player gets his turn to play. After playing his turn, the place of the player on the board will change, so this method is responsible for changing the place and it will also return the instance variable `place`.

Now, let us write the code for this function. The first thing that the user has to do while playing his turn is to roll the dice. So, first we call the method `_roll_dice` and get the number rolled by the player.

```
roll = self._roll_dice()  
self.place += roll  
print(f'You move to square {self.place}\n')
```

We add the rolled number to the place of the player, so the place now changes, and we tell the user his new place. If the place becomes 100, it means that the player has won,

```
if self.place == 100:  
    print(f'Game over ... {self.name} wins')
```

```
    return self.place
```

So, in this case we will print a message and just return from this method, because we have got a winner, the game is over and we do not need to do anything else.

The new place where the player has landed might be the bottom of a ladder or head of a snake. We have to check for these cases also and have to update the place accordingly. For this we should know where the snakes and ladders are present on the board. So, we will store this information in 2 dictionaries. We can define these dictionaries at the top of the file outside the `Player` class.

```
ladders = {3: 34, 9: 14, 12: 96, 20: 42, 32: 51,
37: 65, 63: 99, 69: 90}
snakes = {15: 2, 31: 10, 34: 24, 40: 25, 81: 43,
84: 57, 87: 55, 92: 18}
```

In the `ladders` dictionary, key is bottom of the ladder and value is the top of the ladder so if a player lands on 3 he will go to 34, if he lands on 9 he will go to 14 and so on. In the `snakes` dictionary, key denotes the snake's head and the value denotes the snake's tail. If a player lands on 15 he has to go back to 2, if he lands on 31 he has to go back to 10 and so on.

In our `if` statement, we will add `elif` clauses to check whether the place where the player has landed is at the bottom of a ladder or at the head of a snake.

```
        if self.place == 100:
            print(f'Game over ... {self.name}
wins')

            return self.place
        elif self.place in ladders.keys():
            print('You landed on a ladder,', end='
')

            self.place = ladders[self.place]
            print(f'Climb to {self.place}\n')
        elif self.place in snakes.keys():
```

```

        print(f'You landed on a snake,', end='
')
        self.place = snakes[self.place]
        print(f'Move down to {self.place}\n')
    else:
        pass

```

If the player lands at the bottom of a ladder, then we change the place to the top of the ladder and if he lands at the head of snake then we change the place to the tail of the snake. If none of these cases is True then we do not need to do anything so we just write pass in the else clause.

We know that the player gets an extra turn if he rolls a six. So, we will put the whole code in an infinite loop and will check for that case at the end.

```

def play(self):
    while True:
        roll = self._roll_dice()
        self.place += roll
        print(f'You move to square
{self.place}\n')
        if self.place == 100:
            .....
            .....
        if roll == 6:
            print('You get another chance for
rolling a 6')
            continue
    return self.place

```

If the rolled number was 6, then the continue statement will be executed, and it will make this loop execute again and the player will get a chance to roll the dice again and the whole process will repeat.

While discussing the rules of the game we had seen that the player needs to land exactly at 100 to win. So, suppose if he is at 96, he needs to roll exactly 4 to win, if he rolls anything more than 4 then he cannot move. We have to

handle this situation also, when he rolls a dice but is not able to move. This checking code is written after the dice has been rolled. If the place plus the rolled number becomes greater than 100 then we tell the user that he cannot move, and we show him the number that he needs to roll in order to win. For example, if the user is at 96, and if he rolls a 5 or 6 then a message will be displayed which shows that he cannot move. If he rolls a 4, he wins and if he rolls a 1,2 or 3 he can simply move forward.

```
def play(self):
    while True:
        roll = self._roll_dice()
        if self.place + roll > 100:
            print(f'You cannot move, you need to
roll a {100-self.place} to win\n')
            return self.place
        self.place += roll
        print(f'You move to square
{self.place}\n')
        .....
        .....
```

So, this was the play method of our Player class.

Now let us go back to our main file SnakesLadders.py. We need to import the Player class in this file. As we have seen before, in the players list we will store the instance objects of the Player class.

```
import random
from player import Player
colours = ['BLUE', 'GREEN', 'RED', 'YELLOW']
n = int(input('How many players : '))
while n not in {2, 3, 4}:
    print('You can have only 2,3 or 4 players')
    n = int(input('How many players : '))
players = []
```

```

for i in range(n):
    name = input(f'Enter name of player{i+1}: ')
    colour = random.choice(colours)
    players.append(Player(name, colour))
    colours.remove(colour)
print()

```

The `for` loop iterates `n` times, where `n` is the number of players. So, suppose the number of players is 3 then this loop iterates 3 times. In each iteration, we will get the name of the player and will assign a colour to the player from the `colours` list.

We need to import the `random` module since we have used the `choice` function from this module.

Next, we create a `Player` instance object with entered name and chosen colour and append this object to the `players` list.

After this we remove this colour from the `colours` list so that no other player gets the same colour.

Next, we iterate over the `players` list and display the names and colours of all the players in this list.

```

for player in players:
    print(f'{player.name} gets {player.colour}
coloured pawn')

```

Now, we create a list named `positions` using the following list comprehension.

```

positions = [None for i in range(101)]

```

The size of this list is 101 and it has indices from 0 to 100. We will use elements from index 1 to index 100 to represent the squares on the board. Initially this list contains `None` at all the locations, so initially this list will be like:

```

[None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,

```



```
None, None, None, None, None, None, None, None,  
None, None, None, None, None, None, None, None,  
None, None, None, None, None, None, None, None,  
None, None, None, None, None]
```

This is how we can use the `positions` list to represent our board and maintain the positions of the players on the board. Now let us continue adding code to our file.

We take a Boolean variable `game_over` and make it `False` initially. Then we write a `while` loop which iterates till this variable is `False`.

```
game_over = False  
while not game_over:  
    for player in players: #each player's turn  
        current_position = player.place  
        new_position = player.play()
```

Inside the `while` loop we write a `for` loop, to give each player his turn. So, this `for` loop executes once for each player.

We save the current position of the player in the variable `current_position` and then call the method `play` to get the new position of the player. We save that position returned by the method in the variable `new_position`.

We have already discussed the method `play`, it takes care of all the things like sliding down a snake, climbing a ladder or extra turn for rolling a six. This method returns us the proper new place for the player after checking all this.

The next statement inside `for` loop would be an `if` statement. If `new_position` of the player is 100 then we make `game_over` `True` and write a `break` statement that takes the control out of the `for` loop.

```
if new_position == 100:  
    game_over = True  
    break
```

The `break` terminates the `for` loop. Making the variable `game_over` `True` means that the `while` loop will not iterate anymore.

After this `if` statement, we write another `if` statement for the ‘cannot move situation’. We have seen this situation in the `play` method. This situation occurs when the player is near 100 and rolls a number that makes the total more than 100 (eg. player at 98 and rolled a 5). In this situation `new_position` will be equal to `current_position` as the `play` method would not change the position.

```
        if new_position == current_position:
#cannot move situation
            continue
```

In this case we write the `continue` statement that takes the control to the start of the `for` loop, so the next player gets his turn.

After this, we check whether the new position to which the player has to move is already occupied. If it is occupied (not `None`), then we need to find the player who is at this position. After finding that player we will send him to zero.

```
        if positions[new_position] is not None:
#someone is already present
            for p in players:      #find the player who
is at that position
                if p.colour[:3] ==
positions[player.place]:
                    print(f'Position occupied by
{p.name} ')
                    positions[p.place] = None
                    p.place = 0
                    print(f'{p.name} goes back to 0')
                    break
```

To find that player we iterate over the `players` list and find that player by checking the string that is present at this place in the `positions` list. We print this message that tells the name of the player who is already present at the position where the current player has to move. We put `None` at this place. We make the place of the player equal to 0. So we have sent this

player back to zero and made this slot `None` so that our current player can move here. The `break` statement terminates this `for` loop.

Now we make the current player move to his new position. So, in the `positions` list, we make the `current_position` slot `None` and put the string of the colour of the player in this place.

```
positions[current_position] = None
positions[new_position] = player.colour[:3]
```

So, this was the implementation of our snakes and ladders game. We can execute the `SnakesLadders.py` file to play the game. It would be good if we could print the game board after each move, so next we will see how to do this.

We have this snakes and ladders game board, if we have to print it in our program, we will print it from top to bottom, so we have to print 10 rows.

Row 1	100	99	98	97	96	95	94	93	92	91
Row 2	81	82	83	84	85	86	87	88	89	90
Row 3	80	79	78	77	76	75	74	73	72	71
Row 4	61	62	63	64	65	66	67	68	69	70
Row 5	60	59	58	57	56	55	54	53	52	51
Row 6	41	42	43	44	45	46	47	48	49	50
Row 7	40	39	38	37	36	35	34	33	32	31
Row 8	21	22	23	24	25	26	27	28	29	30
Row 9	20	19	18	17	16	15	14	13	12	11
Row 10	1	2	3	4	5	6	7	8	9	10

Figure 14.12: Printing a Snakes and Ladders board

In first row we have to print from 100 to 91 then in the second row we have to print from 81 to 90, then 80 to 71 and so on. Let us see how we can do it.

We print the first row in reverse order starting from 100. Then we subtract 19 from 100, we get 81. Now we print the second row in forward order starting from 81. Then we subtract 1 from 81, we get 80. And we print next row in reverse order. We repeat this process to print the board. After printing an odd row, we decrease the number by 19. After printing an even row, we decrease the number by 1. Odd rows are printed in reverse order and even

rows in forward order. The following function uses the trick given above to print the board.

```
def print_board():
    number = 100
    for row in range(1, 11):
        if row % 2 != 0: # odd rows 1,3,5...
            for i in range(number, number-10, -1):
                print(f'{str(i):>4}', end=' ')
            number -= 19
            print()
        else: # even rows 2,4,6...
            for i in range(number, number+10):
                print(f'{str(i):>4}', end=' ')
            number += 1
            print()
    print()
```

In the game, we have to print the positions of players also. We have the `positions` list that represents a board. So, we will make use of that list. Let us place this function definition in our file `SnakesLadders.py`. We will make some changes in it to show the players' positions on the board.

```
def print_board(positions):
    number = 100
    for row in range(1, 11):
        if row%2 != 0: # odd rows 1,3,5...
            for i in range(number, number-10, -1):
                print(f'{str(positions[i] if
positions[i] is not None else i):>4}', end=' ')
            number -= 19
            print()
        else: # even rows 2,4,6...
            for i in range(number, number+10):
```

```

        print(f'{str(positions[i] if
positions[i] is not None else i):>4}', end=' ')
        number -= 1
        print()
    print()

```

This function will take `positions` list as the argument. Instead of printing `i`, we are printing the value of the following expression.

```
positions[i] if positions[i] is not None else i
```

We have used an `if else` operator here. If the value at index `i` is not `None` then we print that value otherwise we print `i`.

For example, suppose at index 97, the string 'RED' is present in the `positions` list, then that string will be printed, otherwise 97 will be printed. So, this is how we can print the places of different players on this board.

Now we can call this function in our code:

```

while not game_over:
    for player in players: #each player's turn
        print_board(positions)
        print()
        .....
        .....

```

Project : Log in system

In this project we will create a log in system. When our program runs, the user should see these 3 menus:

1. Sign up
2. Login
3. Forgot password

If the user chooses the SignUp menu, then a new user account is created. A user account contains a username, password, phone number, 2 security questions that will be used if the user forgets the password. If the user chooses this menu, we will ask him to enter all these details.

If the username entered by the user already exists in the system, then we should ask him to enter another username. There will be some conditions for a valid password. If the password entered by the user does not meet these conditions, then we will get the password entered again. Here are the conditions for a valid password:

It should be at least 7 characters long, should have at least 2 letters, should have at least one digit, should not contain any whitespace. If any of these conditions is not met, we display a message telling the user why the password will not be taken. And then we ask the user to enter another password. We keep doing this till the user enters a valid password.

If the length of password is more than or equal to 12 characters, we consider it as strong password and tell the user that your password is strong. If length is less than 12 i.e. if it is 7,8,9, 10 or 11 we tell the user that this is a weak password and ask whether he wants to enter another password.

After entering username and password, we ask for user's phone number and then we show him 2 security questions. These 2 questions should be randomly chosen from a list of questions.

All these details of the user's account should be stored somewhere so that they can be retrieved when a user tries to login to the system.

If the user chooses the second menu option, "Login" then first we ask him the username. If the username does not exist in the system, then we ask him to enter the username again. After this we ask the user to enter the password. If the password does not match the username, print "Wrong password" and ask the user to enter another password. After three attempts of a failed password, don't ask for another password and block the user account which has this username. So, if a user enters wrong password 3 times, then that user account is blocked. Logging in to a blocked account will not be allowed. Thus, after getting the username, you need to first check whether the account is blocked. If the account is blocked then there is no need to proceed further and ask for password.

If the entered password is correct, print the Welcome message.

If the user chooses the third option “Forgot password” then first ask the username. If the account is blocked don’t proceed further. If it is not blocked then show the user 2 security questions that are stored for this user account. If the user answers both of them correctly then show him the password. If the user is not able to answer these questions correctly then generate and send a one-time password to the phone number that is stored for this user’s account. If the user enters the correct OTP then show him the password. In our project we will not write code to actually send any OTP to any number, we will just display the OTP on the screen.

So, this was the whole requirement of the project, let us start implementing it. First, we will create a User class.

-----user.py-----

```
class User:
    def __init__(self, username):
        self.username = username
        self.password = ''
        self.phone = ''
        self.security_questions = {}
        self.blocked = False
    def sign_up(self):
        pass
    def log_in(self):
        pass
    def forgot_password(self):
        pass
```

In the `__init__` method for this class, we have the instance variables `username`, `password`, `phone`, `security_questions` and `blocked`. The first three instance variables will be strings, `security_questions` will be a dictionary in which the key will be the security question and the corresponding value will be the answer.

Since we have decided to show 2 security questions, this dictionary will have 2 pairs of keys and values.

The Boolean instance variable `blocked` will be made `True` if the user's account is blocked. Initially when the user account is created this is taken to be `False`.

Before writing the code of the three methods of this class, let us first write the code in our main file.

-----password.py-----

```
from user import User
import pickle
print('1. Sign up')
print('2. Login')
print('3. Forgot password')
with open('users.pck', 'rb') as file:
    users = pickle.load(file)
usernames = [user.username for user in users]
response = int(input('Enter your choice : '))
```

We will import the `User` class and the `pickle` module. Initially we will show the three menu options. Next, we will open the file `users.pck` in read mode. This file contains a list of all `User` instance objects. We read this file using the `load` method of the `pickle` module. We get a list which contains all the `User` instance objects. Now from this list, we create another list that contains only the usernames. For this we have written a list comprehension in which we are iterating over the `users` list and getting the username for each user. After this we ask the user to enter his choice from the three menu options. This response will be 1, 2 or 3.

If the response is 1(Sign up), then we need to create a new user account.

```
if response == 1:
    name = input('Enter a username for your account : ')
    while name in usernames:
```

```

        name = input('This username already
present, enter another name : ')
        new_user = User(name)
        new_user.sign_up()
        users.append(new_user)

```

We ask for a username; if the username that was entered already exists, then we need to ask the user to enter another username because we can't have two user accounts with the same username; each account needs to have a unique username. In the while loop we are checking the name in the usernames list. This loop will keep on executing until the user enters a name that is not present in the username list. After we get the proper username, we create a new instance object with this username. Then, we call the `sign up` method on this newly created instance object, and we add the new instance object to the `users` list.

Now we will write the `else` part. In this part we will write code for the 2 options “Login” and “Forgot Password”.

In both the cases first, we ask the user to enter his username.

```

else: # Login or Forgot Password
    name = input('Enter your username : ')
    while name not in usernames:
        print('Invalid Username')
        name = input('Enter a valid username : ')
    for user in users:
        if user.username == name:
            if response == 2:
                user.log_in()
            else:
                user.forgot_password()

```

If the username that is entered is not present in the `usernames` list, then we will tell the user that this is an invalid username and will ask him to enter the username again. For this we have written a loop. You can see this loop condition is opposite to that of the loop condition that we wrote in the case

of option 1. There we wanted a username that is not present in the `usernames` list and here we want a username that is present in the list. So, this loop will keep on executing till the user enters a username that is present in the list.

After getting the username, we iterate over the `users` list that contains all the instance objects. And then we call the `log_in` method or the `forgot_password` method for the `User` object that has this name as its username.

```
with open('users.pck', 'wb') as file:
    pickle.dump(users, file)
```

At the end we dump this `users` list to the file `users.pck`.

Now let us finish writing the code for our `User` class. First let us write the code for the `signup` method. The `User` instance object is created with a username, so the username is already there. Next thing we need to get is the password. There are many conditions that need to be checked for a password to be valid, so instead of getting the password using the `input` function we will create another method `_enter_password`. Next, we get the phone number from the user. Then we will show 2 security questions and will get the answers for them.

We have stored all the security questions in a file named `questions.txt`. This file should contain a question on each line. Inside the `sign_up` function we will open this file in read mode and get all the questions in a list. We will shuffle this list using the `shuffle` function from `random` module. After this we show the first 2 questions from this shuffled list and get the answers for them. Next, we place the 2 questions and answers in the dictionary `security_questions`. The question becomes the key and answer becomes the value.

```
def sign_up(self):
    self.password = self._enter_password()
    self.phone = input('Enter your phone number :
')
    print('Answer these two security questions')
```

```

        print('These questions will help you login if
you forget your password ')
        with open('questions.txt', 'r') as file:
            questions = file.readlines()
            random.shuffle(questions)
            answer0 = input(questions[0])
            answer1 = input(questions[1])
            self.security_questions[questions[0]] = answer0
            self.security_questions[questions[1]] = answer1

```

By calling this method we will get the password, phone number and security questions for the User object.

Now, let us write the code for the method `_enter_password`.

We have written the input statement inside an infinite loop, because we want to keep on asking for password till the user enters a valid password.

```

def _enter_password(self):
    while True:
        password = input('Enter password : ')
        if len(password) < 7:
            print('Password should have at least 7
characters')
            continue
        a = d = w = 0
        for ch in password:
            if ch.isalpha():
                a += 1
            elif ch.isdigit():
                d += 1
            elif ch.isspace():
                w += 1
        if a < 2:

```

```

        print('Password should have at least 2
letters')
        continue
    if d == 0:
        print('Password should have at least
one digit')
        continue
    if w > 0:
        print('Whitespace not allowed')
        continue
    if len(password) < 12:
        print('Weak password ')
        response = input('Do you want to enter
another password : (yes/no)')
        if response == 'yes':
            continue
        else:
            break
    else:
        print('Strong password ')
        break
    return password

```

If the length of password is less than 7, we display a message and write the `continue` statement. In this case the control will come to the top of the loop and the user is again asked to enter a password. Next, we count the number of alphabetic characters, digits and whitespace in the entered password. For this we have written a for loop in which we are iterating over the password string and counting all these things.

If number of alphabetic characters is less than 2 or if there is no digit or if there is any whitespace in the password, then we print an appropriate message and write `continue` statement.

If all the conditions are False, then we have checked whether the password entered by the user is weak or strong.

If the length of password is less than 12, then we tell the user that the entered password is weak and give him a chance to enter the password again. If he wants to enter another password then we continue otherwise we break out of the while loop.

If the length of the password is more than or equal to 12 then we tell the user that his password is strong and break out of the loop. At the end we return the password from this method.

Now let us write the code for the method `log_in`.

```
def log_in(self):
    if self.blocked == True:
        print('This account is blocked')
        return
    psswd_attempts = 1
    password = input('Enter password : ')
    while password != self.password:
        if psswd_attempts == 3:
            self.blocked = True
            print('Sorry, no more tries !\n')
            break
        print('Wrong Password')
        password = input('Enter correct password : ')
        psswd_attempts += 1
    else:
        print(f'Welcome, {self.username}')
```

If the account is blocked, we will just return. Otherwise, we will ask for the password. If the entered password is wrong, we ask the user to enter the correct password. We repeatedly ask for the correct password in a loop and we want the user to stop after three failed attempts. So, we will count the

number of attempts and when it becomes equal to 3, then we will block the account, tell the user that he cannot try anymore and break out of the loop.

In the else part of this loop we print the welcome message, because the control will come here when the loop terminates normally i.e. when the condition `password != self.password` becomes False. This condition will be False when the entered password matches the password of the User object, so in that case login is successful.

Now let us come to the method `forgot_password`.

```
def forgot_password(self):
    if self.blocked == True:
        print('This account is blocked')
        return

    for question, answer in
self.security_questions.items():
        response = input(question)
        if response != answer:
            print('You answered it wrong')
            print('Sending an OTP to your phone
ending with ', self.phone[-5:])
            self._send_and_check_otp()
            break
        else:
            print('Your password is ', self.password)
            self.log_in()
```

If the account is blocked, we just print a message and return, otherwise we iterate over the dictionary and show the 2 security questions and get the answers from the user. If any answer is wrong, we call the method `_send_and_check_otp` and break out of the loop.

In the else part of the for loop we show the password and then call the `log_in` method so that the user can now log in with this password. Now let us see the method `_send_and_check_otp`.

```
def _send_and_check_otp(self):
```

```

        otp = random.randint(100000, 999999)
        print(otp)
        n = int(input('Enter the otp sent on your
phone'))
        if n == otp:
            print('Your password is ', self.password)
            self.log_in()
        else:
            print('Wrong OTP')

```

First, we generate a random number between 100000 and 999999 using the `randint` function from the `random` module. We need to send this OTP to the phone, instead of that we are just printing it here. After this we ask the user to enter the OTP, if it is correct, we show the password to the user and call the `log_in` method otherwise we print that the OTP is wrong. So now we have seen the full code for the `User` class.

In our main file `password.py`, we are opening the file `users.pck` and getting a list from it. Before executing this file for the first time we need to create the file `users.pck` with an empty list so that we don't get `FileNotFoundError`. We can execute this file that contains code to create a binary file `users.pck` and dumps an empty list into this file.

-----createuserfile.py-----

```

import pickle
with open('users.pck', 'wb') as file:
    pickle.dump([], file)

```

This code needs to be executed only once in the beginning. If you execute it after you have all the user data in the file, then that data will be erased and you will have a file that has any empty list in it. It is because we have opened this in write mode. If you want to avoid this you can open it in append mode.

After executing this file, we can go to the main file `password.py` and execute it. The `users.pck` file has an empty list so we can make few user

accounts by choosing the Sign Up option. After creating a few accounts, we can check the Login and Forgot Password options.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Magic Methods

15

Magic methods are specially named methods that we can define to make our classes behave like built-in types. With the help of these methods, we can add, subtract or compare our instance objects or we can even index or slice them like other built-in sequences.

These special methods begin and end with double underscore, to distinguish them from other user defined names. They are also called dunder methods due to the double underscore added before and after their name. Here are a few examples of dunder methods.

`__init__` `__add__` `__mul__` `__sub__` `__eq__`
`__len__`

The names of these methods are predefined, and each one has a specific purpose. We are already familiar with `__init__`, which is the most commonly used dunder method. It is called automatically by Python when an instance object is created; we do not have to explicitly call it. Similarly other magic methods are also automatically called in response to certain actions. For example, the method `__add__` will be called when two objects are added using the `+` operator, and the method `__len__` will be called when an object is sent as argument to the `len` built-in function.

These methods are called magic methods as they are magically (automatically) called when a user-defined type is used with a built-in operator or in a particular syntax. Most operators and built-in functions have a specially named method corresponding to it. For example, there is `__sub__` for the `-` operator, `__mul__` for the `*` operator, and so on.

By default, most operators and built-in functions will not work with the objects of user-defined classes. If you want an operator to work for your class, you must define the corresponding magic method. If the magic method corresponding to an operator is not defined in your class and you try to use that operator with your instance objects, then Python will raise an error.

By defining these special methods, we can specify what happens when a built-in operation is used with instance objects of our class. This process is called operator overloading as we are overloading operators by giving them the capability to operate on different types. By overloading an operator, we tell that operator to behave differently depending on the type of its operand.

We have already encountered this concept of operator overloading when we used addition and multiplication operators with numeric types and strings.

```
4 + 5                # Add
'hello' + 'world'    # Concatenate
2 * 3                # Multiply
'hello' * 3           # Repeat
```

The plus operator knows that when its operands are of a numeric type, it needs to add them, and when operands are sequence type, it needs to concatenate them. Similarly, the multiplication operator, when used with numbers, means multiplication, and when used with sequences, means repetition. We can say that the `+` and `*` operators are overloaded. If we want these addition and multiplication operators to work for our instance objects, we need to define the methods `__add__` and `__mul__`. If these methods are defined in our class, then Python will invoke them whenever addition and multiplication operators are used with our instance objects.

So, we can make our instance objects behave like built-in type objects by defining these magic methods. This way, we can have consistency in the interface provided by the built-in classes and the user-defined classes. Being able to multiply or add two instance objects using operators is much more convenient than using different method names. For example, the expression `f1 + f2 * f3` is clearer and more readable than the verbose and confusing expression `f1.add(f2.multiply(f3))`. All developers are familiar with the interface provided by the built-in classes, and it would be good if we could provide the same interface for our own classes. The

familiar interface makes our instance objects intuitive to use. If classes in different libraries use the built-in interface for common operations, developers will not need to learn and get used to different method names.

When we define our own class, we have to decide what type of built-in operations we need our class to support. For example, when making a class `Fraction` or `Matrix` we would like to overload arithmetic operators, but not for `Person` or `Employee` class. There is no sense in adding or multiplying an `Employee` object by another `Employee` object. So, overload only those operators in your class that make sense for the type that you are defining.

You can write any code inside the definition of these methods, but the code that you write should not be totally unexpected; for example, you should not write code for subtracting two fractions in the `__add__` method, although it is possible to do so. It makes things confusing for the user of the class.

There are several predefined magic methods in Python that can be used for different purposes. We will discuss some of the most commonly used ones in this chapter. Some of them will be explored in the *Chapters 17* and *21*.

15.1 Overloading Binary Arithmetic operators

We had created this `Fraction` class in the exercise of the previous chapter:

```
class Fraction:
    def __init__(self, nr, dr=1):
        self.nr = nr
        self.dr = dr
        if self.dr < 0:
            self.nr *= -1
            self.dr *= -1
        self._reduce()
    def show(self):
```

```

        print(f'{self.nr}/{self.dr}')
    def add(self, other):
        if isinstance(other, int):
            other = Fraction(other)
        f = Fraction(self.nr * other.dr + other.nr
* self.dr, self.dr * other.dr)
        f._reduce()
        return f
    def multiply(self, other):
        if isinstance(other, int):
            other = Fraction(other)
        f = Fraction(self.nr * other.nr , self.dr *
other.dr)
        f._reduce()
        return f
    def _reduce(self):
        h = Fraction.hcf(self.nr, self.dr)
        if h == 0:
            return
        self.nr //= h
        self.dr //= h
    @staticmethod
    def hcf(x, y):
        x = abs(x)
        y = abs(y)
        smaller = y if x > y else x

```

```

s = smaller
while s > 0:
    if x % s == 0 and y % s == 0:
        break
    s -= 1
return s

```

If we try to add or multiply two `Fraction` objects using the `+` operator or `*` operator, then we will get a `TypeError`.

```

>>> f1 = Fraction(2, 3)
>>> f2 = Fraction(3, 4)
>>> f3 = f1 + f2
TypeError: unsupported operand type(s) for +:
'Fraction' and 'Fraction'
>>> f4 = f1 * f2
TypeError: unsupported operand type(s) for *:
'Fraction' and 'Fraction'

```

In our class, we have defined the methods `add` and `multiply` that can be used to add or multiply two objects of `Fraction` type.

```

>>> f3 = f1.add(f2)
>>> f3.show()
17/12
>>> f4 = f1.multiply(f2)
>>> f4.show()
1/2

```

If we want our `Fraction` objects to respond to `+` and `*` operators, then we have to define the magic methods named `__add__` and `__mul__` in our class. These methods will do exactly the same work that the methods `add` and `multiply` are doing.


```

def __add__(self, other):
    if isinstance(other, int):
        other = Fraction(other)

    f = Fraction(self.nr * other.dr + other.nr
* self.dr, self.dr * other.dr)

    f._reduce()
    return f

def __mul__(self, other):
    if isinstance(other, int):
        other = Fraction(other)

    f = Fraction(self.nr * other.nr , self.dr *
other.dr)

    f._reduce()
    return f

```

The first parameter as usual is `self` that refers to the object that invokes the method, the second parameter is conventionally named `other`. The parameter `self` will refer to the object on the left side of the operator and the parameter named `other` will refer to the object on the right side of the operator.

Inside the `__add__` method, we create a new `Fraction` object by adding the two fractions and return it from the method. Similarly, in the `__mul__` method we return a new `Fraction` object that is the result of multiplying two `Fraction` objects.

```

>>> f1 = Fraction(2,3)
>>> f2 = Fraction(3,4)
>>> f3 = f1.__add__(f2)
>>> f3.show()

```

17/12

These special methods are generally not called directly like this in the code. They are called automatically when the related syntax is used. If we add two `Fraction` objects, Python will call `__add__` method automatically.

```
>>> f3 = f1 + f2
```

```
>>> f3.show()
```

```
17/12
```

```
>>> f4 = f1 * f2
```

```
>>> f4.show()
```

```
1/2
```

The expression `f1 + f2` is converted to the method call `f1.__add__(f2)`. This method is called on object `f1`, and `f2` is sent as the parameter. The return value of the method is the value of the expression `f1 + f2`. Similarly, `f1 * f2` is converted to the method call `f1.__mul__(f2)`. The interpreter did this magic for us and it will do this translation only if we use the special dunder names. So, whenever the interpreter will see any operator working on a user defined type, it will look for the corresponding dunder method in the class and invoke it, if it is present.

Similarly, we can add a magic method for subtraction. Name of the method for subtraction is `__sub__`.

```
def __sub__(self, other):
    if isinstance(other, int):
        other = Fraction(other)

    f = Fraction(self.nr * other.dr - other.nr
    * self.dr, self.dr * other.dr)

    f._reduce()

    return f
```

Now, we can subtract an integer or a `Fraction` from another `Fraction`.

Here are some binary operators and their corresponding magic methods:

<code>a + b</code>	<code>a.__add__(b)</code>
<code>a - b</code>	<code>a.__sub__(b)</code>
<code>a * b</code>	<code>a.__mul__(b)</code>
<code>a / b</code>	<code>a.__truediv__(b)</code>
<code>a // b</code>	<code>a.__floordiv__(b)</code>
<code>a % b</code>	<code>a.__mod__(b)</code>
<code>a ** b</code>	<code>a.__pow__(b)</code>

Table 15.1: Magic methods for binary operators

15.2 Reverse methods

The special methods of binary operators we have discussed come with corresponding reverse (reflected) variants as well. These reverse variants have the same spelling but start with an `r` prefix. For example, the reverse variant of `__add__` method is `__radd__` method. These reverse methods are used when an operation is performed between objects of different types.

In the definition of `__add__` method of our `Fraction` class, if the parameter is an `int`, then we convert it to `Fraction` and then perform the addition. So, this method is capable of adding a `Fraction` object to an integer.

```
>>> f1 = Fraction(2, 3)
>>> f2 = f1 + 3
>>> f2.show()
11/3
```

Now, let us see what happens if we write integer as the left-hand operand.

```
>>> f2 = 3 + f1

TypeError: unsupported operand type(s) for +: 'int'
and 'Fraction'
```

This did not work because now the left-hand operand is not a `Fraction`.

When the binary operator `+` is evaluated, the interpreter first checks if the class of the left-hand operand provides a `__add__` method that supports the type of the right-hand operand. The expression `f1 + 3` worked because the left-hand operand is of `Fraction` type, and `Fraction` class has a `__add__` method which knows how to add an `int`. The expression `3 + f1` did not work because the left-hand operand is of `int` type, and `int` class does not have a `__add__` method that can add our `Fraction` type.

If the first check fails, the interpreter performs another check. It checks the class of the right-hand operand to see whether it provides a dunder reverse method that supports the type of the left-hand operand. In the case of expression `3 + f1`, since the first check fails, the interpreter will perform a second check in which it will look for a `__radd__` method in `Fraction` class, that supports an `int`. It does not find any, so it fails. To make the expression `3 + f1` work, let us now provide a dunder reverse add method in our `Fraction` class.

```
def __radd__(self, other):  
    return self.__add__(other)
```

In this method, we are just calling the `__add__` method, because the addition operation is supposed to be commutative and `3 + f1` is supposed to be same as `f1 + 3`.

```
>>> f2 = 3 + f1  
>>> f2.show()  
11/3
```

Now, the expression `3 + f1` works. The interpreter calls `__radd__` method on the instance object `f1` and passes `3` as the parameter to this method. The expression `f1 + 3` is evaluated as `f1.__add__(3)` and the expression

`3 + f1` is evaluated as `f1.__radd__(3)`.

To support these mixed-type operations, you can define reverse variants. If the operation is commutative, you can call the normal equivalent method in the reverse variant. For example, if `obj + x` is the same as `x + obj`, then you can just call the `__add__` method in the `__radd__` method. If an

operation is non-commutative, then you can define different behaviour in the reverse method. The subtraction operation is generally non-commutative, so you will have to define the reverse method appropriately.

Here are the normal and reverse methods for different binary operations:

Binary operators	Forward call	Reverse call
<code>a + b</code>	<code>a.__add__(b)</code>	<code>b.__radd__(a)</code>
<code>a - b</code>	<code>a.__sub__(b)</code>	<code>b.__rsub__(a)</code>
<code>a * b</code>	<code>a.__mul__(b)</code>	<code>b.__rmul__(a)</code>
<code>a / b</code>	<code>a.__truediv__(b)</code>	<code>b.__rtruediv__(a)</code>
<code>a // b</code>	<code>a.__floordiv__(b)</code>	<code>b.__rfloordiv__(a)</code>
<code>a % b</code>	<code>a.__mod__(b)</code>	<code>b.__rmod__(a)</code>
<code>a ** b</code>	<code>a.__pow__(b)</code>	<code>b.__rpow__(a)</code>
<code>a << b</code>	<code>a.__lshift__(b)</code>	<code>b.__rlshift__(a)</code>
<code>a >> b</code>	<code>a.__rshift__(b)</code>	<code>b.__rrshift__(a)</code>
<code>a & b</code>	<code>a.__and__(b)</code>	<code>b.__rand__(a)</code>
<code>a ^ b</code>	<code>a.__xor__(b)</code>	<code>b.__rxor__(a)</code>
<code>a b</code>	<code>a.__or__(b)</code>	<code>b.__ror__(a)</code>

Table 15.2: Normal and reverse magic methods for binary operators

The interpreter will try the reverse methods only if the corresponding method is not defined or if it returns `NotImplemented`.

`NotImplemented` is a special value that should be returned by the binary magic methods to indicate that the operation is not implemented with respect to the other type. Let us understand this with the help of an example:

```
class A:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
```

```

        if isinstance(other, A):
            return self.value + other.value
        elif isinstance(other, float) or
isinstance(other, int):
            return self.value + other
        else:
            return NotImplemented

class B:
    def __init__(self, data):
        self.data = data
    def __radd__(self, other):
        if isinstance(other, A):
            return self.data + other.value
        elif isinstance(other, B):
            return self.data + a.data
        else:
            return NotImplemented

a = A(1)
b = B(2)
print(a + b)

```

We have two classes, A and B, and we are trying the operation `a + b`, where `a` is an object of class A and `b` is an object of class B. The first call that the interpreter tries is `a.__add__(b)`. In our class A, the `__add__` method has implemented addition with objects of type A, int, and float. If it gets any other type of object, it does not know how to add it, so it returns `NotImplemented` in this case. This makes sure that the interpreter will try the `__radd__` method of the other class. So, while trying to evaluate `a + b`, the interpreter will try the `__radd__` method of class B.

This method knows how to add an object of type A, so the result of the call `b.__radd__(a)` becomes the result of the operation `a + b`. If class B had also not known how to add an object of type A, then the interpreter would have raised `TypeError`.

So, if an operator method cannot return a valid result for another type, it should return `NotImplemented` instead of returning something else or raising `TypeError`. This way, the interpreter gets the opportunity to perform the operation from the reverse side. If an operator method is defined but returns a value other than `NotImplemented`, then the interpreter will not look for the reverse method in the class of right-hand operand.

In the case when we added an `int` to our `Fraction` class(`3 + f1`), the `__radd__` method was called because the `__add__` method of `int` class returned `NotImplemented`.

15.3 In-place methods

In addition to reverse variants, Python also provides the in-place variants of special methods for binary operators. These in-place variants have the same spelling as their normal equivalents but start with an `i` prefix. These methods are called when we write an augmented assignment statement. We know that an augmented assignment statement is a shortcut for an operation and assignment statement. For example, `a += b` is a shortcut for `a = a + b`.

You can use these in-place methods to define in-place operations on objects. When we studied lists, we saw that augmented assignments are more efficient as they make in-place changes in the object instead of creating a new object. If we want similar behaviour for our objects, we can define the augmented assignment special methods.

Here is the in-place variant for the addition operator of our `Fraction` type

```
def __iadd__(self, other):
    if isinstance(other, int):
        self.nr = self.nr + other * self.dr
    else:
```

```

        self.nr = self.nr * other.dr + other.nr
    * self.dr

    self._reduce()

    return self

```

We are changing `self` and returning it from the method. This behavior differs from that of the `__add__` method which should always return a new object.

If an augmented assignment method is not defined and you write an augmented assignment expression for your instance objects, then the regular method is used. For example, to evaluate `a += b`, first `__iadd__` is checked; if it is not defined, then `__add__` is considered. So, the augmented syntax (such as `a += b` or `a *= b`) is supported for your class even if you have not defined the in-place variants, provided the definitions of normal equivalents are there. However, if you do not define the in-place variant, the operation would not be in-place as the regular methods return a new object.

If you define an in-place variant that modifies `self`, then aliasing between references is not broken after using the augmented assignment statement. In the following example, `f1` and `f2` are referring to the same object and after the statement `f1 += 1` also, they refer to the same object. This is because `f1 += 1` made in-place changes to object referred to by `f1`, instead of making `f1` refer to a new object.

```

>>> f1 = Fraction(1, 2)
>>> f2 = Fraction(1, 3)
>>> f1 = f2
>>> f1 += 1
>>> f1
<__main__.Fraction object at 0x000001B116750050>
>>> f2
<__main__.Fraction object at 0x000001B116750050>

```



```
>>> f1.show()
```

```
4/3
```

```
>>> f2.show()
```

```
4/3
```

If you do not define an in-place variant, then also the augmented syntax is supported but then `f1 += 1` will be evaluated as `f1 = f1 + 1`, where `f1 + 1` returns a new object that is assigned to `f1`. This breaks the aliasing between references `f1` and `f2`.

```
>>> f1 = Fraction(1,2)
```

```
>>> f2 = Fraction(1,3)
```

```
>>> f1 = f2
```

```
>>> f1 += 1
```

```
>>> f1
```

```
<__main__.Fraction object at 0x0000023691A17E50>
```

```
>>> f2
```

```
<__main__.Fraction object at 0x00000236912D0050>
```

```
>>> f1.show()
```

```
4/3
```

```
>>> f2.show()
```

```
1/3
```

The special methods for augmented assignment can return an object other than `self`, but if you want in-place operation then you should modify `self` in-place and return `self`. In-place methods are there so that you can implement augmented assignment efficiently, if there is a need to do so. Immutable built-in types like strings and tuples don't make in-place changes in the object when augmented assignment is used, because immutable types cannot be changed in-place. If the type that you are defining is immutable, then the in-place variants that modify `self` should not be defined in your class. If your type is supposed to be mutable then in-place variants can be

defined for optimized in-place changes. Here are the names of in-place variants of binary operators:

<code>a += b</code>	<code>a.__iadd__(b)</code>
<code>a -= b</code>	<code>a.__isub__(b)</code>
<code>a *= b</code>	<code>a.__imul__(b)</code>
<code>a /= b</code>	<code>a.__itruediv__(b)</code>
<code>a //= b</code>	<code>a.__ifloordiv__(b)</code>
<code>a %= b</code>	<code>a.__imod__(b)</code>
<code>a **= b</code>	<code>a.__ipow__(b)</code>
<code>a <<= b</code>	<code>a.__ilshift__(b)</code>
<code>a >>= b</code>	<code>a.__irshift__(b)</code>
<code>a &= b</code>	<code>a.__iand__(b)</code>
<code>a ^= b</code>	<code>a.__ixor__(b)</code>
<code>a = b</code>	<code>a.__ior__(b)</code>

Table 15.3: Magic methods for augmented assignment

15.4 Magic Methods for comparison

By defining the magic methods for comparison, you can compare your instance objects with the help of standard relational operators. Here are the special methods corresponding to the six relational operators:

<code>a == b</code>	<code>a.__eq__(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>
<code>a < b</code>	<code>a.__lt__(b)</code>
<code>a > b</code>	<code>a.__gt__(b)</code>
<code>a <= b</code>	<code>a.__le__(b)</code>
<code>a >= b</code>	<code>a.__ge__(b)</code>

Table 15.4: *Magic methods for comparison operators*

The operators `==` and `!=` are available by default for every class that we define, this means that we can use them with our instance objects without defining any special methods. The default implementation compares the references; if they refer to the same object, they are considered equal otherwise not.

```
>>> f1 = Fraction(2, 3)
>>> f2 = Fraction(2, 3)
>>> f3 = f2
>>> f1 == f2
False
>>> f2 == f3
True
>>> f1 != f2
True
```

The value of the two fractions `f1` and `f2` is same ($2/3$) but still we get `False`. This is because by default for every class, two instance objects will be considered equal only if they are same object, otherwise they are considered unequal. If `f1` and `f2` refer to same object then only the expression `(f1 == f2)` will return `True` otherwise it returns `False`. So, by default `==` and `!=` operators behave like the `is` and `is not` operators and compare just the identities of the objects.

Generally, we do not want to compare objects based on their identities, instead we want to compare them based on their contents. For this we can define the magic method `__eq__`. Let us define this method for our `Fraction` class.

We can compare fractions by cross-multiplying, which means that the numerator of the left-side fraction is multiplied by the denominator of the right-side fraction and the denominator of the left-hand side is multiplied by the numerator of the right-side fraction. The two results are compared to find

whether the two fractions are equal and, if not equal, which one is smaller or bigger.

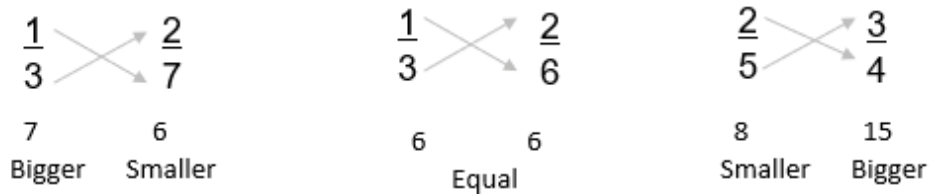


Figure 15.1: Comparing fractions

Based on this logic, here is our own implementation of equality for the `Fraction` class.

```
def __eq__(self, other):  
    return (self.nr * other.dr) == (self.dr *  
    other.nr)
```

Now this dunder method will be called when `==` operator is used with `Fraction` objects. Let us see the change in behaviour after defining this method:

```
>>> f1 = Fraction(2, 3)
```

```
>>> f2 = Fraction(2, 3)
```

```
>>> f3 = f2
```

```
>>> f1 == f2
```

```
True
```

```
>>> f2 == f3
```

```
True
```

```
>>> f1 != f2
```

```
False
```

Now the `Fraction` instance objects are compared based on the data that they contain. We can see that behaviour of `!=` operator has also changed. So, it is not necessary to define the `__ne__` method if you want it to behave just the opposite of `__eq__` method. If `__ne__` is not defined and

`__eq__` is defined, then whenever `!=` operator is used, the interpreter will execute `__eq__` and will invert the result.

There are no default implementations for the other four relational operators. If we want our class to support them, we must define the corresponding magic methods. Here is the method for the `<` operator:

```
def __lt__(self, other):  
    return (self.nr * other.dr) < (self.dr *  
other.nr)
```

Now, we can compare our `Fraction` instance objects using the `<` operator. We can also compare them using the `>` operator, as the interpreter will automatically provide the `__gt__` method.

```
>>> f1 = Fraction(2 , 3)
```

```
>>> f2 = Fraction(1, 5)
```

```
>>> f1 < f2
```

```
False
```

```
>>> f1 > f2
```

```
True
```

Defining both `__lt__` and `__eq__` methods does not mean defining the behaviour of `<=` operator. For that, you have to define the `__le__` method separately.

```
def __le__(self, other):  
    return (self.nr * other.dr) <= (self.dr *  
other.nr)
```

After defining this method, our class will support both `<=` and `>=` operators, as Python will automatically supply the `__ge__` method.

If we define the `__lt__` operator in our class, then our instance objects become sortable, they can be sorted by using the `sorted` built-in function or the list `sort` method and be used in built-in functions `min` and `max`.

```
f1 = Fraction(1, 100)
```

```

f2 = Fraction(2, 3)
f3 = Fraction(5, 6)
f4 = Fraction(1, 3)
L = [f1, f2, f3, f4]
for f in sorted(L):
    f.show()
min(L).show()
max(f1, f2, f3).show()

```

Output-

```

1/100
1/3
2/3
5/6
1/100
5/6

```

If you decorate your class with the `functools.total_ordering` decorator then your class can support all the comparison operators by defining just two magic methods. You need to define the `__eq__` methods and one of these methods (`__lt__`, `__gt__`, `__le__`, `__ge__`). However, this can lead to slower execution as compared to defining all six operators in the class.

```

from functools import total_ordering
@total_ordering
class Fraction:
    def __init__(self, nr, dr=1):
        .....
    def __eq__(self, other):

```

```
        return (self.nr * other.dr) == (self.dr *
other.nr)
```

```
    def __lt__(self, other):
        return (self.nr * other.dr) < (self.dr *
other.nr)
```

15.5 Comparing objects of different classes

If you intend to compare objects of your class with objects of other built-in classes or user defined classes then you should know what exactly happens when these operators are used.

As in binary expression operators, comparison operator methods can also return `NotImplemented` if the operation is not implemented for a given argument.

```
class MyClass:
    def __lt__():
        if isinstance(other, int):
            .....
        elif isinstance(other, MyClass):
            .....
        else:
            return NotImplemented
```

We have implemented the comparison of a `MyClass` object with an `int` object and a `MyClass` object. For any other type of object, the operation is not implemented. The `__lt__` method knows how to compare a `MyClass` object with an `int` object or a `MyClass` object. If it gets any other type, it does not know how to compare so it returns `NotImplemented`. Since `NotImplemented` is returned in this case, the interpreter tries to call some other method for performing the comparison. Comparison operators do not

have separate reverse variants like the binary arithmetic operators, but they are each other's reflections. Methods `__lt__` and `__gt__` are reflections of each other, `__le__` and `__ge__` are reflections of each other. The methods `__eq__` and `__ne__` are their own reflection.

Let us see what happens in the operation `a < b`, where `a` is an instance object of class `A` and `b` is an instance object of class `B`.

The interpreter calls the `__lt__` on the first object and passes the second object as the argument. This is equivalent to `a.__lt__(b)`. If this method returns a value other than `NotImplemented`, that value is returned and used as the result of the operation `a < b`. If the method `__lt__` is not defined in class `A`, or if it returns `NotImplemented` because it does not support comparison with objects of type `B`, then the interpreter looks for the reflection method in class `B`. If the reflection method `__gt__` is not defined in class `B`, then the interpreter raises an error. If it is defined and has implemented how to compare an object of class `B` with an object of class `A`, the value returned is used as the result of the operation `a < b`. This call is equivalent to `b.__gt__(a)`. If this call returns `NotImplemented` since `__gt__` method of class `B` does not know how to compare with an object of class `A`, then interpreter will raise an error.

In the case of `==` and `!=`, instead of raising `TypeError`, the interpreter will use the default implementation as the fallback and will return that result.

Operation	Forward call	Reflective call	Fallback
<code>a == b</code>	<code>a.__eq__(b)</code>	<code>b.__eq__(a)</code>	Return <code>id(a) == id(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>	<code>b.__ne__(a)</code>	Return <code>not(a==b)</code>
<code>a < b</code>	<code>a.__lt__(b)</code>	<code>b.__gt__(a)</code>	Raise <code>TypeError</code>
<code>a > b</code>	<code>a.__gt__(b)</code>	<code>b.__lt__(a)</code>	Raise <code>TypeError</code>
<code>a <= b</code>	<code>a.__le__(b)</code>	<code>b.__ge__(a)</code>	Raise <code>TypeError</code>
<code>a >= b</code>	<code>a.__ge__(b)</code>	<code>b.__le__(a)</code>	Raise <code>TypeError</code>

Table 15.5: Forward, reflective calls and fallback for comparison operations

So, when we do not know how to compare with another type, we should return `NotImplemented`. When `NotImplemented` is returned, the interpreter tries to call the reflection method with the arguments flipped. If we do not return `NotImplemented`, then the interpreter will not call the reflection method even though the reflection method is capable of performing the operation.

15.6 String representation of an instance object

The magic methods `__str__` and `__repr__` are used for converting an instance object into a string. The method `__str__` is invoked when an instance object is converted to a string by calling the `str` built-in function. It is also invoked when an instance object is printed using the `print` function because `print` implicitly calls the `str` built-in function. The method `__repr__` is invoked by the `repr` built-in function, and it is also used to display the object in the interactive terminal. If `__str__` is not defined, then this method is invoked for `str(obj)` and `print(obj)` also.

Both `__str__` and `__repr__` methods return a string representation of the instance, but `__str__` is generally used for the end user of the class; it returns a human-readable and user-friendly string representation of the object. The string returned by `__repr__` is a descriptive and unambiguous string representation of the object. It returns a Python-interpretable text that can be used by programmers for debugging. This text is generally a valid Python expression from which you can re-create the instance object using the `eval` function.

Let us see what happens when we print an instance object of our `Fraction` class.

```
>>> f = Fraction(2, 3)
>>> print(f)
<__main__.Fraction object at 0x000001C9DB23B100>
```

We get a string containing the class name and the object id. The interactive echo and `str` function will also give the same string.

```
>>> f
<__main__.Fraction object at 0x000001C9DB23B100>
>>> str(f)
'<__main__.Fraction object at 0x000001C9DB23B100>'
```

This is the way the interpreter prints an object by default. If we want to print the object in some other way then we have to change this default string representation by defining the `__str__` and `__repr__` methods.

If we want the output for the end user, we would like the data inside the object to be printed in some format. In our `Fraction` class, we had to call the `show` method to display a `Fraction` object in a user-friendly form. Now, let us define the `__str__` method.

```
def __str__(self):
    return f'{self.nr}/{self.dr}'
```

This method returns the string that we were printing in the `show` method. This `__str__` method will automatically be called by `str` and `print` functions.

```
>>> f = Fraction(2, 3)
>>> print(f)
2/3
>>str(f)
'2/3'
```

However, the interactive echo still gives the same output.

```
>>> f
'<__main__.Fraction object at 0x000001C9DB23B100>'
```

To change this, we have to define the `__repr__` method.

```
def __repr__(self):
```

```
    return f'Fraction({self.nr},{self.dr})'
```

The string that is returned is the source code required to instantiate the object. That is why we have the initializer of the class.

```
>>> f
Fraction(2,3)
>>> repr(f)
'Fraction(2,3)'
```

This string, when passed to the `eval` function, will create an equivalent object, and so, for most objects, `eval(repr(obj)) == obj` will be `True` provided we have an appropriate `__eq__` method defined.

```
>>> x = eval(repr(f))
>>> print(x)
2/3
>>> eval(repr(f)) == f
True
```

Containers like lists and dictionaries use `__repr__` for string representation of the contained objects.

```
>>> f1 = Fraction(3,4)
>>> f2 = Fraction(4,5)
>>> f3 = Fraction(1,5)
>>> L = [f1, f2, f3]
>>> L
[Fraction(3,4), Fraction(4,5), Fraction(1,5)]
>>> print(L)
[Fraction(3,4), Fraction(4,5), Fraction(1,5)]
>>> str(L)
'[Fraction(3,4), Fraction(4,5), Fraction(1,5)]'
```

15.7 Construction and destruction of objects

When we instantiate a class, two magic methods are called. First `__new__` is called, it creates the object and returns it, and then `__init__` is called to initialize the newly created object. Most of the classes do not need to define `__new__`, the built-in implementation works in most of the cases. In rare cases, if you want to control the creation process of the instance object, you can define the `__new__` method. The `__init__` method is defined in most of the classes for initialization purposes as we have already seen. The magic method invoked at the time of destruction of an object is `__del__`. Let us learn more about this method, although this method is also not very commonly used.

We know that the Python interpreter performs garbage collection to free up memory space, which means that it automatically destroys objects that are no longer in use. Each object has a reference count, which denotes the number of times the object has been referenced. When the reference count of an object reaches zero, the interpreter removes it automatically, and the memory occupied by the object is freed. This garbage collector works during the program execution and makes sure that there are no unused objects taking up space.

Another thing that we have seen earlier is that the `del` statement does not delete the object, it removes the reference and hence decreases the reference count of the object by one. For example, if names `x` and `y` are referring to the same object, then writing `del x` will not remove the object. It will only remove the name `x` from the scope and decrement the reference count of the object. When the name `y` will stop referring to the object (when it is reassigned, or removed using `del` statement or when it goes out of scope) the reference count of the object will drop to 0 and the interpreter will garbage collect it. Deleting names using the `del` statement is very rare, mostly the names go out of scope and when an object does not have any name referring to it, it is garbage collected.

The magic method `__del__` is automatically invoked when an object is destroyed by the garbage collector and this generally happens when the object's reference count becomes zero. In the following class, we have

defined a `__del__` method that prints a message so that we know when it is getting invoked.

```
>>> class MyClass:
    def __del__(self):
        print('Destroying')

>>> a = MyClass()

>>> b = a

>>> c = a

>>> del a

>>> del b

>>> del c
```

Destroying

We created a `MyClass` object and made the name `a` refer to it. Then we made the names `b` and `c` also refer to the same object. The `__del__` method was not executed when the names `a` and `b` were deleted, it was executed when the name `c` was deleted because then the reference count of the object dropped to zero.

If you want some clean-up actions to be performed when the object is being destroyed, you can define a `__del__` method in your class. This method can be used to free any non-memory resources used by an instance object, for example it can be used to close files, network connections or free other system resources. However, there is no guarantee that `__del__` will be invoked, sometimes it is not invoked even when the program terminates. This can happen if the object's reference count is not zero and the program terminates due to some reason. It is also not always predictable as to when an instance object will be garbage collected. This is why it is not advisable to close files or other connections in this method, as they might never be closed. These things are better handled using the `try...finally` block and context managers which are explained later in this book. In some other languages, destructors are common since they are used to free memory resources. But Python has a garbage collector which automatically reclaims

the memory space and hence there is no need to write any memory reclaiming code in our `__del__` method.

15.8 Making instance objects callable

The method `__call__` is used to overload the calling syntax. If this method is defined in a class, then the instance objects of that class become callable objects, we can call them like a function. This method is automatically invoked when an instance object is called, which means that `obj(arg1, arg2, .....)` is equivalent to `obj.__call__(arg1, arg2, .....)`. The arguments that are sent to the object while calling are sent to this method.

```
class MyClass:
    def __init__(self, data):
        self.data = data
    def __call__(self, value):
        return self.data + value

obj = MyClass(5)
x = obj(2)
print(x)
```

We have a class that defines the `__call__` method, and `obj` is an instance object of this class. We can call this instance object like a function, and behind the scene, interpreter will call the `__call__` method. Since the `__call__` method has one parameter apart from `self`, we can send one argument while calling the instance object. The call `obj(2)` is equivalent to the call `obj.__call__(2)`.

So, `__call__` enables programmers to write classes whose objects behave like functions. These objects when called can accept any type of arguments that functions can accept. We can also send these objects as arguments in places where a function object is accepted.

We can define the `__call__` method when we want our instance objects to behave like functions. It can be useful in cases when we need to retain state information between calls.

15.9 Overloading type conversion functions

Python provides many type conversion functions that can be used to convert one type to another. We have used them in previous chapters for performing conversion between different built-in types. Here are some examples:

```
>>> x = 10.5
>>> int(x)
10
>>> str(x)
'10.5'
>>> bool(x)
True
```

We have seen that `str(obj)` invokes the `__str__` magic method. Similar to `__str__`, there are methods like `__int__`, `__float__`, `__bool__` that will be invoked when `int()`, `float()`, `bool()` functions are called with an object of a user defined class.

<code>int(a)</code>	<code>a.__int__()</code>
<code>float(a)</code>	<code>a.__float__()</code>
<code>bytes(a)</code>	<code>a.__bytes__()</code>
<code>complex(a)</code>	<code>a.__complex__()</code>
<code>bool(a)</code>	<code>a.__bool__()</code>
<code>str(a)</code>	<code>a.__str__()</code>

Table 15.6: Magic methods or type conversion

You can define these magic methods if you want to convert your instance objects to one of these built-in types. We can add the following type

conversion functions to our Fraction class:

```
def __int__(self):
    return self.nr // self.dr

def __float__(self):
    return self.nr / self.dr

def __bool__(self):
    return True if self.nr != 0 else False
```

Now we can use the conversion functions `int()`, `float()` and `bool()` with our Fraction objects.

```
>>> f1 = Fraction(19, 2)
>>> print(int(f1), float(f1), bool(f1))
9 9.5 True
>>> f2 = Fraction(0, 3)
>>> print(int(f2), float(f2), bool(f2))
0 0.0 False
```

The magic method `__bool__` is called by the `bool()` function and is also called in Boolean contexts such as in `if obj:` or `while obj:`. If `__bool__` is not defined, then the interpreter will look for the `__len__` method, and if it returns 0, object is considered False. If both these methods are not defined, then all instance objects of user-defined classes are considered True.

15.10 List of magic methods

We discussed the most commonly used magic methods. There are many more magic methods available. We cannot cover all of them here; for more information, you can consult the official Python documentation. The following tables show some more magic methods:

\

<code>+a</code>	<code>a.__pos__()</code>
<code>-a</code>	<code>a.__neg__()</code>
<code>~a</code>	<code>a.__invert__()</code>

Table 15.7: Magic methods for unary operators

<code>abs(a)</code>	<code>a.__abs__()</code>
<code>reversed(a)</code>	<code>a.__reversed__()</code>

Table 15.8: Magic methods for built-in functions

<code>round(a)</code>	<code>a.__round__()</code>
<code>math.trunc(a)</code>	<code>a.__trunc__()</code>
<code>math.floor(a)</code>	<code>a.__floor__()</code>
<code>math.ceil(a)</code>	<code>a.__ceil__()</code>

Table 15.9 Magic methods for math functions

Instance object creation	<code>__new__()</code>
Instance object initialization	<code>__init__(arg1, arg2, ...)</code>
Instance object deletion	<code>__del__()</code>

Table 15.10 Magic methods for instance creation and destruction

<code>x in a</code>	<code>a.__contains__(x)</code>
<code>a[k]</code>	<code>a.__getitem__(k)</code>
<code>a[k] = x</code>	<code>a.__setitem__(k, x)</code>
<code>del a[k]</code>	<code>a.__delitem__(k)</code>
<code>len(a)</code>	<code>a.__len__()</code>

Table 15.11 Magic methods for emulating collections

<code>iter(a)</code>	<code>a.__iter__()</code>
<code>next(a)</code>	<code>a.__next__()</code>
<code>reversed(a)</code>	<code>a.__reversed__()</code>

Table 15.12 Magic methods for iteration

Entering with code block	<code>__enter__</code>
Exiting with code block	<code>__exit__</code>

Table 15.13 *Magic methods for context management*

The last two categories (Iteration and Context management) will be discussed in separate chapters.

Exercise

1. In the following class, write code for the methods `__eq__`, `__lt__`, `__le__`.

```
class Time:
    def __init__(self, h, m, s):
        self._h = h
        self._m = m
        self._s = h

    #Read-only field accessors
    @property
    def hours(self):
        return self._h

    @property
    def minutes(self):
        return self._m

    @property
    def seconds(self):
        return self._s

    def _cmp(time1, time2):
        if time1.hours < time2.hours:
```

```

        return 1
    if time1.hours > time2.hours:
        return -1
    if time1.minutes < time2.minutes:
        return 1
    if time1.minutes > time2.minutes:
        return -1
    if time1.seconds < time2.seconds:
        return 1
    if time1.seconds > time2.seconds:
        return -1
    return 0

t1 = Time(13, 10, 5)
t2 = Time(5, 15, 30)
t3 = Time(5, 15, 30)
print(t1 < t2)
print(t1 > t2)
print(t1 == t2)
print(t2 == t3)

```

2. Implement `__add__` and `__radd__` methods for the following class `Length`.

```

class Length:
    def __init__(self, feet, inches):
        self.feet = feet
        self.inches = inches
    def __str__(self):

```

```

        return f'{self.feet} feet
{self.inches} inches'

    def add_length(self,L):
        f = self.feet + L.feet
        i = self.inches + L.inches
        if i >= 12:
            i = i - 12
        f += 1
        return Length(f, i)

    def add_inches(self,inches):
        f = self.feet + inches // 12
        i = self.inches + inches % 12
        if i >= 12:
            i = i - 12
        f += 1
        return Length(f, i)

length1 = Length(2,10)
length2 = Length(3,5)
print(length1 + length2)
print(length1 + 2)
print(length1 + 20)
print(20 + length1)

```

3. In the following class define a `__str__` method. Define a `__bool__` method so that any `BankAccount` object can be used in a Boolean context. A `BankAccount` object should be considered True if the balance is non-zero, otherwise it should be considered False.

```

class BankAccount:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance
    def display(self):
        print(self.name, self.balance)
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
a1 = BankAccount('Mike', 200)
a2 = BankAccount('Tom')

```

4. Write an appropriate method in the following Person class so that the code given below works.

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f'{self.name} {self.age}'
    def greet(self):
        if self.age < 80:
            print('Hi, how are you doing?')
        else:
            print('Hello, how do you do?')
p1 = Person('Tom', 20)

```

```
p2 = Person('Bob', 15)
p3 = Person('Yug', 32)
p4 = Person('Sam', 80)
p5 = Person('Jim', 19)
p6 = Person('Kim', 32)
guests = [p1, p2, p3, p4, p5, p6]
for guest in sorted(guests):
    print(guest)
youngest = min(guests)
oldest = max(guests)
print('Youngest guest is', youngest)
print('Oldest guest is', oldest)
```

5. What will be the output of the following code?

```
class VideoCourse:
    def __init__(self, title, instructor,
duration):
        self.title = title
        self.instructor = instructor
        self.duration = duration
    def __len__(self):
        return self.duration
course1 = VideoCourse('Learn Piano', 'Jack',
10)
course2 = VideoCourse('Learn Python', 'John',
15)
print(len(course1), len(course2))
```

Project : Date Class

In this project, we will make a `Date` class that can be used in different programs. First, let us see how our `Date` type should work and what operations it should support.

We should be able to create a new instance object of `Date` type from values of day, month, and year. For example, the following `Date` object represents 9th November 1977.

```
d = Date(9, 11, 1977)
```

Calls like `Date(5, 13, 1973)` and `Date(32, 12, 1987)` should give error as 13 is not a valid month value and 32 is not a valid value for day. Similarly `Date(29, 2, 2001)` should also give error because 2001 is not a leap year.

We should be able to create a `Date` object from a string that is in dd-mm-yyyy format (eg. '09-08-1973'), or from another `Date` object. We should also be able to make a `Date` object from today's date.

The `Date` instance object should have three read-only attributes using which we can access the day, month, and year of the date.

When we print a `Date` object, it should be printed in the following format:

```
9/11/1977
```

The class should have methods that could tell us the day of the week, next Sunday, and next weekday from a given date. It should also have methods for adding or subtracting days, months, and years from a date. The class should also have a method for subtracting a date from another.

We should be able to compare two `Date` objects for equality. If the day, month, and year are equal in both objects, then two objects are considered equal. The operators less than and greater than should also work; for example, `d1 < d2` should return `True` if date `d1` falls before `d2`.

The `+` operator should be used to add a number of days; for example, the expression `d1 + 4` should give a new instance object, in which 4 days are added to the date represented by object `d1`. The reverse (`4 + d1`) should

also work. The minus operator should be used to subtract the number of days. The update assignment operators `+=` and `-=` should also work.

The minus operator, when used between two `Date` objects (e.g., `d2-d1`), should give the number of days between those dates.

So, we have seen the functionalities that our `Date` type should have. Now let us start implementing it:

```
-----date.py-----
```

```
class Date:
    def __init__(self, d, m, y):
        self._d = d
        self._m = m
        self._y = y
        if not self._is_valid():
            raise ValueError('This date is not
valid')
```

This `__init__` method has three parameters after `self`, and the values of these parameters are assigned to the three instance variables `_d`, `_m` and `_y`. These instance variables denote the day, month and year. After assigning values to the 3 variables, we will check the validity of the date. For that we will make another method named `_is_valid`. This method will return `True` if the date represented by the instance object is valid, otherwise it returns `False`.

Inside this method, we will check the values of the three instance variables. If the year is less than 1500 or more than 2500, then the `False` is returned; if the month is less than 1 or greater than 12, then also `False` is returned. We have to check the number of days also but the number of days is not the same for each month. The valid values for day will depend on the month. For example, 31 is a valid value for the day if the month is March, but it is invalid if the month is September. The validity of the day depends on the year also; for example, 29 is a valid value of day for February 2000, but it is not a valid value of day for February 1999. So, we have 1 as the lower limit value for the day, but the upper limit is not fixed; it depends on the value of

the month and year. Thus, for the upper limit, we will call a function `_days_in_month`. This function will return the days in a given month and year.

```
def _is_valid(self):
    if self._y < 1500 or self._y > 2500:
        return False
    if self._m < 1 or self._m > 12:
        return False
    if self._d < 1 or self._d >
    _days_in_month(self._m, self._y):
        return False
    return True
```

If none of the three if conditions is True, then it means that the date is valid and in that case True is returned from the method `_is_valid`.

Now, let us write code for the function `_days_in_month`. Note that this is a function, it is not a method of the class.

If the month is January, March, May, July, August, October or December then 31 is returned. If month is April, June, September or November then 30 is returned. If month is February, then 29 or 28 is returned depending on whether the year is leap or not.

```
def _days_in_month(month, year):
    if month in {1, 3, 5, 7, 8, 10, 12}:
        return 31
    if month in {4, 6, 9, 11}:
        return 30
    if month == 2:
        if is_leap(year):
            return 29
```

```
        else:
            return 28
```

Here is the definition of the `is_leap` function:

```
def is_leap(year):
    return year%4 == 0 and year%100 != 0 or
year%400 == 0
```

This function takes in a year as argument and returns True if the year is leap, otherwise it returns False. A non-centennial year is leap if it is divisible by 4 and a centennial year is leap if it is divisible by 400.

Now, let us write the `__str__` and `__repr__` methods for proper display of the object.

```
def __str__(self):
    return f'{self._d}/{self._m}/{self._y}'
def __repr__(self):
    return f'Date({self._d}, {self._m}, {self._y})'
```

We can test the code that we have written till now. We can either import the `Date` class on the interactive shell or execute the `date.py` file and then test on the prompt.

```
>>> from date import Date
>>> d = Date(5, 13, 1987)
ValueError: This date is not valid
>>> d = Date(32, 12, 1987)
ValueError: This date is not valid
>>> d = Date(29, 2, 2001)
ValueError: This date is not valid
>>> d1 = Date(29, 2, 2000)
>>> d2 = Date(15, 5, 2005)
```

```
>>> d1
Date(29, 2, 2000)
>>> d2
Date(15, 5, 2005)
>>> print(d1)
29/2/2000
>>> print(d2)
15/5/2005
```

Now, let us write a class method to create a `Date` instance object from a string.

```
@classmethod
def from_str(cls, s):
    if len(s) != 10 or s[2] != '-' or s[5] != '-' :
        raise ValueError('String not in correct
format\nCorrect format is
"dd-mm-yyyy"')
    d, m, y = s.split('-')
    return cls(int(d), int(m), int(y))
```

Here, first we are checking whether the argument string is in the correct format, if it is not then we raise a `ValueError`. If it is in the format `dd-mm-yyyy`, then we split the string to get three values and then we create a new `Date` instance object from these values and return it.

The next class method, `from_date`, creates a `Date` instance object from another `Date` instance object. You know that you can't simply write `d3 = d1` because then `d3` will be a reference to object `d1`.

```
@classmethod
def from_date(cls, obj):
    return cls(obj._d, obj._m, obj._y)
```

This method is simple, we are using the values of the instance variables of the current object to create a new object.

Here is another class method that enables us to create a `Date` instance object from the current date:

```
@classmethod
def today(cls):
    from time import ctime
    s = ctime()
    s = s.replace(' ', ' ') #if day is single
digit
    _,m,d,_,y = s.split(' ')
    months = ('','Jan', 'Feb', 'Mar', 'Apr',
'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec')
    return Date(int(d), months.index(m), int(y))
)
```

We have called the function `ctime` from `time` module and stored the return value in the string `s`. This string will be in this format - 'Tue Jul 30 11:40:47 2019'. We need 3 values from this string- month, day and year so we have split it on space and ignored the first and fourth parts using underscore. Thus, the month is saved in variable `m`, the day in variable `d`, and the year in variable `y`. We get the month's name, but to create our instance object, we need the numeral value of the month. So, we have defined a tuple of month names where the first element is an empty string. The index of 'Jan' is 1, the index of 'Feb' is 2 and so on. After this, we have created and returned a `Date` instance object.

Before proceeding further, let us test the 3 methods that we have made:

```
>>> birth_date = '9-8-1973'
>>> d = Date.from_str(birth_date)
ValueError: String not in correct format
```

Correct format is "dd-mm-yyyy"

```
>>> birth_date = '09-08-1973'
>>> d = Date.from_str(birth_date)
>>> print(d)
9/8/1973
>>> d1 = Date.from_str('29-02-2005')
ValueError: This date is not valid
This date is not valid since 2005 is not a leap year.
>>> d1 = Date.from_str('29-02-2008')
>>> print(d1)
29/2/2008
>>> d2 = Date.from_date(d1)
>>> print(d2)
29/2/2008
>>> d3 = Date.today()
>>> print(d3)
26/10/2023
```

Next, we want to create three read only attributes to provide access to year, month and day from the `Date` instance object. For this we will create properties:

```
@property
def year(self):
    return self._y
@property
def month(self):
    return self._m
```

```
@property
def day(self):
    return self._d
```

So, we have these 3 methods with the property decorator and they simply return the value of the instance variables. Let us test them:

```
>>> d = Date(9, 11, 1977)
```

```
>>> d.day
```

```
9
```

```
>>> d.month
```

```
11
```

```
>>> d.year
```

```
1977
```

```
>>> d.month = 6
```

```
AttributeError: property 'month' of 'Date' object
has no setter
```

If we try to assign to any of these attributes, we will get an error.

The next method that we will make will add years to a `Date` object. This method will not in any way change the object on which it is called, it will return a new `Date` instance object.

```
def add_years(self, iyear):
    d = self._d
    m = self._m
    y = self._y + iyear
    if d==29 and m==2 and not is_leap(y):
        d = 28
    return Date(d, m, y)
```

For the new object, day and month will be the same, only the year will change. At the end we are creating and returning an instance object with the values `d`, `m` and `y`. We need to put a small check before creating this object.

Suppose the date represented by `self` object is 29th February 2000 and we need to add three years. The values for `d`, `m` and `y` will be 29, 2 and 2003.

The date represented by this object will be 29th February 2003, but it is not a valid date because 2003 is not a leap year.

So, we need to put a check: if the day is 29 and month is 2 and the year that we get after adding is not a leap year then we make the day 28.

This method for adding years is quite simple. The method for subtracting years is very similar to this one.

```
def sub_years(self, dyear):
    d = self._d
    m = self._m
    y = self._y - dyear
    if d==29 and m==2 and not is_leap(y):
        d = 28
    return Date(d, m, y)
```

Now we will make methods to add and subtract months. Before writing the code for this, first we need to understand the procedure.

Suppose we have the date 5 - 3 - 1980 and we have to add 4 months it. We can simply add 4 to the month value and the new date is 5 - 7 - 1980.

Now suppose we need to add 41 months to the date 5 - 3 - 1980. If we simply add 41 to the month value then we will get 44 which is not a valid month value. We will have to break 41 into months and years and then add the years to year value and months to month value.

$41//12 = 3$ years, $41\%12 = 5$ months

So, the year becomes 1983 (1980+3) and month becomes 8 (3+5) and the new date is 5 - 8 - 1983.

Now, suppose we have to add 34 months to the date 5 - 3 - 1980. On breaking 34, we find that we have to add 2 years and 10 months. The date becomes 5 - 13 - 1982, but the month value 13 is invalid. So, from these 13 months, we will take out 12 months and add 1 year to the year value. Thus, we are left with 1 month, and the year becomes 1983. Thus, the resultant date is 5 - 1 - 1983.

Now, let us add 3 months to the date 31 - 3 - 1980. The date becomes 31 - 6 - 1980, but it is an invalid date because June does not have 31 days. We have to make the value of the day equal to 30, which is the last day of June.

We need to keep all these things in mind while writing the `add_months` method.

```
def add_months(self, imonth):  
    d = self._d  
    m = self._m + (imonth % 12)  
    y = self._y + (imonth // 12)  
    if m > 12:  
        m = m - 12  
        y = y + 1  
    dm = _days_in_month(m, y)  
    if d > dm:  
        d = dm  
    return Date(d, m, y)
```

First, we add the value `imonth % 12` to month value and value `imonth // 12` to year value. The value for day remains the same. If the value of month that we get after adding is more than 12 then we add 1 to the year and subtract 12 from months.

Then, we find out the number of days in the month of the year by using the method `_days_in_month(m, y)` that we have seen earlier. If the value of day is more than the number of days in the month then we change the day

value to `dm`, where `dm` is the last day of the month. At last, we create and return a `Date` object.

Similarly, we can write the method for subtracting months:

```
def sub_months(self, dmonth):
    d = self._d
    m = self._m - (dmonth % 12)
    y = self._y - (dmonth // 12)
    if m <= 0:
        m = m + 12
        y = y - 1
    dm = _days_in_month(m, y)
    if d > dm:
        d = dm
    return Date(d, m, y)
```

First, we are subtracting the values from month and year. After subtracting, the value of `m` can become negative.

So, in that case we will add 12 to months and subtract a year. The last check is the same what we did in `add_months`. Now, let us test the methods `add_years`, `sub_years`, `add_months`, `sub_months`.

```
>>> d = Date(2, 5, 2002)
```

```
>>> d.add_years(10)
```

```
Date(2, 5, 2012)
```

```
>>> d.sub_years(10)
```

```
Date(2, 5, 1992)
```

```
>>> d.add_months(30)
```

```
Date(2, 11, 2004)
```

```
>>> d.add_months(34)
```

```

Date(2, 3, 2005)
>>> d.sub_months(34)
Date(2, 7, 1999)
>>> d1 = Date(29, 2, 2000)
>>> d1.add_years(2)
Date(28, 2, 2002)
>>> d2 = Date(31, 3, 2000)
>>> d2.add_months(6)
Date(30, 9, 2000)

```

Now, we will write the methods for adding and subtracting days from a date. Adding or subtracting days is not as simple as adding or subtracting months and years so let us first understand how we will do it theoretically. We will use the concept of Julian day in these methods, so let us see what a Julian day is. Julian day is the day of the year on which the date falls. For example, Julian day of 1 Jan is 1, 10th Jan is 10, 1st Feb is 32, 5th March 64, 9th July is 190 and for the last day of the year 31st Dec it is 365. These are the Julian days for a non-leap year. For a leap year, each Julian day from 1st March will be 1 more than what it is in a non-leap year. This is because of an extra day in February of a leap year.

Here is the method for getting Julian day for a Date instance object:

```

def _julian(self):
    j = self._d
    for i in range(1, self._m):
        j += _days_in_month(i, self._y)
    return j

```

Let us see how this method works, suppose the date is 9th July 1948. Initially value of `j` will be 9. In the for loop, `i` will take values from 1 to 6, since value of `self._m` is 7. Therefore, the days of all the months from 1 to 6 will be added to `j`. Value of `j` will be 9+31+29+31+30+31+30=191. Finally,

the value of `j` is returned. So, this method will return the Julian day for the `Date` instance object on which it is called.

If we are given a Julian day and a year, we can create a date from it. Here is the function that accepts a Julian day and a year and returns a `Date` instance object:

```
def _date_from_julian(j, year):
    for month in range(1, 13):
        dm = _days_in_month(month, year)
        if j <= dm:
            break
        j -= dm
    return Date(j, month, year)
```

The variable `month` in the `for` loop takes values from 1 to 12. We get the number of days in month `m` using the function `_days_in_month`. If `j` is less than the number of days in the month, then we break out of the loop; otherwise, we subtract the days from `j`. At the end whatever value will be left in variable `j` will be the value for day; month will be equal to value of variable `month` and year will remain the same. At the end we create a `Date` instance object with the values `j`, `month` and `year`.

For example, suppose the Julian day is 200 and year is 1980. Initially month is 1, so we subtract 31 from the Julian day, then month is 2 so we subtract 29 since this year is leap and so on.

<code>month = 1</code>	<code>200 - 31 = 169</code>
<code>month = 2</code>	<code>169 - 29 = 140</code>
<code>month = 3</code>	<code>140 - 31 = 109</code>
<code>month = 4</code>	<code>109 - 30 = 79</code>
<code>month = 5</code>	<code>79 - 31 = 48</code>
<code>month = 6</code>	<code>48 - 30 = 18</code>
<code>month = 7</code>	

Figure 15.2: Finding date for year 1980 and Julain day 200

When the month is 7, we need to subtract 31 but the value left is 18 which is less than 31. So, the break statement will be executed, and the loop will terminate. The date that we get is 18-7-1980. We need the year also to get the date because depending on the year only we will subtract 29 or 28 when the month is February.

We have seen what is a Julian day, how to get a Julian day from a date, and how to get a date when Julian day and year are given. Now let us see how to add days to a Date.

Suppose we have to add 50 days to the date 14 - 6 - 1950. The Julian day for this date is 165. If we subtract 165 from 365 we get 200 which is the number of days left in this year after this date. We have to add 50 days which is less than 200. It means that on adding 50 to this date we will get a date which will fall in the same year. So, in this case we will simply add 50 to 165, and we get the Julian day of the new date, year will be the same. From the Julian day 215 and year 1950 we can get the date which is 3 - 8 - 1950.

14 - 6 - 1950	Add 50 days	$j2 = 165 + 50 = 215$	3 - 8 - 1950
$j1 = 165$		$y2 = 1950$	
$n = 365 - 165 = 200$			

Figure 15.3: Adding 50 days to date 14-6-1950

Now suppose we want to add 250 days to the date 14 - 6 - 1950. Since 250 is greater than 200, the date that we will get after adding 250 days will fall in the next year. We go to the next year by subtracting the remaining days of 1950. The date that we will get will be in 1951 and its Julian day will be 50 (250-200). We have obtained the Julian day and year, and these two can be used to construct the date.

14 - 6 - 1950	Add 250 days	$j2 = 250 - 200 = 50$	19 - 2 - 1951
$j1 = 165$	$y2 = 1951$		
$n = 365 - 165 = 200$			

Figure 15.4: Adding 200 days to date 14-6-1950

Now suppose we want to add 1800 days to the date 14 - 6 - 1950. The value 1800 is much more than 200 (Julian day of the date). So, the date that we

will get will not be in year 1950, it will not be in the next year also, it will be sometime after 4 to 5 years. Let us find it out.

We start with $j_2 = 1800$ and year 1950. First, we subtract 200 from 1800 and come to 1951. Then we subtract 365 days of 1951 and come to 1952. Then we subtract 366 days of 1952 and come to 1953, then we subtract 365 days of 1953 and come to 1954, then we subtract 365 days of 1954 and come to 1955. Now the number left is 139 which is less than the number of days in 1955. It means that the date lies in 1955 and 139 is the Julian day of that date. We can get the date from this Julian day.

14 - 6 - 1950	Add 1800 days	$j_2 = 1800$	19 - 5 - 1955
$j_1 = 165$		$y_2 = 1950$	
$n = 365 - 165 = 200$		$j_2 = 1800 - 200 = 1600$	
		$y_2 = 1951$	
		$j_2 = 1600 - 365 = 1235$	
		$y_2 = 1952$	
		$j_2 = 1235 - 366 = 869$	
		$y_2 = 1953$	
		$j_2 = 869 - 365 = 504$	
		$y_2 = 1954$	
		$j_2 = 504 - 365 = 139$	
		$y_2 = 1955$	

Figure 15.5: Adding 1800 days to date 14-6-1950

Here is the method for adding days to a Date object:

```
def add_days(self, days):
    j1 = self._julian()
    n = 366-j1 if is_leap(self._y) else 365-j1
    if days <= n:
        j2 = j1 + days
        y2 = self._y
    else:
        days -= n
        y2 = self._y + 1
```

```

        k = 366 if is_leap(y2) else 365
    while days >= k:
        if is_leap(y2):
            days -= 366
        else:
            days -= 365
        y2 += 1
        k = 366 if is_leap(y2) else 365
    j2 = days
    return _date_from_julian(j2, y2)

```

First, we get the Julian day of the date and store it in `j1`. Then we find `n` which is the number of remaining days in the year. We have to take care of the leap year case here.

If the number of days to be added is less than or equal to `n`, it means that the new date will fall in the same year, so `j2` is made equal to `j1 + days` and `year` is equal to `self._y`.

In the `else` part we execute a while loop to repeatedly subtract the days of years and advance the years. At the end we create a `Date` instance object with `j2` as Julian day and `y2` as year and return from this method.

Using a similar logic, we can write the method for subtracting days:

```

def sub_days(self, days):
    j1 = self._julian()
    if days < j1:
        j2 = j1-days
        y2 = self._y
    else:
        days = days - j1    # Now subtract days
    from 1st Jan y1

```

```

        y2 = self._y - 1
        k = 366 if is_leap(y2) else 365
        while days >= k:
            if is_leap(y2):
                days -= 366
            else:
                days -= 365
            y2 -= 1
            k = 366 if is_leap(y2) else 365
        j2 = 366-days if is_leap(y2) else 365-
days
        return _date_from_julian(j2,y2)

```

This method returns a new instance object by subtracting days from the `self` object. Here we use a loop to decrement years.

Now, we will write two more methods `diff_ymd` and `diff_days` to get the difference of two dates. In both these methods, the second date should fall before the first date, if it is not so then a `ValueError` will be raised.

```

def diff_ymd(self, date2):
    if date2 > self :
        raise ValueError('Second date should fall
before first date')
    .....

def diff_days(self,date2):
    if date2 > self :
        raise ValueError('Second date should fall
before first date')
    .....

```

The method `diff_days` will give the number of days between the two dates and `diff_ymd` will give the difference between the dates in years, months and days. `diff_days` will return an integer and `diff_ymd` will return a 3-element tuple.

First let us understand the logic of finding the difference between two dates in days, months and years.

Suppose we have to find the difference between the dates 20 - 8 - 1990 and 10 - 5 - 1986. We can simply subtract year from year, month from month and day from day. The difference is 4 years 3 months and 10 days.

Now suppose we need to find the difference between the dates 20 - 2 - 1990 and 10 - 5 - 1986. If we follow the same strategy then we will get a minus value for month. To avoid this, we will add 12 months to the month value of the first date so it becomes 14 and from the year we will subtract 1, so the year becomes 1989. Now we can subtract year from year, month from month and days from days and we get the difference as 3 years 9 months and 10 days.

Now suppose we need to find the difference between the dates 3 - 2 - 1990 and 10 - 5 - 1986. Here both day value and month value of first date are less than the date and month value of the second date. So, on direct subtraction both day and month will come out to be negative. We need to make the day and month values of first date greater than the second date. First, we will work on the day value. We will add 31 days of January to the days value and change the month from 2 to 1. Then we will add 12 to the month value and subtract 1 from the year. So now for the first date, day is 34, month is 13 and year is 1989. On subtraction we get the difference as 3 years 8 months 24 days.

If the month of the first date is January, then we will add 31 days of December and go to the previous year. The year value of first date will be always be equal to or more than year value of second date, because we will put a check for this in the code.

Here is the definition of the method that gives the difference in years, months and days:

```
def diff_ymd(self, date2):  
    if date2 > self:
```



```

        raise ValueError('Second date should
fall before first date')

    d1, m1, y1 = self._d, self._m, self._y
    d2, m2, y2 = date2._d, date2._m, date2._y
    if d1 < d2:
        if m1 != 1:
            d1 += _days_in_month(m1-1, y1)
            m1 = m1-1
        else:
            d1 += 31
            m1 = 12
            y1 = y1 -1
    if m1 < m2:
        m1 = m1 + 12
        y1 = y1 - 1
    return y1-y2, m1-m2, d1-d2

```

In the beginning we have put a check, we will see how to overload the `>` operator in a short while. `d1, m1` and `y1` are day, month and year values of `self` object and `d2, m2, y2` are day, month and year values of the `date2` object.

If `d1` is less than `d2` then we add days to `d1` and decrement `m1`. We will do this for all months except January. If month is January, we will add 31 to `d1`, make `m1` equal to 12 and go to previous year.

If `m1` is less than `m2`, then we add 12 to `m1` and decrement `y1`.

Thus, we have made sure that `y1, m1` and `d1` are not smaller than `y2, m2` and `d2` so we can subtract and return the three results separated by comma.

Before writing the code for `diff_days`, let us first understand the process of getting the difference of two dates in days.

Suppose we want the difference between the two dates 10 - 5 - 1986 and 20 - 8 - 1986. Both are in the same year so we can just subtract the Julian days of both dates and get the difference. The difference is $232 - 130 = 102$ days.

Now suppose we have to find the difference between the dates 10 - 5 - 1986 and 20 - 8 - 1991. When the dates are in different years, the number of days between the two dates can be found out by adding three components.

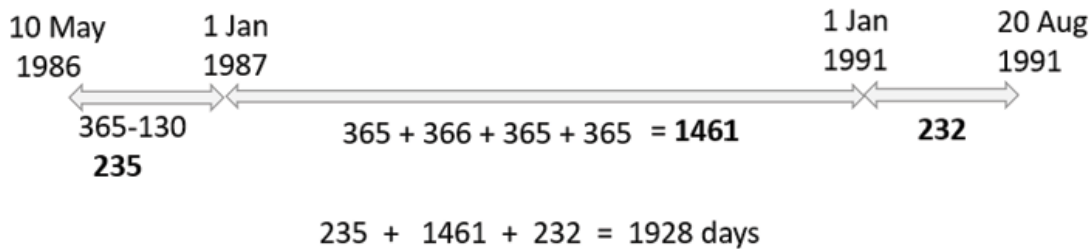


Figure 15.6: Finding difference between dates

From 10 May 1986 to 1 Jan 1987, we have 235 days. We get this number by subtracting the Julian day of 10 May 1986 from 365. Then we calculate the days between 1 Jan 1987 to 1 Jan 1991. We get these days by adding number of days in years 1987, 1988, 1989, and 1990. In the code, we will need to write a loop for this. The third component is the Julian day of the second date. Thus by adding the three numbers we get the number of days between the two dates.

Here is the code for the method `diff_days`. It will give the difference between two dates in days.

```
def diff_days(self, date2):
    if date2 > self:
        raise ValueError('Second date should
fall before first date')
    j1 = self.julian()
    j2 = date2.julian()
    if self._y == date2._y:
        return j1-j2
```

```

d = 0
for year in range(date2._y+1, self._y):
    if is_leap(year):
        d += 366
    else:
        d += 365
if is_leap(date2._y):
    return (366-j2) + d + j1
else:
    return (365-j2) + d + j1

```

In this method also, first we will put a check and make sure that `date2` falls before the `self Date` object. After this check we get the Julian day of the two dates. If both the dates are in the same year, we just return the difference of the two Julian days.

Otherwise, a for loop is executed and the sum of three components is returned. We cannot test the two methods that we have just written because we have not yet defined the `>` operator.

Now we will make our `Date` instance objects work with some operators. We will make a function `_cmp` that we will use inside the magic methods for comparison. This function takes two `Date` instance objects and returns 1 if the first date falls before the second date, -1 if the second date falls before the first date and it returns 0 if the dates are same. We can say that it returns 1 if `date1` is less than `date2`, -1 if `date2` is less than `date1` and 0 if they are equal.

```

def _cmp(date1, date2):
    if date1._y < date2._y:
        return 1
    if date1._y > date2._y:
        return -1

```

```

    if date1._m < date2._m:
        return 1
    if date1._m > date2._m:
        return -1
    if date1._d < date2._d:
        return 1
    if date1._d > date2._d:
        return -1
    return 0

```

The code for this function is simple, first we compare years and return 1 or -1. If years are same, then months are compared. If months are also same then days are compared. And if all three are same then 0 is returned. We can call this function inside the methods `__eq__`, `__lt__` and `__le__`.

```

    def __eq__(self, other):
        return True if _cmp(self, other) == 0 else
False
    def __lt__(self, other):
        return True if _cmp(self, other) == 1 else
False
    def __le__(self, other):
        return True if (_cmp(self, other) == 0 or
_cmp(self, other) == 1) else False

```

Next, we have the `__add__` and `__radd__` methods that can be used to add number of days to a date. These will be called in expressions like `d1 + 40` or `40 + d1` where `d1` is a `Date` instance object. In the `__add__` methods we have called the method `add_days`.

```

    def __add__(self, other):
        if isinstance(other, int):

```

```

        return self.add_days(other)
    else:
        return NotImplemented
def __radd__(self, other):
    return self.__add__(other)

```

In the `__sub__` method, we will call the `diff_days` to find the number of days between two dates, and the method `sub_days` to subtract some number of days. This magic method will be called in expressions like `d1 - d2` or `d1 - 20` where `d1` and `d2` are `Date` instance objects.

```

def __sub__(self, other):
    if isinstance(other, Date):
        return self.diff_days(other)
    elif isinstance(other, int):
        return self.sub_days(other)
    else:
        return NotImplemented

```

Next, we have to write the code for methods `day_of_week`, `next_sunday` and `next_weekday`.

The method `day_of_week` gives the weekday from the date. The day of week can be found out by using a simple formula that uses Julian day.

```

def day_of_week(self):
    weekday_name = ('Saturday', 'Sunday',
                    'Monday', 'Tuesday', 'Wednesday',
                    'Thursday', 'Friday')
    j = self.julian()
    f = (self._y-1)//4
    h = (self._y-1)//100
    fh = (self._y-1)//400

```

```
d = (self._y + j + f - h + fh) % 7
return weekday_name[d]
```

The method `next_sunday` returns a `Date` instance object that represents the next Sunday and the method `next_weekday` returns a `Date` instance object that represents the next weekday.

```
def next_sunday(self):
    day = self.day_of_week()
    weekday_name = ('Sunday', 'Monday',
                    'Tuesday', 'Wednesday', 'Thursday',
                    'Friday', 'Saturday')
    i = weekday_name.index(day)
    return self.add_days(7-i)
def next_weekday(self):
    day = self.day_of_week()
    if day == 'Friday':
        return self.add_days(3)
    elif day == 'Saturday':
        return self.add_days(2)
    return self.add_days(1)
```

You can type `dir(Date)` or `help(Date)` to see all the methods of this class.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Inheritance and Polymorphism

16

The example classes that we have seen are quite short, but the classes written for real-world applications would be complex, lengthy, and will contain a lot of code. It would take a considerable amount of time to develop and test a fully functional class. Sometimes, we may need to write a class that has most of the features of an existing class, along with some additional features. Writing such a class from scratch and testing it would be time-consuming, and it would be good if we could somehow use our existing class to create our new class. Python provides the feature of inheritance for this purpose. By using inheritance, we can create a new class based on an existing class. In our new class, we get all the features of the existing class, and can also add new features and also override (replace) them as needed. Thus, we can easily create new classes by using the tried and tested functionality of existing classes. This reduces time and effort and simplifies the task of writing a new class.

Inheritance is an important feature of object-oriented programming; it is basically a mechanism of creating a new class from an existing class. The new class is the extended and modified version of the existing class. The main advantage of inheritance is that it facilitates code reuse and reduces code duplication. Inheritance also simplifies the design of the program as it lets you represent the real-world problems in a natural and better way. This makes the program more readable and easier to manage.

Let us understand inheritance with the help of an example. Suppose we want to create `Employee` objects which should have the following data members and methods:

Employee
name age address phone salary office_address office_phone
greet() is_adult() contact_details() calculate_tax()

Figure 16.1: Employee class

We already have a `Person` class that has most of the functionality that we need for an `Employee` class.

Person
name age address phone
greet() is_adult() contact_details()

Figure 16.2: Person class

Instead of creating a brand-new `Employee` class from scratch, we can create our `Employee` class by inheriting from the `Person` class.

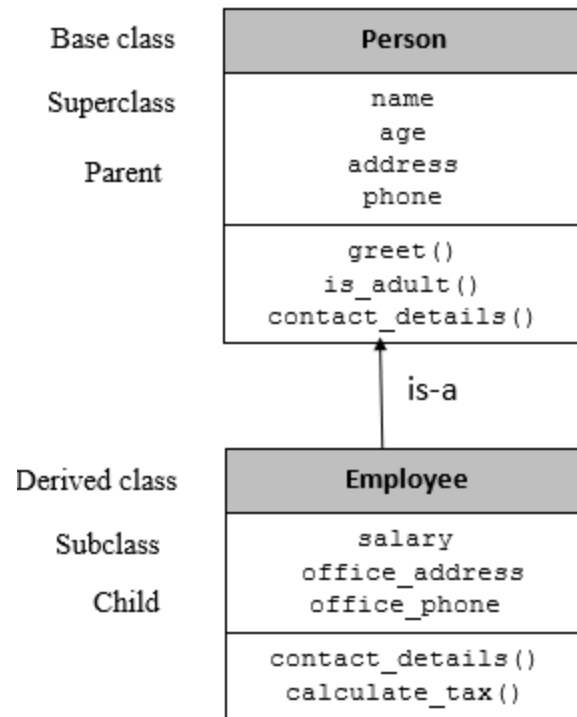


Figure 16.3: *Employee class inheriting from Person class*

The existing class is called the base class and the new class is called the derived class. When you inherit from a class, everything from that class becomes automatically available in the derived class. The derived class inherits members from the base class and also contains its own members. There is no need to copy everything from `Person` class to the `Employee` class. Due to inheritance, `Employee` class has access to everything from the `Person` class and the `Employee` class can have variables and methods of its own also.

So, when you derive a class, that class gets access to everything from the base class, it can add new variables and methods of its own and it can even change the way some methods of the base class work. For example, the `contact_details` method that is there in the `Person` class, will be inherited by the `Employee` class. If you want this method to work differently for `Employee` class, you can provide a separate code for it in the `Employee` class. Thus, a derived class can add its own version of a method which is called overriding.

Derived classes generally have some added functionality and provide more specific behaviour than the base class. Base class is also called the parent

class or super class and the derived class is also called child class or subclass.

In object-oriented terms, the relationship between the base class and derived class is called *is-a* relationship. Derived class is a type of base class, for example an Employee is a Person. So, by using inheritance, you can implement an is-a type of relationship between classes.

16.1 Inheriting a class

We have the following class `Person` with four instance variables and three methods and we have made a new class named `Employee` by inheriting from this class.

```
class Person:
    def __init__(self, name, age, address, phone):
        self.name = name
        self.age = age
        self.address = address
        self.phone = phone
    def greet(self):
        print('Hello I am', self.name)
    def is_adult(self):
        if self.age > 18:
            return True
        else:
            return False
    def contact_details(self):
        print(self.address, self.phone)
class Employee(Person):
```

```
pass
```

When we write a new class from scratch, after the class name, we have the colon but when we write a class by inheriting from an existing class, after the class name we have the name of the existing class inside the parentheses. Since we are creating our `Employee` class by inheriting from the `Person` class, the name `Person` is inside the parentheses. The line `class Employee(Person) :` means create a new class `Employee` that inherits from the `Person` class.

In the class definition we just have a `pass` statement. We have not written anything inside this class but since it is inherited from the `Person` class, it gets access to everything from the `Person` class. Let us create an instance of this class:

```
>>> emp = Employee('Raghu', 30, 'D4, XYZ Street,
Delhi', '994477291')
```

The `Employee` class has access to the `__init__` method of the `Person` class, so all these arguments will be passed to that `__init__` method. This instance object will have all the attributes `name`, `age`, `address` and `phone`.

```
>>> emp.name
'Raghu'
>>> emp.age
30
>>> emp.address
'D4, XYZ Street, Delhi'
>>> emp.phone
'994477291'
```

We can call all the methods of the `Person` class through `emp`.

```
>>> emp.greet()
Hello I am Raghu
```

```
>>> emp.is_adult()
```

```
True
```

```
>>> emp.contact_details()
```

```
D4, XYZ Street, Delhi 994477291
```

So, we can see that the instance object of `Employee` class has access to everything from the `Person` class. Let us use the `isinstance` function on `emp`.

```
>>> isinstance(emp, Employee)
```

```
True
```

```
>>> isinstance(emp, Person)
```

```
True
```

The `isinstance` function returned `True` for the `Person` class also, which proves the is-a relationship between `Employee` and `Person`.

There is another built-in function named `issubclass` that can be used to check whether a class is subclass of another class.

```
>>> issubclass(Employee, Person)
```

```
True
```

This returns `True` because `Employee` is a subclass of `Person` class.

In the process of inheritance, the base class is not changed in any way. The derived class can differentiate itself from the base class in two ways: by adding new data members and methods or by overriding the methods of the base class. We will see how to do this in the coming sections.

16.2 Adding new methods and data members to the derived class

While defining our derived class, we can add new data members and methods that are specific to our derived class. We will again create our

Employee class by inheriting from the Person class but this time, we will add some new methods and data members to our Employee class.

```
class Employee(Person):  
    def __init__(self, name, age, address, phone,  
salary, office_address, office_phone):  
        self.name = name  
        self.age = age  
        self.address = address  
        self.phone = phone  
        self.salary = salary  
        self.office_address = office_address  
        self.office_phone = office_phone  
    def calculate_tax(self):  
        if self.salary < 5000:  
            return 0  
        else:  
            return self.salary * 0.05
```

We have defined two new methods in this class `__init__` and `calculate_tax`. The `__init__` method of the base class is inherited but our derived class needs some additional variables that need to be initialized so the inherited base method will not be sufficient. An Employee object needs to have 3 more instance variables which are salary, office_address and office_phone. In the `__init__` method, the first 4 parameters are the same as in Person class and after that we have added 3 new parameters. The first four lines can be copied from the `__init__` of Person class, and then we have written three more lines to create the three attributes that are specific to the Employee class.

Now when we create an Employee instance, we need to send seven arguments.

```
>>> emp = Employee('Raghu', 30, 'D4, XYZ Street,
Delhi', '994477291', 8000, 'ABC Street, Delhi',
'897657888')
```

Since Employee class has its own `__init__` now, the `__init__` of Person will not be called when we create an Employee instance. The `__init__` of Employee is called and all the seven instance attributes are created. The Employee class inherits the methods `greet`, `is_adult`, `contact_details` from the Person class and has its own method named `calculate_tax`.

```
>>> emp.name
```

```
'Raghu'
```

```
>>> emp.salary
```

```
8000
```

```
>>> emp.calculate_tax()
```

```
400.0
```

16.3 Overriding a base Method

Sometimes you may want a method from the base class but you would like it to behave differently in the derived class. For example, we want a `contact_details` method for Employee class but we want that to have a different definition from what is there in the Person class. In such a case, you can override the method. To override a method, just define a method in the derived class with same name as in the base class. So, let us override the method `contact_details`. In Employee class, we want to display the `office_address` and `office_phone` also.

```
class Employee(Person):
```

```
    def __init__(self, name, age, address, phone,
salary, office_address, office_phone):
```

```

        self.name = name
        self.age = age
        self.address = address
        self.phone = phone
        self.salary = salary
        self.office_address = office_address
        self.office_phone = office_phone
    def calculate_tax(self):
        if self.salary < 5000:
            return 0
        else:
            return self.salary * 0.05
    def contact_details(self):
        print(self.address, self.phone)
        print(self.office_address,
self.office_phone)

```

Now Employee has its own version of `contact_details`, and when an Employee instance will call this method, its own version will be executed instead of the base class version.

```

>>> emp = Employee('Jack', 30, 'D4, XYZ Street,
Delhi', '994477291', 8000, 'ABC Street, Delhi',
'897657888')

```

```

>>> emp.contact_details()
D4, XYZ Street, Delhi 994477291
ABC Street, Delhi 897657888

```

So, if a derived class defines a method with same name as a method in base class, then the derived class method overrides the method of base class, it

effectively hides the base class method. We have seen that this happened with the `__init__` method also. Since we have written a definition for `__init__` in the `Employee` class, the base class version of `__init__` is hidden.

16.4 Invoking the base class methods

While overriding a method, most of the times you want to extend the base class method instead of replacing it fully. So, mostly you need to begin by calling the base class method and then add special code that is specific to the derived class. The base class version can be accessed in the derived class by calling it explicitly using the base class name. For example, in `contact_details` method, instead of copying the code from the base class, we could call the base class version. We have overridden the `__init__` method also, so we could do the same thing in `__init__` also. Instead of copying the code from `__init__` of `Person` class, we could call the `Person` class `__init__`.

```
class Employee(Person):  
    def __init__(self, name, age, address, phone,  
salary, office_address, office_phone):  
        Person.__init__(self, name, age, address,  
phone)  
  
        self.salary = salary  
        self.office_address = office_address  
        self.office_phone = office_phone  
  
    def calculate_tax(self):  
        if self.salary < 5000:  
            return 0  
        else:  
            return self.salary * 0.05
```

```
def contact_details(self):  
    Person.contact_details(self)  
    print(self.office_address,  
self.office_phone)
```

If the derived class is overriding a method and wants to use the functionality of the base version, then it is better to call the base method instead of just copying the code. This reduces code duplication and later if the base class method changes, the change will be reflected in the derived class method. A better way of calling the base class method is by using the super built in function.

```
class Employee(Person):  
    def __init__(self, name, age, address, phone,  
salary, office_address, office_phone):  
        super().__init__(name, age, address,  
phone)  
        self.salary = salary  
        self.office_address = office_address  
        self.office_phone = office_phone  
    def calculate_tax(self):  
        if self.salary < 5000:  
            return 0  
        else:  
            return self.salary * 0.05  
    def contact_details(self):  
        super().contact_details()  
        print(self.office_address,  
self.office_phone)
```

Now there is no need of sending `self` as the first argument. Use of `super` is preferred because using base class name can create confusion in multiple inheritance, where a class inherits from more than one class. Writing `super` makes sure that all the base class versions are called, even from the classes that are inherited indirectly as we will see shortly.

16.5 Multilevel Inheritance

We can have multilevel inheritance which means that from the derived class we can further inherit another class. For example, from the `Employee` class we can inherit a class called `Teacher` and a class called `Accountant`.

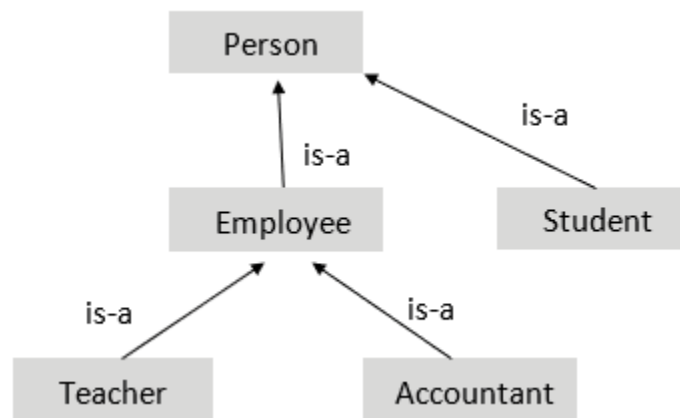


Figure 16.4: Multilevel inheritance

All attributes of `Person` are available in `Employee` and all attributes of `Employee` are available in `Teacher` class and `Accountant` class. From `Person` we have inherited another class named `Student`. We know that there is an is-a relationship between the derived classes and base classes. Therefore, `Teacher` is-a `Employee`, `Accountant` is-a `Employee`, `Employee` is-a `Person`, `Student` is-a `Person`, `Teacher` is-a `Person`, `Accountant` is-a `Person`.

An advantage of inheritance is that you can design your system by using inheritance so that it will reflect the natural relationship between different components of your system. This simplifies the design and makes programs easier to understand. Thus, reusability of code and being able to represent

the system using hierarchy of classes are the two main benefits of inheritance. However, you should use inheritance only when there is some natural relationship between classes. Unnecessary use of inheritance can make the system incomprehensible and can create unwanted dependencies between classes.

16.6 object class

`object` class is a built-in class from which every class automatically inherits. All built-in classes inherit from it and the custom classes that you define also inherit from it.

```
>>> class Person:
    pass

>>> issubclass(Person, object)

True
```

We defined a `Person` class, and we can see that it automatically inherits from the `object` class. So, when we don't specify any base class, the class directly inherits from `object` class. The above class definition is equivalent to explicitly inheriting from the `object` class.

```
>>> class Person(object):
    pass
```

In Python 3, this is redundant. It is not necessary to define a new class by deriving it explicitly from the `object` class. Any class that is defined without an explicit base class will be a derived class of `object`. So, every class inherits from the `object` class directly or indirectly. This class is at the top of any inheritance hierarchy in Python.

Our `Person` class definition is empty, but if we call `dir` function on it, we don't see an empty list.

```
>>> class Person:
    pass
```

```
>>> dir(Person)
```

```
['__class__', '__delattr__', '__dict__',  
 '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattr__', '__getstate__',  
 '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__le__', '__lt__',  
 '__module__', '__ne__', '__new__', '__reduce__',  
 '__reduce_ex__', '__repr__', '__setattr__',  
 '__sizeof__', '__str__', '__subclasshook__',  
 '__weakref__']
```

Most of the attributes in this list come from the `object` class since our `Person` class is implicitly derived from the `object` class.

```
>>> dir(object)
```

```
['__class__', '__delattr__', '__dir__', '__doc__',  
 '__eq__', '__format__', '__ge__',  
 '__getattr__', '__getstate__', '__gt__',  
 '__hash__', '__init__', '__init_subclass__',  
 '__le__', '__lt__', '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__sizeof__', '__str__',  
 '__subclasshook__']
```

The dunder methods available in `object` class are inherited by the classes that you define and so are available in all your classes. You can override these methods by providing your own definition in the class. For example, when you use `==` with your instance objects, Python will call the `__eq__` method defined in the `object` class, which will compare objects based on their identity. If you want the objects to be compared in some other way, you can override `__eq__` by defining it in your class. If you do so, the interpreter will call your version when you use `==` with the instance objects.

All built-in types like `str`, `int`, `dict` are names of classes, and so they are also subclasses of `object` class.

```
>>> issubclass(str, object)
```

```
True
>>> issubclass(int, object)
True
>>> issubclass(dict, object)
True
```

16.7 Multiple Inheritance

Multiple inheritance is not very commonly used as it can make the design quite complex and confusing, but it is good to have an idea about it as you might encounter it in some library or some other code.

Till now, the inheritance that we have seen is single inheritance, which means that a class inherits from a single class. Python supports multiple inheritance, meaning a class can inherit from multiple base classes.

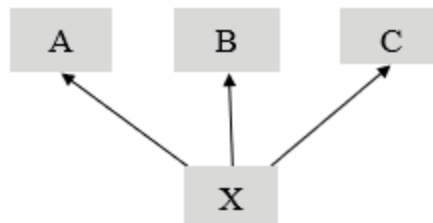


Figure 16.5: Multiple inheritance

Here class X inherits from classes A, B and C. All the data members and methods of all the three base classes will be available in the derived class X.

```
class X(A, B, C):
    pass
```

This is the syntax of defining a new class that inherits from multiple classes. All the base classes are placed inside the parentheses. This class definition creates a new class named X that inherits from classes A, B and C.

Here is another example of multiple inheritance:

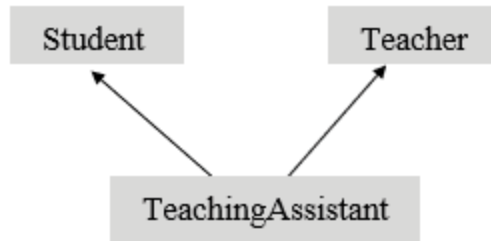


Figure 16.6: *TeachingAssistant inheriting from Student and Teacher*

A teaching assistant is a student who also teaches. Thus, the class `TeachingAssistant` inherits from both the class `Teacher` and `Student`.

```
class Teacher:
    def greet(self):
        print('I am a Teacher')

class Student:
    def greet(self):
        print('I am a Student')

class TeachingAssistant(Student, Teacher):
    def greet(self):
        print('I am a Teaching Assistant')
```

The class `TeachingAssistant` is inherited from classes `Teacher` and `Student`. All the 3 classes have defined a method named `greet`. Let us create an instance of the class `TeachingAssistant` and call the method `greet` on it.

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Teaching Assistant
```

Since the class `TeachingAssistant` has its own `greet` method, it will be called. Now suppose the class `TeachingAssistant` had not defined the `greet` method.

```
class TeachingAssistant(Student, Teacher):  
    pass
```

Now `greet()` is not present in this class, so the interpreter will look for it in the base classes. Both the base classes have the method named `greet`, so now the question is which one will be executed.

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Student
```

This output shows us that the `greet` method from `Student` class was executed. This is because while searching the multiple base classes, the search is performed from left to right. Let us change the order of base classes in the class definition and execute the `greet` method again.

```
class TeachingAssistant(Teacher, Student):  
    pass
```

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Teacher
```

Now the `greet` method from `Teacher` is executed because the `Teacher` class is now on the left in the class definition of `TeachingAssistant`.

All the base classes of a class can be seen using the `__bases__` attribute.

```
>>> TeachingAssistant.__bases__  
(<class '__main__.Teacher'>, <class  
'__main__.Student'>)
```

We get a tuple with both the base classes and the order of the base classes is the same as specified in the class definition.

So, we saw an example of multiple inheritance, but it was just one level deep. There can be multiple levels of inheritance, for example the two class

Student and Teacher could be derived from a common base class Person.

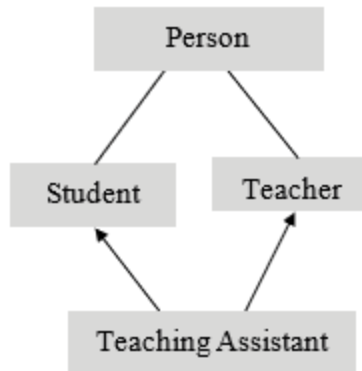


Figure 16.7: Diamond inheritance

This is known as diamond inheritance. We know that every class is derived from `object`, so it is the base class for `Person`. If there are many classes in the system, and both multilevel and multiple inheritance are involved, then the whole structure can be quite complex. As the structure becomes complex, searching for attributes in base classes does not remain straightforward. Conflicts can arise if the base classes contain attributes with the same names. To resolve any sort of conflict while searching for attributes in base classes, Python uses a well-defined algorithm which is the topic of the next section.

16.8 Method Resolution Order (MRO)

The order in which Python searches for attributes in base classes is called method resolution order (MRO). It gives a linearized path for an inheritance structure.

Python computes an MRO for every class in the hierarchy; this MRO is computed using the ‘C3 linearization algorithm’. This algorithm is quite complicated, you can check the documentation if you are interested in the details but roughly it works in a depth first, left to right manner, and it searches each class only once. For example, in our previous diamond inheritance example, the `Person` class can be reached in two ways but it will be looked up only once in MRO.

We can see the MRO for any class using the `__mro__` attribute or the `mro` method or by using the `help` function. If we have an instance and want to see its MRO dynamically, we can use the `__class__` attribute.

Here is the code for the diamond example that we have seen. The classes `Student` and `Teacher` inherit from class `Person`, and the class `TeachingAssistant` inherits from classes `Student` and `Teacher`. All the classes have defined a method named `greet`.

```
class Person:
    def greet(self):
        print('I am a Person')
class Teacher(Person):
    def greet(self):
        print('I am a Teacher')
class Student(Person):
    def greet(self):
        print('I am a Student')
class TeachingAssistant(Student, Teacher):
    def greet(self):
        print('I am a Teaching Assistant')
```

If we use `help` on the class `TeachingAssistant`, we will see MRO for it at the top.

```
>>> help(TeachingAssistant)
Help on class TeachingAssistant in module
__main__:
class TeachingAssistant(Student, Teacher)
|   Method resolution order:
|       TeachingAssistant
```

```
|      Student
|      Teacher
|      Person
|      builtins.object
```

.....

If we use any attribute on an instance of the class `TeachingAssistant`, first it will be searched in the class `TeachingAssistant`, then `Student`, then `Teacher` and then in `Person`, and at last in the built-in object class. The search will stop as soon as the attribute is found, and if the attribute is not found in any of these classes, then an error will be raised.

We can get this MRO in a tuple by using the `__mro__` attribute on the class name.

```
>>> TeachingAssistant.__mro__
(<class '__main__.TeachingAssistant'>, <class '__main__.Student'>, <class '__main__.Teacher'>, <class '__main__.Person'>, <class 'object'>)
```

If we use the `mro()` method, we get this order in a list.

```
>>> TeachingAssistant.mro()
[<class '__main__.TeachingAssistant'>, <class '__main__.Student'>, <class '__main__.Teacher'>, <class '__main__.Person'>, <class 'object'>]
```

To find the MRO through an instance dynamically, we can specify the `__class__` attribute before the `__mro__` attribute.

```
>>> x = TeachingAssistant()
>>> x.__class__.__mro__
(<class '__main__.TeachingAssistant'>, <class '__main__.Student'>, <class '__main__.Teacher'>, <class '__main__.Person'>, <class 'object'>)
```

The class `TeachingAssistant` has its own version of `greet` method so when we call the method `greet` on an instance of `TeachingAssistant`, this method is executed.

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Teaching Assistant
```

Now let us delete the `greet` method from the `TeachingAssistant` class.

```
class TeachingAssistant(Student, Teacher):  
    pass
```

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Student
```

Now the interpreter did not find `greet` in `TeachingAssistant`, so it searched for it in `Student` which is next in line in MRO. Let us delete `greet` from the `Student` class also.

```
class Student(Person):  
    pass
```

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
I am a Teacher
```

Next in the MRO hierarchy is `Teacher`, so this method is executed. Let us delete it from the `Teacher` class also.

```
class Teacher(Person):  
    pass
```

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

I am a Person

Next in line was Person class, so this method is executed. If we delete the greet method from the Person class, also, then next class in MRO will be object class, and it does not have any greet method, so we will get an error.

```
>>> x = TeachingAssistant()
```

```
>>> x.greet()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#17>", line 1, in <module>
```

```
    x.greet()
```

```
AttributeError: 'TeachingAssistant' object has no  
attribute 'greet'
```

So, this is how Python looks for attributes in base classes. Although this order is called method resolution order, it holds true while searching for data attributes also.

16.9 super and MRO

The method resolution order that we saw in the last section is used by the built in function `super()` also. This function always invokes the next class in the MRO.

In [section 16.4](#), we had seen that when we override a method in the derived class and need to call the base class version in that method, we could use base class name or `super()`. In single inheritance, there is not much confusion as every class has a single parent in the inheritance chain, but in multiple inheritance, a class can have multiple parents and using `super` avoids all problems. To understand this, we will again take the same example of multiple inheritance that we saw in the previous section.

```
class Person:
```

```
    def greet(self):
```

```
        print('I am a Person')
class Teacher(Person):
    def greet(self):
        Person.greet(self)
        print('I am a Teacher')
class Student(Person):
    def greet(self):
        Person.greet(self)
        print('I am a Student')
class TeachingAssistant(Student, Teacher):
    def greet(self):
        Student.greet(self)
        Teacher.greet(self)
        print('I am a Teaching Assistant')
```

All the classes have defined the method `greet`. In the `Teacher` class, we have called the base class version of `greet` in the overridden method. Similarly, in `Student` class also we have called the base class version of `greet`. The class `TeachingAssistant` has two base classes so we have called `greet` versions from both the base classes.

Now let us create a `TeachingAssistant` instance and call `greet` method on it.

```
>>> x = TeachingAssistant()
>>> x.greet()
I am a Person
I am a Student
I am a Person
I am a Teacher
```

I am a Teaching Assistant

The `greet` method of `TeachingAssistant` class is executed. Inside this method, first the method `Student.greet()` is executed, then `Teacher.greet()` is executed and then the message 'I am a Teaching Assistant' is printed. `Student.greet` in turn calls `Person.greet`, and `Teacher.greet` also calls `Person.greet`. So, the `greet` method from `Person` class will be called two times. This repetition can be avoided if we replace the base class names with `super`.

```
class Person:
    def greet(self):
        print('I am a Person')

class Teacher(Person):
    def greet(self):
        super().greet()
        print('I am a Teacher')

class Student(Person):
    def greet(self):
        super().greet()
        print('I am a Student')

class TeachingAssistant(Student, Teacher):
    def greet(self):
        super().greet()
        print('I am a Teaching Assistant')

>>> x = TeachingAssistant()
>>> x.greet()
I am a Person
I am a Teacher
```

```
I am a Student
```

```
I am a Teaching Assistant
```

Now the greet from Person was executed only once, this is because the function super follows MRO.

```
>>>help(TeachingAssistant)
|  Method resolution order:
|      TeachingAssistant
|      Student
|      Teacher
|      Person
|      builtins.object
```

.....

This is the MRO for TeachingAssistant class. When the super() function is invoked in TeachingAssistant class, it refers to Student, because it is the next in MRO. In Student if again super is present, it invokes Teacher class as it is the next in MRO. And then super inside Teacher class invokes the class Person as it is the next in MRO. We can see that super() follows MRO and it does not always call the parent of a class, it calls the one that is next in line based on MRO.

So, we have seen how these super calls perfectly called the base class versions without any repetition. The repetition that we get if we don't use super, can cause hard to find bugs when __init__ is called using base class names. There is an example in the exercise which will help you understand this.

Now suppose we have an instance of Student class, and we call the greet method with that Student instance.

```
>>> s = Student()
>>> s.greet()
I am a Person
```


I am a Student

The method `super` in `Student` class will invoke the `greet` method of `Person` class. This is because in MRO of `Student` class, `Person` comes next.

```
>>> help(Student)
| Method resolution order:
|     Student
|     Person
|     builtins.object
```

Thus we have seen that the method resolution order is used by Python when searching for attributes in base classes and it is also used by the built in function `super`.

It is good to use `super()` whether you are using single inheritance or multiple inheritance. In multiple inheritance, the advantage is obvious. In single inheritance, also `super` can be beneficial if there are some updates made in the future like changing the name of the base class or switching to multiple inheritance. Thus, it makes the code more maintainable.

16.10 Polymorphism

The three main features of object-oriented programming are - encapsulation, inheritance and polymorphism. We have seen the first two, now let us see what is polymorphism. The meaning of the word polymorphism is the ability to take many forms. In programming, this means the ability of code to take different forms depending on the type with which it is used. The behaviour of the code can depend on the context in which it is used. Let us understand this with the help of an example.

```
def do_something(x):
    x.move()
    x.stop()
```

We have this function `do_something` that has a parameter `x`. Inside this function, two methods are called on `x`. We know that Python is a dynamically typed language; there are no type declarations. The type of parameter `x` is not declared, we can send any type of object to this function. We could send a `list` object or a `str` object, but in that case, we will get error because `str` and `list` types do not support the methods `move` and `stop`. The function `do_something` will work correctly as long as we send objects of those types that support the two methods `move` and `stop`.

Next, we have defined three classes that have the methods `move` and `stop`. The implementation for these methods is different in each one of them.

```
class Car:
    def start(self):
        print('Engine started')
    def move(self):
        print('Car is running')
    def stop(self):
        print('Brakes applied')
class Clock:
    def move(self):
        print('Tick Tick Tick')
    def stop(self):
        print('Clock needles stopped')
class Person:
    def move(self):
        print('Person walking')
    def stop(self):
        print('Taking rest')
    def talk(self):
```

```
print('Hello')
```

Let us create instance objects of these classes.

```
>>> car = Car()
```

```
>>> clock = Clock()
```

```
>>> person = Person()
```

We can send all these instance objects to the `do_something` function since all three of them support the `move` and `stop` functions.

```
>>> do_something(car)
```

```
Car is running
```

```
Brakes applied
```

```
>>> do_something(clock)
```

```
Tick Tick Tick
```

```
Clock needles stopped
```

```
>>> do_something(person)
```

```
Person walking
```

```
Taking rest
```

So, any object that supports the two operations `move` and `stop` can be sent to this function. The behaviour of `move` and `stop` depends on the type of the object that they are operating upon. This is polymorphism, the same code can take different forms. While executing the code of function `do_something`, interpreter does not care about the type of `x`; any object that supports the two methods `move` and `stop` will work regardless of its specific type. Python is not concerned about what an object is, it just needs to know what an object does. Let us see another example.

```
class Rectangle:
```

```
    name = 'Rectangle'
```

```
    def __init__(self, length, breadth):
```

```
        self.length = length
```

```

        self.breadth = breadth
    def area(self):
        return self.length * self.breadth
    def perimeter(self):
        return 2 * (self.length + self.breadth)
class Triangle:
    name = 'Triangle'
    def __init__(self, s1, s2, s3):
        self.s1 = s1
        self.s2 = s2
        self.s3 = s3
    def area(self):
        sp = (self.s1 + self.s2 + self.s3) / 2
        return ( sp*(sp-self.s1)*(sp-self.s2)*(sp-
self.s3) ) ** 0.5
    def perimeter(self):
        return self.s1 + self.s2 + self.s3
class Circle:
    name = 'Circle'
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius
    def perimeter(self):
        return 2 * 3.14 * self.radius

```

```

def find_area_perimeter(shape):
    print(shape.name)
    print('Area : ', shape.area() )
    print('Perimeter : ', shape.perimeter() )

r1 = Rectangle(13, 25)
r2 = Rectangle(14, 16)
t1 = Triangle(14, 17, 12)
t2 = Triangle(25, 33, 52)
c1 = Circle(14)
c2 = Circle(25)

```

We have three classes named `Rectangle`, `Triangle`, and `Circle`. All the three classes have the methods named `area` and `perimeter` and all of them have a class variable `name`. In the `Rectangle` class, we have two instance variables `length` and `breadth` and the area is calculated by multiplying them and the perimeter by the formula $2 \times (\text{length} + \text{breadth})$. In the class `Triangle`, we have three instance variables which represent the three sides, the area is calculated using Heron's formula and the perimeter is calculated by adding the three sides. In the `Circle` class, there is only one instance variable which is the radius, area is πr^2 and perimeter is $2\pi r$. We have created a polymorphic function `find_area_perimeter` and created two instance objects of each class. Let us call the function with these instance objects as a parameter.

```

>>> find_area_perimeter(t2)
Triangle
Area :  330.0
Perimeter :  110
>>> find_area_perimeter(c1)
Circle
Area :  615.44

```

```
Perimeter : 87.92
>>> find_area_perimeter(r2)
Rectangle
Area : 224
Perimeter : 60
```

We can see that the code inside the function `find_area_perimeter` could take different forms depending on the type of shape.

Now, suppose we have a list of these objects, and we want to find out the total area and perimeter of all the shapes in this list:

```
shapes = [r1, r2, t1, t2, c1, c2]
total_area = 0
total_perimeter = 0
for shape in shapes:
    total_area += shape.area()
    total_perimeter += shape.perimeter()
print(total_area, total_perimeter)
```

In the `for` loop, we are iterating over the list and calling the `area` and `perimeter` methods on each object. After that, we print the total area and perimeter. This is again an example of polymorphic code.

Other object-oriented languages might need these classes to be derived from a common base class to exhibit this polymorphic behaviour. However, in Python there is no such restriction, polymorphism in Python does not depend on inheritance. For polymorphism to occur you just need to define different classes which have commonly named methods.

Python's polymorphism is based on duck typing, which comes from the old saying, 'If it walks like a duck and quacks like a duck, then it is a duck.' Different objects that have common method names can be treated in the same general way. Let us see some benefits of polymorphism.

You can write generic code that can work with objects of different classes. When this generic code is executed, Python uses polymorphism to call the appropriate method for each instance object.

Polymorphism makes your code concise and flexible and provides a sort of abstraction. When writing the generic code, a programmer need not think about the specific classes that will use the code.

The code becomes easy to update also, you can easily add new types. The functions that are already written can work with new types that you define in future as long as those new types support the required operations. For example, in future you can add a `Rhombus` class with `area` and `perimeter` methods, and you can easily use it with the polymorphic code that we have seen before.

The behavior shown by overloaded operators is also polymorphism. An overloaded operator takes different forms depending on the type it is operating upon. For example, the `+` operator can be used with integers, strings, and lists. Its behavior varies based on the type it interacts with, thus exhibiting polymorphism. The following function can take different forms depending upon the type of objects `a` and `b`.

```
def func(a, b):  
    print(a + b)  
    print(a * b)
```

It will work correctly for objects of any type that support addition and multiplication.

16.11 Abstract Base classes

We have seen that if we have to define a group of classes that have similar features and show common behavior, we can define a base class and then inherit the classes from it. In the derived classes, we have the choice to either use the base class version of a method or override it. There can be scenarios when it does not make sense to implement some methods in the base class. We need to define a method in the base class just to provide a

common interface for the derived classes. We do not need such a base class to be instantiated.

In the following example we have defined a base class Shape from which different classes like Rectangle, Triangle can be derived.

```
class Shape:
    def area(self):
        pass
    def perimeter(self):
        pass
    def draw(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
    def area(self):
        return self.length * self.breadth
    def perimeter(self):
        return 2 * (self.length + self.breadth)
    def draw(self):
        print('Drawing a rectangle')

class Triangle(Shape):
    def __init__(self, s1, s2, s3):
        self.s1 = s1
        self.s2 = s2
        self.s3 = s3
```



```

    def area(self):
        sp = (self.s1 + self.s2 + self.s3) / 2
        return ( sp*(sp-self.s1)*(sp-self.s2)*(sp-
self.s3) ) ** 0.5

    def perimeter(self):
        return self.s1 + self.s2 + self.s3

    def draw(self):
        print('Drawing a triangle')

```

The base class Shape has three methods, each with an empty body. Each derived class will implement the area, perimeter and draw methods in its own way and so it will override these methods. It is not possible to provide a definition for these methods in the base class, since they will perform different operations depending on the type of object. The definition of these methods in the base class provides a common interface for the derived classes. With a common base class, your program becomes easier to extend. When developers have to add new classes like Circle, Rhombus, they can inherit from the Shape class and maintain a common interface.

We can create an instance of Shape class, but it does not make much sense. There is no use of instantiating a Shape class, its purpose is just to serve as a base class for the other classes.

Such types of classes that are meant to be inherited and not instantiated should be marked as abstract base classes. An abstract base class generally represents a model or an abstract concept – something that has no physical form; for example, Shape is an abstract concept, while Rectangle and triangle represent real things. To mark a class as an abstract base class, we have to make use of the abc module of the standard library.

```

from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod

    def area(self):

```

```

        pass

    @abstractmethod
    def perimeter(self):
        pass

    @abstractmethod
    def draw(self):
        pass

```

The Shape class now inherits from the ABC class of abc module. The methods area, perimeter, and draw are now decorated with the `abstractmethod` decorator, so they are abstract methods. By making a method abstract we force all the derived classes to implement that method. If the derived class does not implement an abstract method, then there will be an error while instantiating that class.

To make a class an abstract class we have to inherit from the `abc.ABC` class and it should have at least one abstract method in it. To make a method an abstract method we have to apply the `@abstractmethod` decorator from the abc module. We cannot create any instance object of an abstract class, and any class that inherits from an abstract class should override all its abstract methods.

In our example, Shape class is an abstract class so it cannot be instantiated. The derived classes that can be instantiated are called concrete classes. The abstract class provides a sort of blueprint or a template for its subclasses. It defines the methods that the subclasses should implement. An abstract class is not meant to be instantiated; it exists only to be used as a base class that provides a basic foundation for the derived classes. Derived classes that implement the abstract methods are concrete classes that represent the real things that are modeled, and they can be instantiated.

We have seen in polymorphism that if objects have common method names, they can be treated in the same general way. Abstract base classes provide a strict common interface that has to be followed by the subclasses. They force the subclasses to use the same method names for similar types of

tasks, and hence, it becomes easier to maintain the class hierarchies and achieve polymorphism.

In our next example program, Employee is an abstract class, while the classes PartTimeEmployee, FullTimeEmployee and TemporaryEmployee are concrete classes.

```
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
    def contact_details(self):
        print(self.name, self.phone)
    @abstractmethod
    def compute_salary(self):
        pass

class PartTimeEmployee(Employee):
    def __init__(self, name, phone, hours):
        super().__init__(name, phone)
        self.hours = hours
    def compute_salary(self):
        print('Calculating salary of part time employee')

class FullTimeEmployee(Employee):
    def compute_salary(self):
        print('Calculating salary of full time employee')

class TemporaryEmployee(Employee):
```

```
def compute_salary(self):  
    print('Calculating salary of temporary  
employee')  
e1 = PartTimeEmployee('Jack', 999909090, 4)  
e2 = FullTimeEmployee('Jim', 989898989)  
e3 = TemporaryEmployee('John', 789898989)  
employees = [e1, e2, e3]  
for e in employees:  
    e.contact_details()  
    e.compute_salary()
```

Since the method of computing the salary differs for each employee, there is no point in implementing the method `compute_salary` in the base class. Every derived class should override this method to give its own implementation. This is why it is marked as an abstract method. Each subclass is expected to provide an implementation for `compute_salary` method.

An abstract class can have non-abstract methods as well, which the derived classes do not need to override. For example, in the `Employee` class, the method `contact_details` is not an abstract method and so it is not necessary for the derived classes to override it. The user has the choice to either use the base class implementation or override the method and define its own implementation.

Most of the time, abstract methods have an empty body in the abstract class. They are there only for defining the common interface, so their body generally contains a `pass` statement. However, the abstract methods of an abstract class can contain some basic implementation that the concrete subclasses can call by using `super`. Even if the abstract method is implemented in the abstract base class, the subclass has to override it. The subclass can call the base implementation by using `super` and then add its own code for any additional tasks.

You can also declare property methods, class methods, or static methods as abstract:

```
@property
@abstractmethod
def name(self):
    pass

@classmethod
@abstractmethod
def method1(cls):
    pass

@staticmethod
@abstractmethod
def method2():
    pass
```

There are many predefined abstract base classes available in the standard library. The `collections.abc`, `numbers`, and `io` modules define abstract base classes that can be inherited. When you want to define collections that share the same interface as that of built-in types, you can inherit from one of the classes in `collections.abc` module. This module differs from the module `abc` that contains `ABC` and the `abstractmethod` decorator.

16.12 Composition

We have seen how to reuse the code of an existing class by inheriting a class from it. Another way to use the code of an existing class is composition (containership). Inheritance and composition are two different design constructs or design concepts: inheritance is used when you want to implement *is-a* relationship between classes while composition is used when there is a *has-a* relationship between classes. For example, a car is-a

vehicle but has-a engine. An engine is not a kind of a Car but it is a part of a Car. There is a has-a relationship between Car and Engine, so you have to use composition. When you use composition, you embed one or more objects inside another object. So, we can make composite objects that contain other objects called components; for example, a Car object can be viewed as a composite object which has an engine, brakes, gears, etc. Let us see how we can achieve composition in Python.

Till now, we have been using instance variables of built-in types in our classes. To implement the concept of composition, we will make instance variables that refer to objects of other user-defined classes. Whenever we want to use any attribute of the contained class, we will have to use it through the instance. In the following program, we have made the classes Engine and Brakes, and then inside the Car class we have instantiated these classes. So, the Car class is the composite class, while the Engine and Brakes classes are component classes.

```
class Engine:
    def __init__(self, power):
        self.power = power
    def start(self):
        self.draw_current()
        self.spin()
        self.ignite()
    def draw_current(self):
        print('Drawing current')
    def spin(self):
        print('Spinning')
    def ignite(self):
        print('Igniting')
class Brakes:
```

```
def __init__(self,weight):
    self.weight = weight
def activate(self):
    print('Activating brakes')
def release(self):
    print('Releasing brakes')
class Car:
    def __init__(self,name, engine, brakes):
        self.name = name
        self.engine = engine
        self.brakes = brakes
    def start(self):
        self.engine.start()
    def stop(self):
        self.brakes.activate()

e = Engine(120)
b = Brakes(5)
car = Car('Breeze', e, b)
car.start()
car.stop()
```

Output-

Drawing current

Spinning

Igniting

Activating brakes

In the `__init__` of `Car` class, we have created two instance variables of type `Engine` and `Brakes`, and used these instance variables inside the methods of the `Car` class. Through these instance variables, we call the methods of the `Engine` and `Brakes` class and hence get access to the implementation of these classes. When the `Car` object calls its `start` method, the embedded `Engine` object calls its `start` method, in turn, and when the `Car` object calls the `stop` method, the embedded `Brakes` object calls its `activate` method. The composite class is the controller that passes calls to the contained objects.

Composition makes the class easier to understand and use. The composite class can focus on the main task and can delegate different sub-tasks to the contained objects. So, each class can focus on performing a specific task, instead of a single complex class performing all the tasks. Composition also helps in reuse of code. You can use any class as a component in different classes.

When your class becomes too lengthy with many instance variables and methods, you can think of making a separate class for some of the parts of that class. Then you can include an instance of that new class in your class. You can also make use of existing classes in your class. For example, in the following class we have made use of the existing `Person` class and `datetime.date` class in our `Book` class. The `datetime.date` class from the standard library is used in the `Person` class also.

```
from datetime import date

class Person:
    def __init__(self, name, y, m, d, address,
phone):
        self.name = name
        self.address = address
        self.date_of_birth = date(y, m, d)
        self.phone = phone
    def contact_details(self):
```



```

        print(self.address, self.phone)

@property
def age(self):
    return (date.today() -
self.date_of_birth).days // 365

class Book:
    def __init__(self, title, pages, y, m, d,
author):
        self.title = title
        self.pages = pages
        self.publishing_date = date(y, m, d)
        self.author = author

    def display(self):
        print(f'{self.title} published in
{self.publishing_date.year}, ', end='')
        print(f'written by {self.author.name}')

    def author_details(self):
        print(f'Author name : {self.author.name},
age : {self.author.age}')
        self.author.contact_details()

    def __lt__(self, other):
        return (self.publishing_date) <
(other.publishing_date)

author1 = Person('Devank', 2010, 4, 29, '122 Madhi
Nath', 998998987)

author2 = Person('Devanshi', 1999, 5, 15, '256
Adyar', 878237288)

```

```
book1 = Book('Divine Dinosaurs', 200, 2020, 4, 29,
author1)

book2 = Book('Rocket Science', 200, 2021, 4, 29,
author1)

book3 = Book('How to overcome laziness', 500,
2010, 4, 29, author2)

books = [book1, book2, book3]

for book in books:
    book.display()

print()

print('List of books sorted by publishing date')
for book in sorted(books):
    print(book.title)

print()

print('List of books by young authors')
for book in books:
    if book.author.age < 18:
        print(book.title)

print()

print(f'Author details of "{book1.title}"')
book1.author_details()
```

Output-

Divine Dinosaurs published in 2020, written by
Devank

Rocket Science published in 2021, written by
Devank

How to overcome laziness published in 2010,
written by Devanshi

List of books sorted by publishing date

How to overcome laziness

Divine Dinosaurs

Rocket Science

List of books by young authors

Divine Dinosaurs

Rocket Science

Author details of "Divine Dinosaurs"

Author name : Devank, age : 13

122 Madhi Nath 998998987

We have instantiated the Date class and the Person class in our Book class and used the instances in the methods of the Book class.

Whenever you have to copy a composite object that contains other embedded objects, you should perform a deep copy by using the deepcopy function from the copy module.

Exercise

1. Create a class named Course that has instance variables title, instructor, price, lectures, users(list), ratings, avg_rating. Implement the methods __str__, new_user_enrolled, received_a_rating and show_details. From the above class, inherit two classes VideoCourse and PdfCourse. The class VideoCourse has instance variable length_video and PdfCourse has instance variable pages.
2. What will be the output of this code?

```
class Mother:
    def cook(self):
        print('Can cook pasta')
class Father:
    def cook(self):
        print('Can cook noodles')
class Daughter(Father, Mother):
    pass
class Son(Mother, Father):
    def cook(self):
        super().cook()
        print('Can cook butter chicken')
d = Daughter()
s = Son()
d.cook()
print()
s.cook()
```

3. What will be the output of this code ?

```
class Person:
    def greet(self):
        print('I am a Person')
class Teacher(Person):
    def greet(self):
        Person.greet(self)
        print('I am a Teacher')
```

```

class Student(Person):
    def greet(self):
        Person.greet(self)
        print('I am a Student')
class TeachingAssistant(Student, Teacher):
    def greet(self):
        super().greet()
        print('I am a Teaching Assistant')
x = TeachingAssistant()
x.greet()

```

4. In the following inheritance hierarchy, we have written code to add 'S' to the id of the Student, 'T' to the id of the Teacher, and both 'T' and 'S' to the id of the Teaching Assistant. What will be the output of this code? If the code does not work as intended, what changes do we need to make?

```

class Person:
    def __init__(self, id):
        self.id = id
class Teacher(Person):
    def __init__(self, id):
        Person.__init__(self, id)
        self.id += 'T'
class Student(Person):
    def __init__(self, id):
        Person.__init__(self, id)
        self.id += 'S'
class TeachingAssistant(Student, Teacher):

```

```

        def __init__(self,id):
            Student.__init__(self,id)
            Teacher.__init__(self,id)
x = TeachingAssistant('2675')
print(x.id)
y = Student('4567')
print(y.id)
z = Teacher('3421')
print(z.id)
p = Person('5749')
print(p.id)

```

5. What will be the output of the following code?

```

class Base:
    def method1(self):
        print('Base : method1 ')
    def method2(self):
        print('Base : method2 ')
    def method3(self):
        print('Base : method3 ')
class Derived(Base):
    def method2(self):
        print('Derived : method2 ')
    def method3(self):
        super().method3()
        print('Derived : method3 ')

```

```
b = Base()
d = Derived()
b.method1()
b.method2()
b.method3()
d.method1()
d.method2()
d.method3()
```

6. In the chapter, we saw examples of inheriting from user-defined classes. We can also inherit from built-in classes or any class from the standard library. The new class will have all the functionality of the base class and can have additional functionality.

Write a class `CustomString` by inheriting from the built in `str` class. This class should have methods `spacify`, `space_to_underscore`, `reverse`, `count_vowels`, `is_palindrome`. The method `spacify` should return a string in which each character of the original string is separated by a space, `space_to_underscore` should return a string in which all spaces are replaced with underscores, `reverse` should return the reversed string, `count_vowels` should return the number of vowels in the string and `is_palindrome` should return `True` if the string is a palindrome, otherwise `False`.

```
my_string = CustomString('Madam I am Adam')
print('Reversed:', my_string.reverse())
print('Number of vowels:',
my_string.count_vowels())
print('Is Palindrome:',
my_string.is_palindrome())
print(my_string.spacify())
print(my_string.space_to_underscore())
```

Output-

Reversed: madA ma I madaM

Number of vowels: 6

Is Palindrome: False

M a d a m I a m A d a m

Madam_I_am_Adam

17.1 Iterables

An iterable object is capable of returning its members one at a time. Such an object can be iterated over in a `for` loop and in other iteration contexts. Most built-in containers are iterables, e.g., lists, tuples, sets, dictionaries, strings.

An object is considered iterable if we can get an iterator from it when it is passed to the `iter` built-in function. So, an iterable object responds to the built-in function `iter` by returning an iterator object. We will take a few objects and will send them to the `iter` function one by one to see how they respond.

```
>>> L = [1, 2, 3]
>>> s = 'abc'
>>> d = {1: 'a', 2: 'b'}
>>> x = 200
>>> iter(L)
<list_iterator object at 0x000001F9CBFED100>
>>> iter(s)
<str_iterator object at 0x000001F9CC0BADC0>
>>> iter(d)
<dict_keyiterator object at 0x000001F9CC0C0AE0>
>>> iter(x)
```

```
TypeError: 'int' object is not iterable
```

The `list` object, `str` object and `dict` object respond to the `iter` function by returning an iterator and so they are iterables. The `int` object is not an iterable as it does not give an iterator when sent to the `iter` function.

Iterable objects are not limited to physical containers only. They also include virtual sequences that are computed lazily on demand. We will see later how to create these types of objects. Some built-in functions also return objects that are iterables. For example, the object returned by `range` is an iterable.

```
>>> a = range(1, 10)
>>> iter(a)
<range_iterator object at 0x000001F9CBFEBEB0>
```

The object returned by `range` is an iterable since it responds to the `iter` function by returning an iterator. Now let us see what is an iterator.

17.2 Iterators

An iterator is an object that represents a stream of data. It produces a stream of values, one at a time. An iterator responds to the built-in function `next` by returning the next item from the data stream that it represents. When you pass an iterator object to the function `next`, it returns the next item. When there are no more items left, it raises the `StopIteration` exception.

```
>>> L = [10, 20, 30, 40]
>>> it = iter(L)
>>> it
<list_iterator object at 0x02063B50>
```

We have a list that is an iterable object, and we got an iterator from this list object by sending it to the `iter` function. Now we will call the built-in function `next` with this iterator.

```
>>> next(it)
10
```

```
>>> next(it)
20
>>> next(it)
30
>>> next(it)
40
>>> next(it)
StopIteration
```

Each time we call `next`, we get an item from the list, starting from the first item. When there are no more values left in the list to return, the `StopIteration` error is raised by the `next` function. The iterator is exhausted; now, how many ever times we call `next` on it, we will get this error only.

So, the `next` function returns one item at a time and when there are no more items left, it raises the `StopIteration` exception. There is no way to go back and restart this iterator. If you want to iterate over the list again, you must get a fresh iterator by calling the `iter` function.

```
>>> it = iter(L)
```

Now, we have a fresh iterator, and we can start iterating again.

```
>>> next(it)
10
>>> next(it)
20
```

Now, let us get an iterator from the iterable returned by the `range` function.

```
>>> x = range(1, 8, 2)
>>> i = iter(x)
```

Now we have this iterator `i`, which represents the data stream 1, 3, 5, 7. When we will call `next` on it, these items will be returned one by one.

```
>>> next(i)
```

```
1
>>> next(i)
3
>>> next(i)
5
>>> next(i)
7
>>> next(i)
StopIteration
```

When the iterator is exhausted, and there are no items to return, the call to the next function raises a `StopIteration` exception.

It is possible to create multiple active iterators for an iterable object, and each iterator will maintain its own state of progress. For example, here, we have created three separate iterators for a list object.

```
>>> L = ['a', 'b', 'c', 'd', 'e']
>>> i1 = iter(L)
>>> i2 = iter(L)
>>> i3 = iter(L)
>>> next(i1)
'a'
>>> next(i1)
'b'
>>> next(i1)
'c'
>>> next(i2)
'a'
>>> next(i2)
'b'
>>> next(i3)
'a'
```

```
>>> next(i1)
'd'
>>> next(i2)
'c'
>>> next(i3)
'b'
```

We can see that each iterator is independent and knows its location.

Now, we know the difference between an iterator and an iterable. An iterable responds to the `iter` function and returns an iterator, while an iterator responds to the `next` function and returns the next element. A list object is an iterable, but it is not an iterator. We cannot use it with the `next` function since we will get an error if we use it.

```
>>> next(L)
TypeError: 'list' object is not an iterator
```

Similarly, the object returned by `range` is an iterable, not an iterator. So, you cannot use `next()` on it.

```
>>> x = range(1,5)
>>> next(x)
TypeError: 'range' object is not an iterator
```

The `next()` function works only for an iterator. It does not work with an iterable.

We have seen that an iterable responds to the `iter` function by returning an iterator. Now, let us see what happens when we send an iterator to the `iter` function.

```
>>> L = ['a', 'b', 'c', 'd', 'e']
>>> i1 = iter(L)
>>> i1
<list_iterator object at 0x000001E3FE70BEB0>
>>> iter(i1)
<list_iterator object at 0x000001E3FE70BEB0>
```

We sent the list iterator `il` to the `iter` function, and we got the same iterator object back. There was no error; this means that an iterator object also responds to the `iter` function and returns an iterator, which is the same object that was passed. We can also check this using the `is` operator.

```
>>> il is iter(il)
True
```

This confirms that when an iterator object is passed to the `iter` function, the `iter` function simply returns the same iterator object.

An iterable is an object that, when passed through the `iter` function, returns an iterator, and an iterator is an object that produces the next item using the `next` function, and when there are no more items, it raises a `StopIteration` exception. When an iterator is passed to the `iter` function, the iterator itself is returned. This way, you can say that an iterator is always an iterable object because it responds to the `iter` function by returning an iterator. But an iterable object is not always an iterator, as it cannot respond to the function `next`. We can think of iterators as special iterables that act as their own iterators.

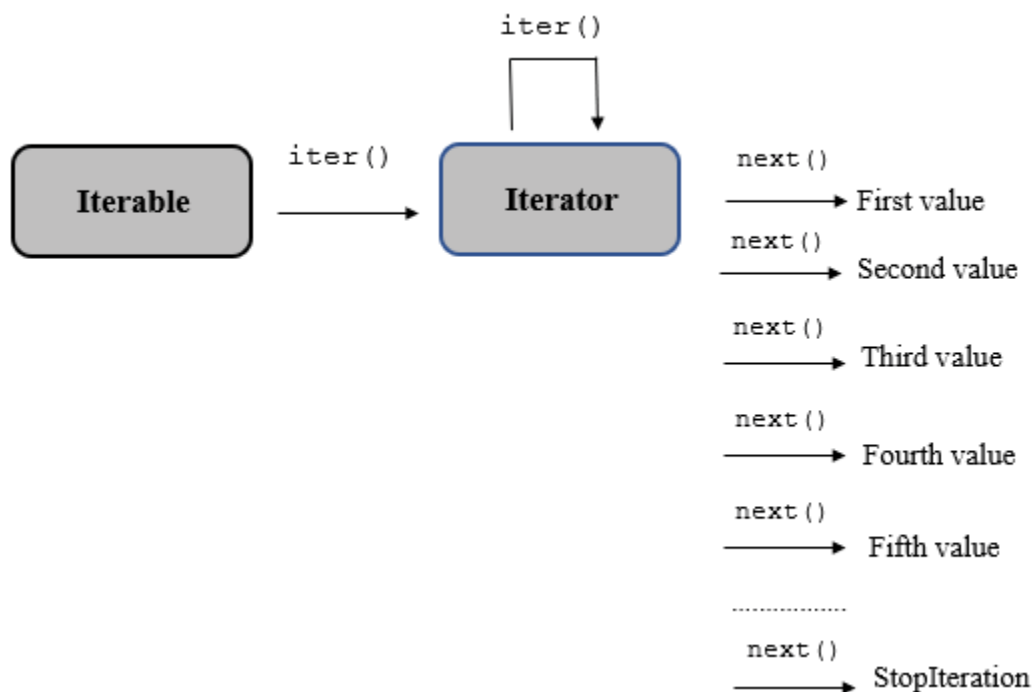


Figure 17.1: Iterable and iterator

We know that whenever we call the built-in function `len` on an object `x`, the `__len__` method of the object is invoked. If you call `reversed` function, then the `__reversed__` method of the object is invoked.

```
len(x)          x.__len__()
reversed(x)     x.__reversed__()
```

Similarly, when you call the `iter` function on an object, the `__iter__` method is invoked and when you call the `next` function, the `__next__` method is invoked.

```
iter(x)         x.__iter__()
next(x)         x.__next__()
```

So, we can say that any object is iterable if it implements the `__iter__` method which returns an iterator. An iterator is an object that implements the `__iter__` method which returns the iterator itself. An iterator also implements the `__next__` method that returns the next item from the data stream that the iterator represents. The `__next__` method should return the next item, and for performing this, an iterator object should maintain some sort of internal state to know which item it has to deliver next. The `__next__` method should raise the `StopIteration` exception when there are no more items to deliver.

If an object has `__iter__` method defined, then it can respond to the `iter` function and so, that object is an iterable. Iterators also have `__iter__` defined, so they are also iterables and can be used in any place where iterables are accepted.

There is one more thing that we need to know about the `iter` function. The `iter` function checks for the `__iter__` method and if it is not implemented, then it checks for the `__getitem__` method. If this method is found then also the object `x` is iterable and can be used in all iteration contexts. These methods all together form the Iterator protocol. Let us see some examples now:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
```

We have a list and its iterator:

```
>>> dir(L)
['__add__', '__class__', '__class_getitem__',
 '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__',
 '__gt__', '__hash__', '__iadd__', '__imul__',
 '__init__', '__init_subclass__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse',
 'sort']
```

We can see the `__iter__` method in the output of `dir(L)`. There is no `__next__` method because list is not an iterator, it is just an iterable.

```
>>> dir(it)
['__class__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__iter__',
 '__le__', '__length_hint__', '__lt__', '__ne__',
 '__new__', '__next__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__setstate__', '__sizeof__', '__str__',
 '__subclasshook__']
```

In the output of `dir(it)`, we can see both `__iter__` and `__next__` methods.

In the output of `dir(L)`, we can see that `__getitem__` method is also present; it gives the object the indexing capability of a sequence. The `iter` function will look for the `__iter__` method, since it is present here, it will not call the `__getitem__` method. The `__getitem__` method is called by the `iter` function only if the `__iter__` method is not available.

17.3 for loop Iteration Context – How for loop works

After learning about iterators and iterables, we are now in a position to learn about how the `for` loop of Python works behind the scenes.

```
for item in iterable:
```

```
.....
```

```
.....
```

This is the syntax of a `for` loop, and we know that it can iterate over any iterable object. The loop works by taking items one by one from the iterable object, and the loop body is executed once for each item. We have used the `for` loop to iterate over various different objects. It can iterate over non-sequence types such as sets and it can iterate even over the objects returned by some methods and functions. Here are some examples of `for` loop usage:

```
d = {1:'a', 2:'b'}
```

```
for key in d:
```

```
.....
```

```
for val in d.values():
```

```
.....
```

```
for key, val in d.items():
```

```
.....
```

```
for item in [1, 2, 3]:
```

```
.....
```

```
for item in (1, 2, 3):
```

```
.....
```

```
for ch in 'abcd':
```

```
.....
```

```

for element in {5, 3, 1}:
    .....
for number in range(2, 20, 3):
    .....
for i, val in enumerate([10, 20, 30]):
    .....
for x, y in zip([1, 2, 3], 'abc'):
    .....
for line in open('data.txt', 'r'):
    .....

```

The objects that we have used in these examples after the `in` keyword are of different types, but the `for` loop works uniformly for all of them. The usage of the loop remains the same irrespective of the type and implementation of the object. Later, we will see that we can use this loop to iterate over our own user-defined type objects also. Now let us try to understand how this `for` loop is able to achieve this abstraction.

When the interpreter sees a `for` loop, it calls the `iter` function with the object that is written after the `in` keyword, and it gets an iterator. It, then, keeps calling the `next` function on that iterator till the `StopIteration` error is raised.

Consider this simple loop, which iterates over a list and prints each element of the list.

```

L = [1, 2, 3]
for item in L:
    print(item)

```

The explanation given below shows how this `for` loop works:

```

it = iter(L)           # Get the iterator
item = next(it) # Get a value from the iterator and assign it to the
loop variable

```

```

print(item)                # Execute the loop body (1 is printed)
item = next(it)            # Get a value from the iterator and
assign it to the loop variable
print(item)                # Execute the loop body (2 is printed)
item = next(it)            # Get a value from the iterator and
assign it to the loop variable
print(item)                # Execute the loop body (3 is printed)
next(it)    raises StopIteration

```

First `iter` is called on the list `L`, and an iterator object is obtained. Then the `next()` function is called, and the object returned by `next()` is assigned to the target variable. After this, Python executes the loop body for this value of `item`. Calling of `next()` on the iterator object and execution of loop body is continued till the iterator is exhausted, in which case the call to `next` raises a `StopIteration` exception. The exception `StopIteration` is handled, and the loop finishes normally.

If we write this whole thing using a `while` loop, it would look like this:

```

L = [1, 2, 3]
it = iter(L)
while True:
    try:
        item = next(it)
        print(item)
    except StopIteration:
        break

```

These `try` and `except` are keywords that we will see later on in [Chapter 20](#). Inside the `try` block, we write the statements that can raise an exception, and in the `except` block, we specify what happens when an exception is caught.

So, first, we get the iterator, and then in the infinite `while` loop, we keep on calling `next` and executing the loop body, and the loop breaks when the

exception is caught. There is no need for us to write such a loop because the `for` loop automates this whole process. It creates a temporary unnamed variable to hold the iterator object for the duration of the loop.

In our next `for` loop, we have a call to `items` method of the `dict` type.

```
d = {1: 'a', 2: 'b'}
for key, val in d.items():
    print(key, val)
```

The `items` method of `dict` type returns an iterable object, so Python obtains an iterator by calling `iter` on the iterable object returned by this method, and then the same process of calling `next` is followed.

In the next `for` loop, we have a call to `enumerate` function:

```
for item in enumerate([10, 20, 30]):
    print(item)
```

The `enumerate` function returns an iterator, and we know that an iterator is iterable because it can respond to the `iter` function. Thus, the whole iteration mechanism that we have just seen works here too.

So, the `for` loop iterates over the iterator object that it gets by applying the `iter` function to the object placed after the `in` keyword. We can place an iterable or an iterator because both of them return an iterator when passed through `iter` function.

17.4 Iteration Tools

Iterators are used internally not only in the `for` loops, but in many other operations. Here are some iteration contexts in which Python expects an iterable object:

- `for` loops
- List, dictionary, and set comprehensions
- Tuple unpacking
- Unpacking actual parameters with `*` in function calls

- Sequence Assignment
- Slice assignment
- `in` and `not in` operators
- `extend` method of `list` class
- `join` method of `str` class
- `list`, `tuple`, or `set` built-in functions

Like `for` loops, comprehensions also work on iterable objects. Sequence unpacking, unpacking arguments in function calls, sequence, and slice assignment, `in` and `not in` operators, the `extend` method of the `list` class, the `join` method of `str` class all make use of iterators internally. `list`, `tuple`, or `set` built-in functions that are used to make new object from iterables also use iterators. The same iteration mechanism that is applied in a `for` loop is applied in these iteration tools also.

Here are some built-in functions that accept iterables and process them.

<code>any</code>	<code>all</code>	<code>max</code>	<code>min</code>
<code>sum</code>	<code>reversed</code>		
<code>sorted</code>	<code>zip</code>	<code>enumerate</code>	<code>map</code>
<code>filter</code>			

All these iteration tools accept iterables and use the iteration protocol to scan them. These iteration tools make a temporary iterator object like the `for` loop does, and use that iterator internally to iterate. All of them do the same thing, fetch an iterator with `iter` and then call `next` repeatedly till `StopIteration` occurs.

Later we will see how to make our own iterable objects that implement the iterator protocol, so we can use those objects also in all these iteration contexts.

17.5 Iterator vs Iterable

There are many built-in functions and methods that return iterables and iterators. Here are a few examples:

<code>range()</code>	returns an iterable
<code>dict.keys()</code>	returns an iterable
<code>dict.items()</code>	returns an iterable
<code>dict.values()</code>	returns an iterable
<code>enumerate()</code>	returns an iterator
<code>zip()</code>	returns an iterator
<code>reversed()</code>	returns an iterator
<code>open()</code>	returns an iterator
<code>map()</code>	returns an iterator
<code>filter()</code>	returns an iterator

Some of the functions and methods return iterables, and some return iterators. We need to know the type of object that we are dealing with (iterable or iterator), otherwise we will get unexpected results. Let us see why it is so.

We know that after an iterator is exhausted, it raises a `StopIteration` exception each time `next` is called on it. There is no way to reset or restart an iterator. If you need to iterate over the same data stream again, then you will have to create a new fresh iterator by calling `iter` function over the iterable from which you got the iterator.

So, an iterator becomes a useless throw-away object once it is exhausted. It is not possible to reset or restart an iterator. You need to get a fresh iterator if you need to iterate again. The iteration tools that we have seen in the previous section, work internally by calling `iter` on the iterable and so they get a fresh iterator automatically.

Suppose we have an iterable `x` and we use it in these 3 iteration contexts:

```
for i in x:
    pass
L = list(x)
n = max(x)
```

All these iteration tools will call `iter` on the iterable, get fresh iterators, and will use those iterators. This code will work as expected, and there is no

problem here.

Now suppose we have an iterator `y`. You know that iterators can respond to the `iter` function, which can be used in any iteration context. So, we use this iterator `y` in the same three places:

```
for i in y:
    pass
L = list(y)
n = max(y)
```

The `for` loop will work as expected, but the next two statements will not work as expected and they will not show any error also. This is because when an iterator is passed through the `iter()` function, you get the same iterator back. The `for` loop exhausted the iterator `y`, and when the `list` function called `iter` function on `y`, it got the same exhausted iterator back. When it called the `next` function on the exhausted iterator, a `StopIteration` error was raised, which was caught. Similarly, in the last statement, when `iter` will be called on `y`, `y` will be only returned which is already exhausted. An exhausted iterator is like an empty container.

From these three iteration contexts, whichever is written first will work, and the other two will not work. If you write the `max` function before the `for` loop and `list` function, then the `max` function will work but the other two will not work, because in that case the `max` function will consume the iterator and the other two will get the exhausted iterator.

So, you can iterate many times over an iterable, but you can iterate only once over an iterator.

This is because when an iterable is passed to `iter` function, it returns a fresh iterator while when an iterator is passed to `iter` function it returns the same iterator.

If a function returns an iterable object, then that object supports multiple iterations, while the iterator objects support just one iteration. This is because an iterable can be used to get a fresh iterator every time `iter` is used on it. But for an iterator, the same iterator will be returned every time `iter` is used on it.

Lists and other containers are iterables so they also support multiple iterations. Let us understand this with the help of an example:

```
>>> x = range(3, 0, -1)
>>> y = reversed([1, 2, 3])
>>> iter(x)
<range_iterator object at 0x00000203EF60CE50>
>>> iter(y)
<list_reverseiterator object at
0x00000203EFB7B1F0>
```

We have called the `range` and `reversed` functions and they returned objects `x` and `y` respectively. Both these objects return an iterator when passed to the `iter` function. Since they respond to `iter`, we can use these objects in for loops.

```
>>> for i in x:
...     for j in x:
...         print(f'i={i}, j={j}')
```

We have used `x` in this nested loop structure. In the outer loop, `i` will take values 3, 2 and 1 because

`range(3, 0, -1)` represents these values. In the inner loop also, we have used `x`, so for each iteration of the outer loop, `j` will take values 3, 2 and 1. Here is the output that we get:

```
i=3, j=3
i=3, j=2
i=3, j=1
i=2, j=3
i=2, j=2
i=2, j=1
i=1, j=3
i=1, j=2
```



```
i=1, j=1
```

Now, let us write the same nested `for` loop for the object `y` returned by the `reversed` function.

```
>>> for i in y:
...     for j in y:
...         print(f'i={i}, j={j}')
```

We expect the same output here because the reverse of 1,2,3 is 3,2,1 and hence the object `y` represents the values 3, 2, 1. We get a different output, now, which is as follows:

```
i=3, j=2
i=3, j=1
```

The reason for this output is that the function `reversed` returns an iterator.

```
>>> y is iter(y)
True
```

The expression `y is iter(y)` returns `True`, which means that `y` is an iterator.

```
>>> x is iter(x)
False
```

The expression `x is iter(x)` returns `False`, which means that `x` is an iterable, not an iterator.

The `range` function returned an iterable, which gives a fresh iterator whenever `iter` is called on it. The `reversed` function returned an iterator which returns itself when `iter` is called on it.

Let us see what happens in the nested `for` loops. First, let us see what happens when we use `x`. In the outer `for` loop, `iter(x)` is called, which gives out a fresh iterator, then `iter(x)` is called in the inner `for` loop, and it also gives a fresh new iterator. Both the iterators were separate, and they maintained their own state of progress.

Now, let us see what happened when we used `y` in the nested loops.

In the outer loop, `iter(y)` was called, and it returned iterator `y`; the function `next` was called on iterator `y` which returned 3, so `i` was assigned value 3. For the first iteration of the outer `for` loop, the inner loop will execute. So, in the inner loop, `iter(y)` is called, and it returns `y` itself. Now `next` is called on `y`, and the next value for iterator `y` is 2, because it has already returned 3. So, `j` is assigned value 2. Again, the inner loop calls `next` on `y`, and this time, it returns 1, which is assigned to `j`. Next time, when the inner loop calls `next` on `y`, the `StopIteration` error is raised, and the inner loop terminates.

The control goes to the outer loop, and it resumes execution, so it calls `next` on iterator `y`, but iterator `y` is already exhausted, so it just raises the `StopIteration` exception, and the outer loop also terminates. This is why we get a different output.

When we used `x` in the loops, both the `for` loops were working with separate iterators so there was no problem.

These types of surprising results can also occur in other iteration contexts if you are not aware of the type of the object you are dealing with. Let us see one more example:

Again, we take these two objects returned by `range` and `reversed` functions.

```
>>> x = range(3, 0, -1)
>>> y = reversed([1, 2, 3])
>>> list(x)
[3, 2, 1]
>>> list(y)
[3, 2, 1]
```

We created a list from object `x` and also from object `y`. Now we will create tuples from these objects.

```
>>> tuple(x)
(3, 2, 1)
>>> tuple(y)
( )
```

`tuple(y)` gives an empty tuple and this is because the object `y` is an iterator which was exhausted by the `list` function. So, again we saw that the object `x` returned by `range` supports multiple iterations, while the object `y` returned by `reversed` supports only a single iteration.

Now, suppose we have a file object returned by the `open` function:

```
>>> f = open('data.txt', 'r')
```

We can create a list from this file object:

```
>>> list(f)
['first line\n', 'second\n', 'third\n',
'fourth\n']
```

We get a list that contains all the lines of this file. After this, we use this file object in a `for` loop.

```
>>> for line in f:
        print(line)
```

This `for` loop will not give any output, because the file object is an iterator object and therefore supports only a single iteration.

```
>>> f is iter(f)
True
```

This is `True`, which confirms that this file object is an iterator.

The object returned by the `open` function is an iterator, so it is possible for us to iterate over the contents of the file in chunks without storing the whole file in memory. This is beneficial if the file is very large and will occupy a lot of space if stored in a data structure like a list.

In this section, we saw that for an iterable object, multiple active scans are supported because there can be different active iterators associated with that iterable. For an iterator, only a single scan is supported because you have only a single iterator. An iterator can be used for only one pass over the set of values.

17.6 Creating your own Iterator

In this section, we will see how to write classes to create our own iterable objects. In the last section, we saw the difference between objects that support multiple active iterations and objects that support only a single iteration. When we write our own class for creating an iterable object, we have to decide whether we want our objects to support multiple active iterations or a single iteration. First, we will see a few examples of writing classes that create iterable objects that support multiple iterations.

In our first example, we will create iterable objects, which, when iterated over, will give out cubes of numbers, and these objects will support multiple iterations.

```
class Cubes:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
    def __iter__(self):
        return CubesIterator(self)

class CubesIterator:
    def __init__(self, source):
        self.source = source
        self.current = source.start
    def __next__(self):
        if self.current > self.source.stop:
            raise StopIteration
        else:
            x = self.current
            self.current += 1
            return x * x * x

x = Cubes(2,8)
for i in x:
    print(i, end=' ')
print('Sum =',sum(x))
```

Output-

```
8 27 64 125 216 343 512 Sum = 1295
```

The instance objects that are created from class `Cubes` will be iterable objects. `x` is an instance object of `Cubes` class, so it is an iterable object that represents cubes of numbers from 2 to 8. We can use it in any iteration tool; in the program, we have used it in a `for` loop and the `sum` function. The `for` loop will print the cubes of all the numbers from 2 to 8, and the `sum` function will return the sum of all the cubes from 2 to 8.

Let us understand the code for the class `Cubes`. The initializer method takes two arguments, and inside it, we have created two instance variables `start` and `stop`. The `__iter__` method should return a fresh iterator object every time it is invoked. So, inside this method we create and return an instance of the class `CubesIterator`. Now, let us understand the code for the class `CubesIterator`.

The `__init__` method has a parameter named `source` which will accept the iterable that has to be iterated over. In the `__iter__` method of the `Cubes` class, when we create an instance of this `CubesIterator` class, we have sent `self` as an argument, which is the iterable that needs to be iterated over.

Inside the `__init__` method, we have written `self.source = source`, so now this iterator class has access to all the instance variables of the object named `source`. Next we create an instance variable `current` that is set equal to `source.start`. Now, let us see the `__next__` method.

If the `current` is greater than the `stop` of the `source` object, then a `StopIteration` exception will be raised. If it is not, then the cube of the current number will be returned, and the value of the `current` will be increased by 1. So, this class will create stateful iterator objects that store the current state. The state in this iterator is kept inside the instance variable `current`. When the method `__next__` is called, it produces and returns the result for the current call and it also modifies the state for the next call. So, you can think of an iterator as a value factory; whenever you request the next value from it, it knows how to compute it because it holds the current internal state. It remembers its state between calls.

In our next example, we have created a class named `Fibonacci` which will create iterable objects that give out Fibonacci numbers up to a certain value. In Fibonacci series, each number is the sum of previous two numbers.

```
# Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89
```

```
class Fibonacci:
    def __init__(self, max):
        self.max = max
    def __iter__(self):
        return FiboIterator(self)

class FiboIterator:
    def __init__(self, source):
        self.source = source
        self.a = 0
        self.b = 1
    def __next__(self):
        f = self.a
        if self.a > self.source.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return f

x = Fibonacci(100)
for i in x:
    print(i, end=' ')
print(55 in x, 50 in x)
```

Output-

```
0 1 1 2 3 5 8 13 21 34 55 89 True False
```

We have created an instance object of class `Fibonacci` and used it in a for loop and in the `in` operator. The class `Fibonacci` has two methods `__init__` and `__iter__`. The `__init__` method takes an argument

and creates an instance variable `max` which denotes the number up to which we want to generate the Fibonacci numbers. The `__iter__` method returns an iterator object of the class `FiboIterator`. This is the class that maintains the state information.

In the `__init__` method of `FiboIterator`, we have created three instance variables. `source` is the iterable that has to be iterated over. The state inside the iterator is maintained with the help of instance variables `a` and `b`. The instance variable `a` is initialized to 0 and `b` to 1.

The `__next__` method is responsible for calculating and returning the next term of the series.

First, we save the value of `self.a` in a variable named `f`, because we will return this value at the end. If `self.a` is more than `self.source.max`, then the `StopIteration` exception is raised. Otherwise, we have the statement `self.a, self.b = self.b, self.a + self.b` that updates the values of instance variables `a` and `b`. Variable `self.a` is made equal to `self.b`, and `self.b` is made equal to `self.a + self.b`, and then `f` is returned. When the method `__next__` will be called, the current Fibonacci number will be returned and the state is also modified for the next call of `__next__`.

In the two examples that we have seen, we created two classes each: the iterable class and the iterator class. The iterator class needs to access the data members of the iterable class, and that is why the iterable class passes a reference of its current object to the initializer of the iterator class. Instead of passing reference to the object, you can simply pass the instance variable that will be needed by the iterator. For example, in `FiboIterator` you could pass `self.max`, and in `CubesIterator`, you could pass `self.start` and `self.stop`. It will also work, but it is better to pass the `self` object instead of passing individual instance variables. When you pass the source object, in the iterator you only have to create instance variables that are responsible for maintaining the state.

We saw two examples where we created objects that supported multiple active iterations. If we do not want our objects to support multiple iterations then the whole thing can be put inside a single class.

```
class Fibonacci:
```

```

def __init__(self, max):
    self.max = max
    self.a = 0
    self.b = 1
def __iter__(self):
    return self
def __next__(self):
    f = self.a
    if f > self.max:
        raise StopIteration
    self.a, self.b = self.b, self.a + self.b
    return f
x = Fibonacci(100)
for i in x:
    print(i, end = ' ')
print(50 in x, 55 in x)

```

Output-

```
0 1 1 2 3 5 8 13 21 34 55 89 False False
```

In this program, we have only a single class, and its objects will support a single iteration.

The `__iter__` method now returns `self` instead of returning a fresh iterator. The `__next__` method is now written in this class itself instead of a separate class. The state-maintaining variables `a` and `b` are also a part of this class. Now, we do not need a separate iterator class; this class is its own iterator.

From the output, we can see that object `x` supports only a single iteration. The `in` operator did not work because after the `for` loop, the iterator was exhausted. When the `in` operator demanded an iterator, it got the same exhausted iterator because now the `__iter__` method returns `self` instead of returning a fresh iterator.

This way, we can write a single class for both the iterator and the iterable. This type of class will create an iterator object that supports only a single iteration. Let us write a single iterator class for the Cubes example that we have seen.

```
class Cubes:
    def __init__(self, start, stop):
        self.current = start
        self.stop = stop
    def __iter__(self):
        return self
    def __next__(self):
        if self.current > self.stop:
            raise StopIteration
        x = self.current
        self.current += 1
        return x*x*x

x = Cubes(2,8)
for i in x:
    print(i, end = ' ')
print('Sum =',sum(x))
```

Output-

```
8 27 64 125 216 343 512 Sum = 0
```

Previously, we made two classes to support multiple iterations, but now, since we have put everything in the same class, only a single iteration is supported. The `for` loop exhausted the iterator, and hence the `sum` function did not work.

So, when you need to create your own iterator, you can define a class that has the iterator interface, a class that has `__iter__` and `__next__` methods.

In the `__next__` method of the `Cubes` class, if you remove the condition `if self.current > self.stop` then this class will give you

infinite iterators.

```
class Cubes:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        x = self.current
        self.current += 1
        return x * x * x

x = Cubes(2)
print(next(x))
print(next(x))
print(next(x))
x = Cubes(2)
for i in x:
    if i > 150:
        break
    print(i, end=' ')
```

Output-

```
8
27
64
8 27 64 125
```

Now there is no need for the variable `stop`. Now `x` is an iterator that can provide values infinitely. If you execute the `for` loop that we had in our previous program, then it will not terminate because the iterator will keep on providing values, and the loop becomes infinite. Even the function `sum` will get stuck if you run it with this iterator.

We can get the values manually from this iterator:

```
>>> next(x)
8
>>> next(x)
27
>>> next(x)
64
```

The `next` function will keep on giving the cubes. When you have to use an infinite iterator in a `for` loop, you have to add a terminating condition with `break` inside the loop because this iterator will never stop giving values.

17.7 Making your class Iterable

The instance objects created from the custom classes that we write are not iterable by default. We cannot use them in any iteration context. For example, we have a `Stack` class that is an implementation of stack data structure which is a last in first out (LIFO) data structure.

The instance objects of this class are not iterable, which means that if we have an object of this class in any iteration context, it will not work.

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def size(self):
        return len(self.items)
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if self.is_empty():
            raise RuntimeError("Stack is empty")
```

```
        return self.items.pop()
    def display(self):
        print(self.items)
```

The interpreter does not know how to iterate over objects of this class. If we want objects of this class to be used in iteration contexts, then they should support the iteration protocol. To make this class iterable, we can add a `__iter__` method to this class.

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def size(self):
        return len(self.items)
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if self.is_empty():
            raise RuntimeError("Stack is empty")
        return self.items.pop()
    def display(self):
        print(self.items)
    def __iter__(self):
        return iter(self.items)

stack = Stack()
stack.push(20)
stack.push(30)
stack.push(10)
stack.push(89)
for item in stack:
```

```
        print(item)
print(min(stack), max(stack), sum(stack))
```

Output-

```
20
30
10
89
10 89 149
```

We know that the `__iter__` method should return an iterator. In this class, the items are internally held in the list named `items`. Inside the `__iter__` method, we have called the built-in function `iter` on `items` list. This call returns an iterator on the list and we return it from the method. So, the `__iter__` method returns an iterator that iterates over the `items` list. This means that the iteration was actually delegated to the `items` list. Now, we can use instance objects of the `Stack` class in iteration contexts. Whenever we will iterate over an instance object of this class, the iteration will actually happen over the `items` list.

In the program, we have used an instance object of this class `Stack` in a for loop and the built-in functions `min`, `max`, and `sum`. When we are iterating over an object of `Stack` class, the iteration actually happens over the contained list. The iteration request is passed to the list, named `items`. A list is an iterable object, so we need not worry about how the next item will be returned. If you have written a container class that internally contains an iterable like a list or a tuple, then you can make the instance objects of your class iterable by simply adding a `__iter__` method to the class. Inside the `__iter__` method, you can delegate the iteration to the contained iterable.

In these types of cases where you are just delegating the iteration to the contained iterable, you need not worry about how the next element is coming, there is no need to write the `__next__` method, because the contained iterable knows how to deliver the next element.

When things are not as straightforward as in our `Stack` example, we need to write the `__next__` method also. In such cases, it is better to create a

separate iterator class. Let us see this with the help of an example.

We have the following class in which we have two lists named `grocery_items` and `stationery_items`, each list contains a tuple of item and quantity. This instance objects of this class are not iterable,

```
class Cart:
    def __init__(self):
        self.grocery_items = []
        self.stationery_items = []
    def add_stationery(self, item, quantity=1):
        self.grocery_items.append( (item,
quantity) )

    def add_grocery(self, item, quantity=1):
        self.stationery_items.append( (item,
quantity) )
cart = Cart()
cart.add_grocery('rice')
cart.add_stationery('pen',3)
cart.add_stationery('eraser')
cart.add_stationery('pencil',5)
cart.add_grocery('bread',2)
cart.add_grocery('pasta')
```

We want to make this class iterable. We want that when any object of `Cart` is iterated over, first it should return elements of `grocery_items` list one by one and then elements of `stationery_items` list one by one. To make this class iterable we will add a `__iter__` method that will return an object of `CartIterator` type.

```
def __iter__(self):
    return CartIterator(self)
```

Here is the code for the class `CartIterator`:

```
class CartIterator:
```

```

    def __init__(self, source):
        self.source = source
        self.i = 0

    def __next__(self):
        if self.i >=
(len(self.source.grocery_items) +
len(self.source.stationery_items)):
            raise StopIteration

        if self.i < len(self.source.grocery_items):
            item =
self.source.grocery_items[self.i]
        else:
            item =
self.source.stationery_items[self.i
- len(self.source.grocery_items)]

        self.i += 1

        return item

```

In the `__init__` method, we have the source and we create an instance variable `i` for maintaining the current state of the iterator. This class contains the `__next__` method, which returns the next item. On iterating, we want items from both the contained lists. If value of `i` is more than the sum of lengths of both these lists, then it will mean that no more items are left so a `StopIteration` error is raised. First the items are returned from the grocery list, then from the stationery list. The variable `item` is initialized accordingly and returned, and the value of `i` is increased for the next time.

Now, the objects of class `Cart` have become iterable, and we can use them in a `for` loop like this:

```

for item, quantity in cart:
    print(item, quantity)

```

Output-

```
pen 3
eraser 1
pencil 5
rice 1
bread 2
pasta 1
```

So, to add iterator behavior to your class, you need to define a `__iter__` method that returns an object with a `__next__` method. If the class itself defines the `__next__` method, then `__iter__` can simply return `self`.

17.8 Some More Iterators

In this section, we will write three iterator classes that can be used to iterate a sequence in reverse order, in repeated cyclic order and in alternate order. Let us start by writing an iterator that iterates over a sequence in such a way that each alternate item is returned:

```
class Alternate:
    def __init__(self, source):
        self.source = source
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index >= len(self.source):
            raise StopIteration
        item = self.source[self.index]
        self.index = self.index + 2
        return item

L = [1, 2, 3, 4, 5, 6, 7, 8]
for i in Alternate(L):
```



```
    print(i, end=' ')
for ch in Alternate('intelligent'):
    print(ch, end=' ')
```

Output-

```
1 3 5 7 i t l i e t
```

We have three methods in this iterator class, `__init__`, `__iter__` and `__next__`. The `__init__` method has a parameter named `source` which is the sequence that will be iterated. We create an instance variable for the `source`, then we create an instance variable named `index` and initialize it to 0. This is the variable that will be responsible for maintaining the state. The `__iter__` method just returns `self`. In the `__next__` method, we raise `StopIteration` error if the `index` becomes greater than the length of `source`. Otherwise, we save the element at the current `index` in the variable `item` and after that we increment `index` by 2 because we want every alternate item from the source iterable. At last, we return `item`.

Next, we have the iterator class for iterating a sequence in reverse order.

```
class Reverse:
    def __init__(self, source):
        self.source = source
        self.index = len(source)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.source[self.index]

L = [1, 2, 3, 4, 5, 6, 7, 8]
for i in Reverse(L):
    print(i, end=' ')
```

```
for ch in Reverse('intelligent'):
    print(ch, end=' ')
```

Output-

```
8 7 6 5 4 3 2 1 t n e g i l l e t n i
```

Here also we take the source in the initializer, which is the sequence that we want to iterate. The instance variable `index` is initialized to the length of source. The method `__iter__` returns `self`. In the `__next__` method, `StopIteration` is raised if the `index` becomes zero. Otherwise, the `index` is decremented, and the element at the current `index` is returned.

Here is our last example. This class can be used to iterate the sequence repeatedly.

```
class Cycle:
    def __init__(self, source):
        self.source = source
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.index = (self.index + 1) %
(len(self.source))
        return self.source[self.index]
count = 0
for i in Cycle([1,2,3]):
    print(i, end= ' ')
    count+=1
    if count == 10:
        break
```

Output-

```
1 2 3 1 2 3 1 2 3 1
```

This iterator will give out elements infinitely, because `StopIteration` is never raised. It is an infinite iterator so while using it in a loop, you have to be careful about adding the terminating condition.

All three iterators that we saw will support only a single active iteration; if you want to support multiple active iterations, you will have to write two classes. Let us write another class for the Reverse iterator.

```
class Reverse:
    def __init__(self, source):
        self.source = source
    def __iter__(self):
        return ReverseIterator(self.source)

class ReverseIterator():
    def __init__(self, source):
        self.source = source
        self.index = len(self.source)
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.source[self.index]

x = Reverse('Knowledge')
for c in x:
    print(c, end = ' ')
for c in x:
    print(c, end = ' ')
```

Output-

```
e g d e l w o n K e g d e l w o n K
```

Now, we can see that multiple iterations will be supported. There are lots of readymade iterators that are available in the `itertools` module, which we will see later in this chapter.

17.9 Lazy evaluation

Lazy evaluation is an evaluation strategy in which the evaluation of an expression is delayed until its value is actually needed. The evaluation is postponed until it is demanded.

Iterators are lazy; they perform lazy evaluations. Iterators do not perform any work until we ask them to provide the next value. After providing the next value, they again go idle and come into action only when they are asked to provide the next value again. So, whenever the next value is asked, an iterator computes it and returns it. You can think of an iterator as a lazy factory of values which produces one value at a time when required.

For example, suppose we have an iterator that returns squares of numbers from 1 to 100:



Figure 17.2: Calling next() on an iterator

When we call the method `next` on this iterator, it returns a square and then sits idle. Again, when we ask for a value, it gives a square and then sits idle. It produces the values only when asked. It did not compute all the squares at once and stored them in memory.

An iterator evaluates and provides only one value at a time, so there is no need to allocate memory for the whole dataset that the iterator represents. If you do not require all the data at once, you can get the next item from the dataset without keeping the whole dataset in memory.

This delayed evaluation strategy, results in saving memory resources. This laziness of iterators also makes them suitable to represent infinite datasets. You cannot write a list or a set that has infinite data, but you can write iterators that produce an infinite stream of data. So, iterators are useful when you need to scan datasets that are too big to fit in memory or when you need infinite series of data that is not possible to store in a collection.

For very large data sets, lazy evaluation also avoids intermediate pauses that occur due to the whole thing being computed at once. The computation time is divided because things are computed only on request. For example, suppose you have 1 million data values to be processed, if you create a list then it would take lot of space in memory and will also take lot of time to compute. If you use an iterator, then the values will be produced one at a time, and so you do not have to worry about the memory running out or the program getting halted due to long computation.

Thus, iterators save memory resources and execution time, and they can represent infinite datasets. The client code gets access to an object that can give infinite or very large number of values, but pays only for the computation costs of the values that it actually uses.

The `range` function used to return a list of numbers in earlier versions, but in Python 3 it returns an iterable to save memory. That is why we had to wrap the results of `range` function in a list to see all the values at once.

The opposite of lazy evaluation is strict or eager evaluation, where there is no delay in the evaluation of expressions. For example, the following list comprehension computes the whole list of cubes at once and stores it in memory.

```
cubes = [x*x for x in range(1,1001)]
```

Another advantage of iterators is that they provide a common interface to traverse different types of objects in the same way, irrespective of their internal structure and implementation.

17.10 itertools Module

The `itertools` module contains special functions to create iterators for efficient looping. It has three broad categories of iterators - infinite iterators, iterators terminating on the shortest input sequence and combinatoric iterators. We will see the usage of some of them in the upcoming sections:

Infinite iterators

```
count()    cycle()    repeat()
```

Iterators terminating on the shortest input sequence

```
accumulate()      chain()          compress()
dropwhile()       filterfalse()    groupby()   islice()
pairwise()        takewhile()      tee()
zip_longest()
```

Combinatoric iterators

```
combinations()      combinations_with_replacement()
permutations()       product()
```

First, we will use infinite iterators. These iterators produce infinite sequences. When used inside a loop, they should be used in such a way that they stop at some point, otherwise we will be stuck in an infinite loop.

```
>>> from itertools import count
```

We have imported the `count` function from the `itertools` module. This function gives an iterator that will keep on counting infinitely.

```
>>> it = count(start=10, step=3)
```

This iterator will start from 10 and keep adding 3 to the previous value.

```
>>> next(it)
```

```
10
```

```
>>> next(it)
```

```
13
```

```
>>> next(it)
```

```
16
```

If we use it in a `for` loop, it will go on infinitely.

```
>>> for i in count(10, 3):
```

```
...     print(i)
```

```
...
```

```
10
```

```
13
```

```
16
```

```
19
```

```
..
```

```
..
```

We need to put a `break` statement in the loop to stop it.

```
>>> for i in count(10,3):
```

```
...     if i > 30:
```

```
...         break
```

```
...     print(i)
```

```
10
```

```
13
```

```
16
```

```
19
```

```
22
```

```
25
```

```
28
```

Next, let us see the `cycle` function that produces an infinite sequence from a finite sequence.

```
>>> from itertools import cycle
```

This function cycles through the items of its iterable argument. It is also an infinite iterator.

We will call this function with a list as the argument.

```
>>> colors = cycle(['Red', 'Blue', 'Green'])
```

```
>>> next(colors)
```

```
'Red'
```

```
>>> next(colors)
```

```
'Blue'
```

```
>>> next(colors)
```

```
'Green'
```

```
>>> next(colors)
'Red'
```

This will keep on giving Red Blue Green Red Blue Green.

The `repeat` function from the `itertools` module returns an iterator which produces the object for the specified number of times. If the number of times is not specified, then it produces the object infinitely.

```
>>> from itertools import repeat
>>> x = repeat('Red', 4)
>>> next(x)
'Red'
>>> next(x)
'Red'
>>> next(x)
'Red'
>>> next(x)
'Red'
>>> next(x)
StopIteration
```

If we do not specify the second argument, then it will keep on returning 'Red' infinitely.

These were the functions that created infinite iterators, we have the other category of functions which create iterators that terminate on the shortest input sequence. First let us see the `chain` function from this category. This function chains different iterables.

```
>>> from itertools import chain
>>> L = [1, 2]
>>> s = {7, 8, 9}
>>> for i in chain(L, s):
...     print(i, end = ' ')
1 2 8 9 7
```


This function chained together the two iterables, list `L` and set `s`.

If we need a portion of the sequence represented by an iterator, the normal slicing operator cannot be used. We have to use the `islice` function that can produce a finite sequence from an infinite sequence.

```
>>> from itertools import cycle, islice
>>> fruits = cycle(['Apple', 'Mango', 'Banana',
'Grapes'])
```

We get an infinite sequence by calling the `cycle` function. Now, we will use the `islice` function to get a sliced iterator which we will use in a `for` loop.

```
>>> for x in islice(fruits,5,10):
...     print(x, end=' ')
```

Mango Banana Grapes Apple Mango

The `accumulate()` function calculates the accumulated values. By default, it calculates the sum.

```
>>> from itertools import accumulate
>>> L = [1, 2, 3, 4, 5, 6]
>>> for i in accumulate(L):
...     print(i, end = ' ')
1 3 6 10 15 21
```

You can also provide a function as the second argument if you do not want the default addition. The function that you provide should take 2 arguments and return a single value.

Next, we have the `zip_longest()` function which is like the `zip` built-in function but it can also work with iterables that are of different lengths.

```
>>> from itertools import zip_longest
>>> L1 = [1, 2, 3]
>>> L2 = [10, 20, 30, 40, 50]
>>> list(zip(L1, L2))
[(1, 10), (2, 20), (3, 30)]
```

```
>>> list(zip_longest(L1, L2))
[(1, 10), (2, 20), (3, 30), (None, 40), (None,
50)]
```

We have two lists of different sizes. The `zip` function stops when the smaller iterable is over. The `zip_longest` function goes on till the end of the longest iterable, and the empty places are filled with `None`. If we want, we can specify a fill value.

```
>>> list(zip_longest(L1, L2, fillvalue=0))
[(1, 10), (2, 20), (3, 30), (0, 40), (0, 50)]
```

Now all the empty places are filled with 0 instead of `None`.

Next, let us see the `permutations` function that gives all the permutations of elements in a given iterable.

```
>>> from itertools import permutations
>>> print(list(permutations([1, 2, 3, 4], 2)))
[(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4),
(3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3)]
>>> print(list(permutations([1, 2, 3, 4], 3)))
[(1, 2, 3), (1, 2, 4), (1, 3, 2), (1, 3, 4), (1,
4, 2), (1, 4, 3), (2, 1, 3), (2, 1, 4), (2, 3, 1),
(2, 3, 4), (2, 4, 1), (2, 4, 3), (3, 1, 2), (3, 1,
4), (3, 2, 1), (3, 2, 4), (3, 4, 1), (3, 4, 2),
(4, 1, 2), (4, 1, 3), (4, 2, 1), (4, 2, 3), (4, 3,
1), (4, 3, 2)]
```

The two calls give us all the permutations of length 2 and length 3.

There are other functions also that we have not discussed here, you can check the full list with their details on the Python website. So, before writing an iterator, check if something like that is available in `itertools` module.

17.11 Generators

The task of implementing iterators can be simplified by using generators. We have seen how to create custom iterators using the object-oriented way, i.e., by defining a class that has `__init__`, `__next__`, and `__iter__` methods. For example, we saw the `Cubes` class which when instantiated created an iterator object that gave out cubes of numbers. These types of simple iterators can be implemented in a much easier way by writing generators.

There are two types of generators: generator functions and generator expressions. Both of them are used to create generator objects which are actually iterators. A generator object is a kind of iterator, and we get a generator object by writing a generator function or a generator expression.

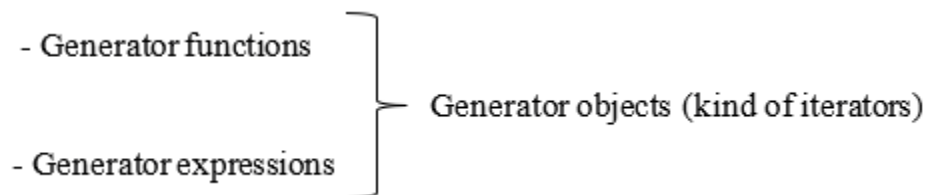


Figure 17.3: Generators

When you write a generator, you do not need to worry about writing the `__iter__` and `__next__` methods. You get the iterator interface automatically. So, when you want to get an iterator without writing a class, you can write generators. In fact, writing a class to define your own iterator is very rare. Generally, the automated syntax of generators is preferred to get your own iterators. However, if you need to create complex iterators or need to give access to some extra attributes and methods, then you will have to write class-based iterators. Now we will start with generator functions:

```
class Cubes:
    def __init__(self, start, stop):
        self.current = start
        self.stop = stop
    def __iter__(self):
        return self
    def __next__(self):
```

```

        if self.current > self.stop:
            raise StopIteration
        x = self.current
        self.current += 1
        return x*x*x
x = Cubes(2,5)

```

We have seen this class before; the instance object of this class is an iterator, which gives out cubes of numbers. Now, we will write a generator function that will give us a generator object that is similar to the iterator `x`.

```

>>> def cubes(start, stop):
...     for n in range(start, stop+1):
...         yield n * n * n
>>> y = cubes(2, 5)

```

Right now, do not worry about what is written inside the function. What you need to understand is that when we call this function, it returns a generator object, which is actually an iterator.

```

>>> y
<generator object cubes at 0x000001E3263A75B0>
>>> y is iter(y)
True

```

By writing this generator function, we were able to get an iterator without worrying about any of the methods needed to satisfy the iterator protocol. They are automatically implemented for us. If we call the `dir` function for the object `y`, we can see the `__iter__` and `__next__` methods.

```

>>> dir(y)
[ '.....', '__iter__', '.....', '__next__', '.....' ]

```

We can use the `next` function to get values from this generator object.

```

>>> next(y)
8
>>> next(y)

```

```
27
```

```
>>> next(y)
```

```
64
```

```
>>> next(y)
```

```
125
```

```
>>> next(y)
```

```
StopIteration
```

The calls to `next` function give us the cubes, and the `StopIteration` error was raised to signify the end of data. This generator object `y` behaves just like the iterator object `x` (instance of `Cubes`) would have behaved.

Generators automatically implement the iterator protocol and that is why they can be used in any iteration context. Let us use the object `y` in a `for` loop:

```
>>> for i in y:
```

```
...     print(i, end = '  ')
```

This loop will not give us any output because the generator object `y` is an iterator, and it cannot be used, once it is consumed. We exhausted this iterator when we used it in the `next` function, so this loop got an exhausted iterator. Let us call the generator function again and get a new iterator:

```
>>> y = cubes(2, 5)
```

```
>>> for i in y:
```

```
...     print(i, end = '  ')
```

```
8  27  64  125
```

Now, the `for` loop works. So, these generator objects cannot be used for multiple active iterations.

We saw that when a generator function is called it gives us a generator object which is an iterator. In the next section, we will see how to write a generator function and the differences between a generator function and a normal function.

17.12 Generator function vs Normal function

To understand what generator functions are, we will compare them with the normal functions that we already know. Both of them are defined by using the `def` statement. Python considers a function as a generator function if one or more `yield` statement appears inside the function. The `yield` statement consists of the `yield` keyword followed by a value. So, if you see a `yield` statement inside a function, then it is a generator function.

Normal Function	Generator Function

<code>def fn():</code>	<code>def gen_fn():</code>
<code>.....</code>	<code>.....</code>
<code>.....</code>	<code>yield value</code>
<code>.....</code>	<code>.....</code>

Now let us see how both of them behave when they are called. We know that when we call a normal function, the code inside the function is executed and the value that is there in the `return` statement is given out, and if there is no `return` statement, `None` is given out.

When you call a generator function, the code written inside the function is not executed, instead a generator object is given out which can be assigned to a variable. Calling a generator function does not execute the function's code, it just creates and returns a generator object. We have seen that this generator object is actually an iterator and it can be iterated over.

```
x = fn()      # Calling a normal function executes
its code, returns a value

g = gen_fn()  # Calling a generator object gives a
generator object
```

The code of a generator function is not executed when the generator function is called, so when and how is this code executed? This code is executed when you iterate over the generator object either automatically or manually. You can iterate automatically by using any iteration tool like a `for` loop or a comprehension and you can iterate manually by using the `next` function.

So, when a normal function is called, its code is executed and when a generator function is called it just returns a generator object. When this generator object is iterated over, then the code of the generator function gets executed. The code of a normal function is executed each time it is called and code of a generator is executed each time it is iterated over.

A normal function executes and returns a single value, while a generator function produces a sequence of values. These values are produced by iterating over the generator object. It is similar to what we have seen in iterators. The values that are produced and given out are created in the `yield` statement.

The normal function gives out its value using the `return` statement, while the generator function gives out its values using the `yield` statement.

Each time you call a normal function, the code inside it is executed from the beginning. When a `return` statement is encountered, the function execution stops, all local variables are destroyed and the value in the `return` statement is given out. A normal function does not remember anything about the previous calls, it always starts with the same initial state.

A generator function is different from a normal function in that it retains the state when it was last called. During the execution of a generator function, the function execution is stopped when a `yield` statement is encountered, and value in the `yield` statement is given out. When the function execution stops due to the `yield` statement, the local variables including the parameters are not destroyed, function remembers values of all the local variables and also the place where the function execution stopped so that in the next execution the function resumes from there. So, when next time the generator is invoked by iterating over it, the code does not execute from the beginning but it continues where the previous execution had stopped.

So, the code of a normal function always starts executing from the beginning of the function i.e., the first line, while a generator when executed starts from the place where the previous call had left. The difference between a `return` statement and a `yield` statement is that the `return` statement when executed throws away the local state of the function while the `yield` statement retains the local state of the function. Let us understand all this with the help of examples.

We have the following generator function which is not of any use but it will help us understand how generator functions work:

```
>>> def gen_fn():
...     n = 0
...     print('ABC', n)
...     n += 2
...     yield 10
...     print('GHI', n)
...     print('XYZ')
...     yield 20
...     print('JKL', n)
...     n *= 5
...     yield 30
...     print('MNO', n)
>>> g = gen_fn()
```

We called the generator function and got the generator object in variable `g`. Now we will iterate over this generator object manually using the `next` function.

```
>>> v = next(g)
ABC 0
```

First three statements of the function were executed and then the `yield` statement was encountered so the execution stopped and 10 was returned, which is assigned to variable `v`. We can see that the value of `v` is 10.

```
>>> v
10
```

Again, we call the `next` function on this generator object.

```
>>> v = next(g)
GHI 2
XYZ
```


In previous call, the execution had stopped at `yield 10`, so now the execution starts from the statement which is just after it. Next two statements are executed and again a `yield` statement is encountered so the execution stops and this time 20 is returned.

```
>>> v
20
```

Note that the value of `n` was remembered from the previous call. Again, we call the `next` function.

```
>>> v = next(g)
JKL 2
>>> v
30
>>> v = next(g)
MNO 10
StopIteration
```

This time the whole function code was finished and there was nothing to `yield` so now in this case the function execution stops and the `StopIteration` error is raised to indicate the exhaustion of the generator object. This error is raised to indicate that it has generated all the values and there are no more values left to provide.

If you try to reiterate over this generator object and you cannot, it is because it has been exhausted. Any attempt to iterate over this generator will raise the `StopIteration` error.

```
>>> v = next(g)
StopIteration
>>> for i in g:
...     print(i)
```

If we use it in a `for` loop, nothing happens because this exhausted generator object raises the `StopIteration` error, which is caught by the loop and immediately terminates.

It is not possible to restart or reiterate an exhausted generator object. If we want to iterate again, then we have to create another generator object by calling this function.

```
>>> g = gen_fn()
```

Now we have this fresh generator object. When we write this loop, it works.

```
>>> for i in g:
```

```
...     print(i)
```

```
...
```

```
ABC 0
```

```
10
```

```
GHI 2
```

```
XYZ
```

```
20
```

```
JKL 2
```

```
30
```

```
MNO 10
```

Generally, the `yield` statement in a generator function appears inside a loop, but here we have used it multiple times to make the working clear.

A generator function is like a generator factory, you can call it many times to get generator objects, each one will have their own state information, independent of each other.

Now, let us see the `cubes` generator that we have seen before:

```
def cubes(start, stop):  
    for n in range(start, stop+1):  
        yield n * n * n
```

In this generator function, we have used the `yield` statement inside a loop. We will get a generator object by calling this function with arguments 2 and 5.

```
y = cubes(2, 5)
```

Now, we will call the `next` function for this generator object.

```
>>> next(y)
```

```
8
```

When this `next` function is called, the execution starts from the `for` loop, and the value of `n` is 2. The `yield` statement is executed, so $8(2*2*2)$ is returned. The function execution has stopped, but the loop has not finished so when next time we will iterate over this generator object the loop will continue from where it had left. So now let us call `next` again.

```
>>> next(y)
```

```
27
```

The loop continues, `n` becomes 3 and then the `yield` statement is executed. 27 is returned and execution stops, but the loop is still not fully finished. Again, we call `next`.

```
>>> next(y)
```

```
64
```

```
>>> next(y)
```

```
125
```

Now the loop has finished, so it will terminate. There is nothing to execute and return, so the next time when we call `next`, the `StopIteration` error is raised.

```
>>> next(y)
```

```
StopIteration
```

Now this generator object is exhausted. This is how a generator function works and produces values.

Since the `yield` statement can be inside a loop, you can write generators that give long sequences or even infinite sequences. Let us change the generator function so that it gives the cubes infinitely.

```
def cubes(start):
```

```
    n = start
```

```
    while True:
```

```

        yield n * n * n
        n = n + 1
y = cubes(2)

```

Now, we do not have the parameter `stop` in our generator function. The variable `n` is initialized to `start` and we have written the `yield` statement inside an infinite while loop. So now we have an infinite generator object, which will give cubes infinitely.

In [Section 17.6](#), we had created this iterator that produced Fibonacci numbers.

```

class Fibonacci:
    def __init__(self, max):
        self.max = max
        self.a = 0
        self.b = 1
    def __iter__(self):
        return self
    def __next__(self):
        f = self.a
        if f > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return f
x = Fibonacci(100)
for i in x:
    print(i, end = ' ')
print(50 in x, 55 in x)

```

Now, let us write a generator function to do the same job. The generator function will produce an iterator automatically for us.

```

def fibo_gen(max):
    a = 0

```

```

    b = 1
    while a < max:
        yield a
        a, b = b, a + b
fib = fibo_gen(100)
for i in fib:
    print(i, end=' ')

```

Output-

```
0 1 1 2 3 5 8 13 21 34 55 89
```

This generator function generates Fibonacci numbers. If you want an infinite generation of numbers, then in place of the condition `a < max`, you can write `True`.

In [Section 17.6](#), we saw that we need two classes to support multiple scans.

```

class Fibonacci:
    def __init__(self, max):
        self.max = max
    def __iter__(self):
        return FiboIterator(self)
class FiboIterator:
    def __init__(self, source):
        self.source = source
        self.a = 0
        self.b = 1
    def __next__(self):
        f = self.a
        if f > self.source.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return f

```

```
x = Fibonacci(100)
for i in x:
    print(i, end = ' ')
print(55 in x, 50 in x)
```

The `__iter__` method of the `Fibonacci` class should return an iterator, so we have created an instance of the `FiboIterator` class and returned it. Generators provide an easy way to get an iterator, so we can use a generator function here. Instead of writing the whole `FiboIterator` class for instantiating an iterator object, we can simply make the `__iter__` method a generator. So, then it will return a generator object which is an iterator.

```
class Fibonacci:
    def __init__(self, max):
        self.max = max
    def __iter__(self):
        a = 0
        b = 1
        while a < self.max:
            yield a
            a, b = b, a+b
x = Fibonacci(100)
for i in x:
    print(i, end = ' ')
for i in x:
    print(i, end = ' ')
```

Now this construct supports multiple active iterators. So, you can define your iterable class by implementing its `__iter__` method as a generator.

17.13 Generator expressions

There are two ways of writing generators; the first way is the generator function, which we have already seen; now, let us see what generator expressions are.

Generator expression is an expression that returns an iterator also known as generator object. This generator object returns values one by one when used in an iteration context such as for loop.

Generator expressions are syntactically almost similar to list comprehensions, but the difference is that generator expressions return a generator object instead of a list. Generator objects generate one value at a time, while the comprehensions save all the values in memory. This is why generator expressions consume less memory as compared to comprehensions, and there is no waiting time as all the values are not computed at once. The saving in memory and time is crucial when the number of data values is very large. Let us see an example:

```
>>> (n * n * n for n in range(2, 6))
<generator object <genexpr> at 0x000001F5FD451E70>
```

This is an example of a generator expression. We know that list comprehensions are enclosed in square brackets, while set and dictionary comprehensions are enclosed in curly braces and we do not have anything like tuple comprehensions, so when we have a comprehension like expression enclosed in parentheses, it is a generator expression. This generator expression gives us a generator object which is lazily evaluated, and it can be iterated over.

```
>>> g = (n * n * n for n in range(2, 6))
>>> g
<generator object <genexpr> at 0x000001F5FF58B1B0>
```

g is a generator object, let us call next function for it.

```
>>> next(g)
8
>>> next(g)
27
>>> next(g)
```

64

```
>>> next(g)
```

125

```
>>> next(g)
```

StopIteration

Now it is exhausted, so we cannot use it again.

We had seen that the same work can be done by a generator function also. So, if you have a simple generator function, you can think of writing a generator expression instead of defining a full generator function. But with a generator function you have an advantage that you call it again to get a new generator object, with generator expression you cannot do that.

As in comprehensions, you can use `if` clause and nested `for` inside the generator expression. For example, we could write:

```
>>> g = (n*n*n for n in range(2,6) if n%2 ==0)
```

This will give cubes of only even numbers.

A generator expression can be used in line also, for example:

```
>>> for i in (n * n * n for n in range(2, 6) if n
% 2 == 0):
...     print(i)
```

Generator expressions are written inside parentheses. If you are writing the generator expression as a single argument to a function call, then the parentheses of the function call are sufficient, there is no need to write 2 pairs of parentheses, one for the call and other for the generator expression. But if there are more than one argument, then you need to enclose the generator expression in parentheses otherwise you will get a syntax error.

For example, in the following function call, we have sent a generator expression as argument.

```
>>> func(n * n for n in range(2,4))
```

Since this is the only argument, the parentheses are not required. If we have more arguments, then we will have to put the parentheses.

```
func((n*n for n in range(2,4)), 'x')
```


Exercise

1. Objects of `float` type are iterables.
(A) True (B) False
2. The built in function `iter()` returns an _____ from an _____.
(A) iterator, iterable (B) iterable, iterator
3. The value of expression `x is iter(x)` is _____ if `x` is an iterator.
(A) True (B) False
4. An iterable cannot represent an infinite source of data.
(A) True (B) False
5. Dictionaries are iterables that produce an iterator which gives one key at a time when used in an iteration context.
(A) True (B) False
6. The _____ method should raise `StopIteration` exception to indicate completion.
(A) `__next__` (B) `__iter__`
7. Any function that contains a _____ statement is a generator function.
(A) `return`
(B) `yield`
(C) `goto`
8. Which one creates a generator object.
(A) `(x*x for x in range(1,10))`
(B) `[x*x for x in range(1,10)]`
(C) `{x*x for x in range(1,10)}`
9. `x = {10, 20, 30}`
`y = iter(x)`

Which of these is correct?

(A) y is an iterable

(B) y is an iterator

(C) Both are correct

10. `x = range(1, 4)`

`y = open('data.txt', 'r')`

x is an _____ and y is an _____.

(A) iterable , iterator (B) iterator , iterable

11. According to iterator protocol, the _____ method should return an iterator object that implements a _____ method which is responsible for carrying out the actual iteration.

(A) `__next__` , `__iter__`

(B) `__iter__` , `__next__`

12. (1) An iterable object responds to the `iter()` function by returning an iterator.

(2) An iterator object responds to the `iter()` function by returning an iterator.

(A) only (1) is True

(B) only (2) is True

(C) Both (1) and (2) are True

(D) Both (1) and (2) are False

What will be the output of code given in questions 13 to 21?

13. `L = [1, 2, 3]`

`it = iter(L)`

`for i in it:`

`print(i, end=' ')`

`print(sum(it))`

```
14. t = (3, 5, 7)
    it = iter(t)
    a, _, b = it
    print(a, b)
15. L = [1, 2, 3]
    x = iter(L)
    print(next(x), end=' ')
    L[2] = 300
    print(next(x), next(x))
16. L1 = [1, 2]
    L2 = ['a', 'b']
    x = zip(L1, L2)
    L = list(x)
    t = tuple(x)
    d = dict(x)
    print(L, t, d)
17. def inc_gen():
    i = 1
    while True:
        yield i
        i = i + 1
    inc = inc_gen()
    print(next(inc), end=' ')
    print(next(inc), end=' ')
    print(next(inc), end=' ')
```

```
18. def gfn():  
    yield 1  
    yield 2  
    return 10  
    yield 3  
    yield 4  
  
    for i in gfn():  
        print(i, end = ' ')
```

```
19. def gfn():  
    x = 1  
    while True:  
        if x <= 5:  
            yield x  
        else:  
            return  
        x += 1  
  
    for n in gfn():  
        print(n, end=' ')
```

```
20. class Odd:  
    def __init__(self, max):  
        self.num = 1  
        self.max = max  
  
    def __iter__(self):  
        self.num = 1  
        return self
```

```

def __next__(self):
    if self.num > self.max:
        raise StopIteration
    self.num += 2
    return self.num - 2

x = Odd(20)
for i in x:
    print(i, end = ' ')
print()
print(sum(x))
for i in x:
    print(i, end = ' ')

```

21. class PowerTwo:

```

def __init__(self, max = 1):
    self.max = max

def __iter__(self):
    self.n = 1
    return self

def __next__(self):
    if self.n <= self.max:
        result = 2 ** self.n
        self.n += 1
        return result
    else:
        raise StopIteration

```

```

x = PowerTwo(5)
for i in x:
    print(i, end = " ")
print()
print(sum(x))

```

22. The output of both these code extracts is the same, which one takes less memory?

(A)

```
def func(a):
    return [i * i for i in a]
L = list(range(100))
for i in func(L):
    print(i)
```

(B)

```
def gfunc(a):
    for i in a:
        yield i * i
L = list(range(100))
for i in gfunc(L):
    print(i)
```

23. Write a class for implementing an infinite iterator that gives out Fibonacci numbers.

24. Write two classes, `Squares` and `SquaresIterator`, to implement an iterable that gives squares of numbers and supports multiple scans.

25. Implement an iterator that gives squares of numbers using a single class.

26. Write two classes, `Factorial` and `FactorialIterator`, to implement an iterable that gives factorials of numbers and supports multiple scans.

27. The range function does not take a float value as a step, so calls like `range(1, 10, 0.5)` do not work. Write your own version of range that can accept a float value as the step.
28. Write a generator function that behaves like the built-in `enumerate` function.
29. Write a generator function that takes a number `n` and then yields values from `n` to 1 and then again up to `n`. If `n` is 5, the values that are generated are: 5 4 3 2 1 2 3 4 5
30. Write an infinite generator that yields values 1, -1, 2, -2, 3, -3, ... Use it in a loop with a break statement.
31. The following code gives a `TypeError`:

```
L = [1, 2, 3, 4, 5]
def combine1(a, b, c):
    return a + b + c
for i in combine1(L, range(10,15), 'ABCDEF'):
    print(i, end = ' ')
```

Output-

```
TypeError: can only concatenate list (not
"range") to list
```

Write a generator function named `combine` that can yield values from the list, range function, and string.

32. Write a generator that generates squares of numbers.
33. Write a generator function that accepts a sequence and generates its numbers in reverse order.
34. Write an infinite generator function that generates strings 'Jan', 'Feb', 'Mar', ... 'Dec'.
35. Write a generator expression that generates all the non-empty lines in a file.
36. The following code prints the dot product of 2 lists. (Dot product is the sum of the products of the corresponding numbers in two

sequences.)

```
L1 = [5, 10, 15, 20]
```

```
L2 = [1, 2, 3, 4]
```

```
dot_product = sum([a * b for a, b in zip(L1,  
L2)])
```

```
print(dot_product)
```

How can you make this code memory efficient?

37. Write a generator function to generate the first n multiples of a number. For example, the first 5 multiples of 6 are 6, 12, 18, 24, 30.

38. Write a generator function that accepts a number and generates its factors. For example:

Factors of 500 are 1, 2, 4, 5, 10, 20, 25, 50, 100, 125, 250, 500

39. Write a program to generate factor pairs of a number. For example, if the number is 500, the generator should generate these tuples- (1, 500) (2, 250) (4, 125) (5, 100) (10, 50) (20, 25)

40. What will be the output of the following code. If the program does not give the desired output, what can you do to correct it?

```
def generate_squares(start, stop):
```

```
    while start <= stop:
```

```
        yield start * start
```

```
        start += 1
```

```
g = generate_squares(2, 9)
```

```
L = [4, 9, 16, 25, 36, 49, 64, 81]
```

```
def func(data):
```

```
    print(sum(data))
```

```
    for i in data:
```

```
        if i % 2 == 0:
```

```
            print(i, end=' ')
```



```
print()
```

```
func(L)
```

```
func(g)
```

41. Write an iterator class named `Triplets` that can be used to create iterators that produce tuples of 3 adjacent elements successively from a sequence. Here is an example of the usage of that class:

```
L = [21, 33, 65, 18, 81, 24, 46, 68, 79, 89, 90, 91]
```

```
x = Triplets(L)
```

```
for i in x:
```

```
    print(i)
```

```
names = ['Raj', 'Dev', 'Sam', 'Pam', 'Tom', 'Ram', 'Kim', 'Rob', 'Sim', 'Tim']
```

```
for i in Triplets(names):
```

```
    print(i)
```

Output-

```
(21, 33, 65)
```

```
(18, 81, 24)
```

```
(46, 68, 79)
```

```
(89, 90, 91)
```

```
('Raj', 'Dev', 'Sam')
```

```
('Pam', 'Tom', 'Ram')
```

```
('Kim', 'Rob', 'Sim')
```

42. Write an iterator class named `AdjacentElements` which is the generalized version of the `Triplets` class you wrote in the previous question. It can be used to create iterators that produce tuples of adjacent elements of any length from a sequence. Here is an example of usage of that class:

```

L = [21, 33, 65, 18, 81, 24, 46, 68, 79, 89,
     90, 91, 12, 93, 24, 95]
for i in AdjacentElements(L, 4):
    print(i)
for i in AdjacentElements(L, 3):
    print(i)

```

Output-

```

(21, 33, 65, 18)
(81, 24, 46, 68)
(79, 89, 90, 91)
(12, 93, 24, 95)
(21, 33, 65)
(18, 81, 24)
(46, 68, 79)
(89, 90, 91)
(12, 93, 24)

```

43. Write a generator function that generates prime numbers infinitely.
44. What will be the output of this code?

```

def max_limit(data, maximum):
    for item in data:
        if item < maximum:
            yield item
def min_limit(data, minimum):
    for item in data:
        if item > minimum:
            yield item

```

```
L = [2, 8, 4, 1, 5, 6, 7, 9]
d = {'a': 5, 'b': 4, 'c': 5, 'd': 2, 'e': 9}
print(sum(max_limit(L, 5)), end=' ')
print(sum(min_limit(d.values(), 4)))
```

45. What is the difference between these two lines of code?

```
total = sum([n*n*n for n in range(1,
1000000)])

total = sum(n*n*n for n in range(1, 1000000))
```

46. The following two loops perform the same work; which one has a better performance in terms of memory usage?

```
for line in open('data.txt').readlines():
    print(line, end=' ')

for line in open('data.txt'):
    print(line, end= ' ')
```

18.1 Prerequisites for understanding decorators

Before learning about decorators, let us recapitulate some of the points that we have learnt in functions because they are crucial for understanding decorators.

We know that functions are first-class objects in Python, which means that a function can be assigned to different variables, sent as an argument to a function, and returned from a function. Let us see some examples of this. Suppose we have a function named `fn`:

```
def fn():  
    pass  
f1 = fn  
f2 = fn  
fn()  
f1()  
f2()
```

We know that `def` is an executable statement that creates a function object, so when the above `def` statement is executed, a function object is created and is assigned to name `fn`. A function name is just a reference to the function object, and we can make multiple names refer to the same function object. We have assigned `fn` to `f1` and `f2`, so the names `f1` and `f2` also

refer to the same function object to which `fn` is referring. Therefore, now we can call the function using any of the names `fn`, `f1` or `f2`.

Next, we have defined a function `func1`, which accepts a function as argument, and calls that function inside its body using the parameter name `f`.

```
def func1(f):  
    .....  
    f()  
    .....  
func1(fn)
```

While calling `func1`, we have sent function `fn` as the argument. In the definition of function `func1`, the parameter name is `f`, so for the function call `func1(fn)`, `f` refers to the same function object to which `fn` is referring. Thus inside the function when the statement `f()` is executed, the function `fn` gets called.

Now, let us see an example of a function being returned from a function.

```
def func2():  
    .....  
    .....  
    return g  
f3 = func2()  
f3()
```

Here, we have defined the function `func2` and it returns `g`, which is a function or more specifically, we can say that it is a reference to a function object.

We have called the function `func2` and assigned its return value to name `f3`. So `f3` also starts referring to the same function object to which `g` is referring. We can use the name `f3` as a regular function which means that we can call it with the function calling syntax.

These types of functions that accept a function as an argument or return a function are called higher-order functions. So, the functions `func1` and

`func2` that we saw are both higher-order functions.

It is possible to define a function inside the definition of another function, which means that inside the body of a function, we can write a `def` statement. In the following example, we have defined a function `g` inside the definition of the function `func`:

```
def func() :  
    .....  
    .....  
    def g() :  
        pass  
    .....  
    .....
```

Now, whenever the function `func` will be called, the `def` statement inside it will be executed, and it will create a function object that will be assigned to name `g`. The name `g` is local to the function `func`, so it can be used inside this function only. You cannot call the function `g` outside the function `func`. If you want to call it, you have to call it inside this function.

```
def func() :  
    .....  
    .....  
    def g() :  
        pass  
    .....  
    .....  
    g()
```

As we have seen, it is possible to return a function from a function. Let us return the function `g` from the function `func`:

```
def func() :  
    .....  
    .....
```

```

def g():
    pass
.....
.....
g()
return g
f1 = func()
f1()
print(f1.__name__)

```

The statement `return g` means that the function object to which `g` refers is being returned from the function. We have called function `func` and assigned the return value to name `f1`. So, the name `f1` refers to the function object created by the `def` statement that defined `g`. We can say that `f1` becomes an alias for the inner function `g`, and so when we call `f1`, we are actually calling the function `g`. If you access the `name` attribute of `f1`, it will print `g`. So, you cannot directly call the function `g` outside of `func`, because of its local scope. But if you return it from `func`, you can indirectly access it.

Any variable that exists in the scope of the outer function can be accessed in the inner function.

```

def func(f):
    x = 6
    .....
    def g():
        print(x)
        f()
    .....
    .....
    g()
    return g

```

Here we have a variable `x` defined inside `func`, and we can easily access it inside `g`. The function `func` has a parameter `f` which is supposed to be a function, this name `f` is also available to the inner function. You can call `f` inside the inner function. So, the inner function has access to variables of the outer function. Inside this inner function, these variables are called free variables.

This inner function, along with the free variables, is a closure. Therefore, a closure is an inner function that has access to and remembers variables in the scope of the outer function, even when the outer function has finished executing.

Now, after this review of functions, we are ready to learn about decorators.

18.2 Introduction to decorators

A decorator is a callable that takes a callable as input and returns a callable. This is the general definition of a decorator. The callable in this definition can be a function or a class. In our initial discussion, we will talk about decorator functions that are used to decorate functions; we will talk about classes later on. So, for the time being, we can think of a decorator as a function that takes a function as input and returns a function.

Decorators are used to add some functionality to a function. They allow you to execute some extra code before or after the execution of a function. This extra work is done without making any changes to the source code of the function. So, by using a decorator we can extend the behavior of a function without actually modifying its code.

A decorator is a function that takes another function as an argument, decorates it with the extra functionality and gives you a decorated function as the result.

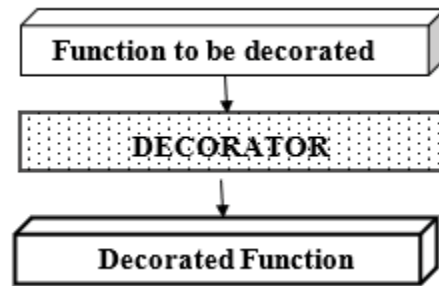


Figure 18.1: Decorators

Decorators are functions, so they are reusable pieces of code; we can apply a decorator to different functions to add the same functionality to all of them without changing their code.

There are some decorators that are built-in (e.g., `classmethod`, `staticmethod`) and many third-party libraries also provide decorators for some common functionalities. These all are readymade decorators that we can use to decorate our functions. We can also define our own decorators; these are called user defined decorators.

18.3 Writing your first decorator

```
def func1():  
    print('Hello world')  
    print('Welcome to Python')
```

We have this simple function named `func1`. Our requirement is that whenever we execute this function, we need the statement `print('Hi ... Starting execution')` to be executed before the execution of the function and the statement `print('Bye ... finished executing\n')` to be executed after the execution of the function. But we do not want any changes inside the code of this function.

To do this, we will define a decorator function and use that decorator to decorate this function with the extra code that is to be executed before and after the execution of the function. Once we have a nice working decorator, we can use it to decorate other functions with the extra code. So, now let us see how to create a decorator to add this extra code:

```
def my_decorator(fn):
    def wrapper():
        print('Hi ... Starting execution')
        fn()
        print('Bye ... finished executing\n')
    return wrapper
decorated_func1 = my_decorator(func1)
```

Our decorator is a function, so we define it by writing `def`, and we have given it the name `my_decorator`. It has one parameter named `fn`. When we call this decorator function, we will send the function that we want to decorate as the argument.

The decorator function takes a function as argument and decorates it, so now let us see how it does the decoration. We have defined an inner function and named it `wrapper` because it will wrap our original function with the extra code. Inside this `wrapper` function we have called the parameter function `fn`. This function call will actually call the undecorated original function that is sent as an argument. As we have seen before, the inner function gets access to the variables of the outer function, so the `wrapper` function can access `fn`, which is the parameter of the outer function.

Before execution of the argument function, we want `print('Hi ... Starting execution')` to be executed, so we write it before `fn()`, and after execution of the function, we want `print('Bye ... finished executing\n')` to be executed, so we write it after `fn()`. The `wrapper` function does the work that our original function did, and it also does the extra work. We can think of it as the decorated version of our original function. After defining this inner function, we just return it from the outer `my_decorator` function.

Then we called the `my_decorator` function, and we sent `func1` as argument because we want it to be decorated with the extra code.

```
decorated_func1 = my_decorator(func1)
```

When this call executes, the code inside the function body of `my_decorator` will be executed, so the inner `def` statement will

execute, which creates a new function object. The reference to this function object is returned which is assigned to the name `decorated_func1`.

Now, we can use the name `decorated_func1` as a function which means that we can call it with function syntax.

```
decorated_func1()
```

The name `decorated_func1` is a reference to the wrapper function object that was created inside `my_decorator` so when we call `decorated_func1`, the code that is there inside wrapper will be executed. Therefore, first `print('Hi ... Starting execution')` will execute, then the undecorated function that was sent as argument will execute and then `print('Bye ... finished executing\n')` will execute. This is the output that we will get:

```
Hi ... Starting execution
```

```
Hello world
```

```
Welcome to Python
```

```
Bye ... finished executing
```

So `decorated_func1` is the decorated version of the function `func1`. When it is executed, `func1` executes, and the extra code executes. If we call `func1`, we still get the usual undecorated output. But our requirement was that when we call `func1`, we get the decorated output. For that, we can assign the return value of `my_decorator` to `func1` instead of `decorated_func1`.

```
func1 = my_decorator(func1)
```

Now, we have reassigned the name `func1` with the return value of the decorator, so after this statement, `func1` no longer refers to the function object created by `def func1()`. Instead, it refers to the function object created by the `def wrapper()`. Now, when we call `func1`, the wrapper function will be executed, which calls the original function as well as the additional code, and so we will get the decorated output.

Let us once again see how our decorator function is working. It takes the undecorated function as argument, then it defines an inner function, inside which it executes the undecorated function and the extra decoration code, and then it returns this inner function. When we call our decorator function,

the inner function is created and reference to it is returned, and we assign the return value to `func1`. So now `func1` refers to the wrapper function. Now when `func1` is called, the code inside `wrapper` is executed. And this code executes the extra decoration statements plus the original `func1` because `fn` is a reference to the original `func1`. The net effect is that our `func1` got decorated with the extra code. By writing the statement `func1 = my_decorator(func1)`, we modified `func1` so that it does its own work and the extra work specified in the decorator.

18.4 Applying your decorator to multiple functions

By defining a decorator, we actually create a reusable piece of code that can be applied to any function to extend its functionality. So, we can use our `my_decorator` function to decorate other functions also. Suppose we have these 2 functions, `func2` and `func3`.

```
def func2():
    for i in range(10):
        print(i, end=' ')
    print()
def func3():
    print('Learning decorators')
```

We can use our decorator function to decorate these functions also.

```
func2 = my_decorator(func2)
func3 = my_decorator(func3)
```

Now `func2` and `func3` also have been decorated, so when we call them, we will get the decorated output.

```
func2()
func3()
```

Output-

```
Hi ... Starting execution
```

```
0 1 2 3 4 5 6 7 8 9
```

```
Bye ... finished executing
```

```
Hi ... Starting execution
```

```
Learning decorators
```

```
Bye ... finished executing
```

Generally, decorators are placed in separate modules so that they can be reused in many places.

18.5 Automatic decoration syntax

The reassignment statement that we are writing to decorate a function is the manual way of applying decorator to a function. This pattern is very common so Python provides an automatic way of applying the decorator. You just need to add the `@` sign and decorator name before the definition of function that you want to decorate. Let us decorate the function `func3` using the automatic syntax.

```
@my_decorator
def func3():
    print('Learning decorators')
```

The function `func3` is decorated using the automated decoration syntax and so now we do not need to write the statement `func3 = my_decorator(func3)`. The `@` syntax automates this reassignment of the function name.

We have already used this syntax before when we had defined class methods, static methods and properties. There we had applied decorators that are predefined in Python, here we are applying a user defined decorator.

The two ways of decorating are equivalent, use of `@` syntax is just syntactic sugar that is used in place of the reassignment statement.

When we use the assignment way of decoration, both the undecorated and decorated versions of the function exist in the program. Before the

assignment, if you use the function, you get the undecorated version while after the assignment statement you get the decorated version.

If you decorate a function using `@` syntax, then only the decorated version exists in the program, because the function is decorated as soon as it is defined.

```
@my_decorator
def func3():
    print('Learning decorators')
```

When you write this code, it means that define the function `func3` and then immediately apply the decorator on the function. It is equivalent to writing this:

```
def func3():
    print('Learning decorators')
func3 = my_decorator(func3)
```

Most of the times the undecorated version of the function is not required, so `@` syntax is mostly used. This syntax is easier to use and increases readability also. The decoration is done near the function definition so it is easier for anyone to understand that the function has been decorated.

So, we have seen how to define a decorator and how to apply it to a function. We can see that code of these functions has not been changed. When the decorator is assigned, you get the modified or extended behaviour. If you want the original behaviour, you can remove the decorator. You get the modified behaviour only when it is decorated.

Next, we will see a few examples of user defined decorators.

18.6 Decorator Example: Timer

We have three functions that perform different tasks and we want to calculate the time that each one of them takes to execute.

```
def func1():
    x = 999 ** 99999
def func2():
```

```
L = [x for x in range(9999999)]  
def func3():  
    x = (66 * 9999) ** 99988
```

First let us write the code to time the function func1.

```
from time import time  
start_time = time()  
func1()  
end_time = time()  
print(f'func1 took {end_time - start_time}  
seconds')
```

We have invoked the `time` function before the execution of `func1` and after the execution of `func1`, and we have stored the start time and end time in two separate variables. By subtracting `start_time` from `end_time` we get the time that was taken by this function to execute. At the end, we print the message that shows the time taken by `func1`. This way we get to see the time that this function takes for execution.

If we want to time the other 2 functions also, then we have to write the same code.

```
start = time()  
func2()  
end = time()  
print(f'func2 took {end - start} seconds')  
start = time()  
func3()  
end = time()  
print(f'func3 took {end - start} seconds')
```

We are repeating the same code, which means that it is time to make a function for the repetitive task. So let us make a decorator function for this timing task.

```
def timer(fn):  
    def wrapper():
```

```

        from time import time
        start = time()
        fn()
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return wrapper

```

The function `timer` accepts a function as argument. Inside the function we have defined another function `wrapper`. Inside `wrapper`, we have called the argument function `fn` and also written the timing code. In the `print` call, we have used the `__name__` attribute of `fn` instead of the specific function name. At the end, the `wrapper` function is returned from this decorator.

Now we can decorate our functions with this decorator named `timer` and there will be no need to write separate timing code for each one.

```

@timer
def func1():
    x = 999**99999

@timer
def func2():
    L = [x for x in range(9999999)]

@timer
def func3():
    x = (66*9999)**99988

func1()
func2()
func3()

```

Output-

```

func1 took 0.031242847442626953 seconds
func2 took 0.5936102867126465 seconds

```



```
func3 took 0.09372854232788086 seconds
```

We will get the same output as before, and now we have a decorator which can be used to time other functions also.

18.7 Decorator Example: Logger

Our next decorator will log function calls to a file.

```
def logger(fn):
    def wrapper():
        from time import ctime
        with open('log.txt', 'a') as fout:
            fout.write(f'{fn.__name__} called at
{ctime()}\n')
        fn()
    return wrapper
```

As usual, this decorator has parameter `fn` which will accept a function argument. Inside the `wrapper` function, we have called `fn` and before this call we have logged the call to file `log.txt`. The function name and current time are written to the file `log.txt`. Let us decorate our functions with this new decorator.

```
@logger
def func1():
    x = 999 ** 99999

@logger
def func2():
    L = [x for x in range(9999999)]

func1()
func2()
```

When we open the file `log.txt`, we will see that all the calls to these decorated functions will be logged into that file.

18.8 Decorator Example: Counting function calls

The following decorator helps us keep track of the number of calls that are made to a function.

```
def calls_counter(fn):
    def wrapper():
        wrapper.number_of_calls += 1
        fn()
    wrapper.number_of_calls = 0
    return wrapper

@calls_counter
def func1():
    x = 999 ** 99999

@calls_counter
def func2():
    L = [x for x in range(9999999)]

func1()
func2()
func1()
func2()
func1()
print(func1.number_of_calls)
print(func2.number_of_calls)
```

Output-

3
2

When the decorator function is executed, the inner `def` executes and creates a function object. Remember that the execution of `def` statement does not mean execution of the statements written inside that function, they

are executed only when the function is called. After the execution of `def wrapper()` statement, we add the attribute named `number_of_calls` to the function object that was created by `def`. This attribute is initialized to zero, and it will be used to keep count of how many times the function is called.

Inside the `wrapper` function, we increase the value of this attribute by 1. So, whenever the `wrapper` function will be executed, this attribute will be increased by 1.

We applied this decorator to functions `func1` and `func2`, so these functions got a new attribute attached to them which tells us the number of times they are called.

18.9 Applications of decorators

We have seen that we can use decorators to count function calls, to calculate the time that a function takes to execute, for logging function calls to a file or to some other location.

You can add debugging information to the function, for example you can write a decorator to show what arguments were passed to the function and what was the return value. Decorators can be used to check for prerequisites before a function executes. For example, you can write decorators that can check the argument types or values. By using decorators, you can ensure that the arguments are of specific type or are in a certain limit. You can even sanitize the arguments before they are passed to the function.

You can create decorators to check for authentication and access privileges. Authentication may include validation of usernames and passwords and access control specifies the permissions of a given user to access a given function. Django which is a popular web framework, uses the `@login_required` and `@permissions_required` decorators for checking the login status and permissions of the user before viewing a specific web page.

Decorators can be useful in cleaning up operations after the function's execution. They can be used to check and sanitize the return value or can be

used in exception handling. They can be used in caching and to register functions in a task runner or a signalling system.

If you have some generic code that is to be executed before or after a function's execution, you can create a decorator for that. Decorators allow us to modify behaviour of an existing function without actually modifying its code. You can write decorators for common functionalities and hence can make those functionalities reusable. So, you can avoid copying and pasting generic code in different functions.

18.10 Decorating functions that take arguments

We have written four decorator functions - `my_decorator`, `timer`, `logger` and `calls_counter`. We can apply these decorators only to those functions that don't take any arguments and don't return any value.

Here is our `timer` decorator and we have two functions `func1` and `func2`. The function `func1` does not take any argument while `func2` takes a single argument.

```
def timer(fn):
    def wrapper():
        from time import time
        start = time()
        fn()
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return wrapper

@timer
def func1():
    L = [x for x in range(999999)]

@timer
```

```
def func2(n):
    total = 0
    for i in range(n):
        total += i
    print(total)
func1()
func2(999999)
```

Output-

```
func1 took 0.05585026741027832 seconds
Traceback (most recent call last):
  File "C:\test.py", line 22, in <module>
    func2(999999)
TypeError: timer.<locals>.wrapper() takes 0
positional arguments but 1 was given
```

The call to `func1` worked but the call to `func2` failed. When we called `func2`, the `wrapper` gets called, and it does not take any argument and that is why we get the error. So, our `timer` decorator can be applied to only those functions that don't take any arguments. This is because the `wrapper` inside the decorator does not take any arguments. Let us change it so that it takes one argument.

```
def timer(func):
    def wrapper(n):
        from time import time
        start = time()
        func(n)
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return wrapper
```

Now we made the `wrapper` function take one argument, the `wrapper` forwarded this argument to the original undecorated function because it is

the function that actually needs this argument and will use it. Now when we will run the program, it will work for `func2` but not for `func1`. After this change, this decorator can be applied to only those functions that take one argument. We cannot apply it to `func1` function as it does not take any argument. The `wrapper` function which is returned from the decorator should take the same number of arguments as the original undecorated function expects.

We would like our `timer` decorator to be generalized so that it can be applied to all functions irrespective of the number of arguments that they take. To make our decorator generic, we can use `args` and `kwargs` in the function header to collect variable number of positional and keyword arguments. So, now we will make our `wrapper` function take variable number of positional and keyword arguments, and then we will forward them to the undecorated function.

```
def timer(fn):
    def wrapper(*args, **kwargs):
        from time import time
        start = time()
        fn(*args, **kwargs)
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return wrapper
```

Now this decorator will work for both `func1` and `func2`, and it will work for any function irrespective of the number of arguments that it takes. Similarly, we can change our other decorators also so that they can be applied to any function.

18.11 Returning values from decorated functions

Let us see what happens when we use our `timer` decorator with a function that returns a value.

```

def timer(fn):
    def wrapper(*args, **kwargs):
        from time import time
        start = time()
        fn(*args, **kwargs)
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return wrapper

@timer
def func(n):
    print('func executing')
    return n * 2

s = func(9)
print(s)

```

Output-

```

func executing
func took 0.003990888595581055 seconds
None

```

When we called `func`, we stored the return value in variable `s`. On printing `s`, we get `None` which means that `None` was returned from the function. In the process of decorating the function, we lost its return value. This is because `wrapper` does not explicitly return any value so `None` is returned. Inside the `wrapper` function, we are just calling the undecorated function, and we are not storing its return value anywhere and so it is lost. This decorator works fine for functions that don't return any value, but for functions returning a value we need to make changes in it.

```

def timer(fn):
    def wrapper(*args, **kwargs):
        from time import time
        start = time()

```

```

        result = fn(*args, **kwargs)
        end = time()
        print(f'{fn.__name__} took {end - start}
seconds')
    return result
return wrapper

```

When we call the original function inside `wrapper`, we store the return value in the variable `result`. The value of `result` is returned from the `wrapper` function. Now when we execute our program, we will get the following output.

```

func3 executing
func3 took 0.007977008819580078 seconds
18

```

This decorator will work for those functions also that don't return any value, because `None` is returned from functions that don't return a value. Thus, for functions that don't return any value, `None` will be assigned to `result` and so `None` will be returned from `wrapper`. So, now our timer decorator will work properly with any function.

18.12 Decorator Example: Checking return values

We can write decorators that can be applied to functions to check their return values. Here is a decorator that raises a `ValueError` if the return value of a function is not within the limit 0 to 1000.

```

def limit_return_value(fn):
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        if result < 0 or result > 1000:
            raise ValueError('Value returned is
not within limits')

```



```

        return result
    return wrapper
@limit_return_value
def func1(x, n):
    return x ** n
@limit_return_value
def func2(n):
    total = 0
    for i in range (n):
        total += i
    return total

x = func1(2,5)
print(x)
y = func2(600)
print(y)

```

Inside the `wrapper` function, the first line is a call to the original function. This is because we don't have any extra code to be executed before the function's execution. After the function's execution, we want to check the return value, so we have put the check after the call.

We have applied this decorator to the functions `func1` and `func2`, both of them return a value. The call to `func1` works because it returns a value which is within our specified range. The call to `func2` raises a `ValueError` since a value that was more than 1000 was returned.

So, this way we can put a check on the return value of a function by applying a decorator.

If we want, we can create a decorator to sanitize the return value, for example we can convert the return value into a different format and then return the modified return value from the wrapper.

18.13 Decorator Example: Checking argument values

We can write decorators that can check argument values, for example the following decorator will ensure that only integers are sent to the function that it decorates.

```
def accepts_ints(fn):
    def wrapper(*args, **kwargs):
        arguments = args
        arguments += tuple(kwargs.values())
        for argument in arguments:
            if not isinstance(argument, int):
                raise TypeError('This function
accepts only integer arguments')
        result = fn(*args, **kwargs)
        return result
    return wrapper

@accepts_ints
def func1(x, n):
    return x ** n

@accepts_ints
def func2(n):
    total = 0
    for i in range (n):
        total += i
    return total

x = func1(2, n=5.8)
y = func2(50)
print(x, y)
```

We want to check the argument types before the function call, so we have written our extra decoration code before the call to the undecorated function. We have collected all the positional arguments and keyword arguments in a single tuple and then iterated over that tuple to check the type of each argument. While iterating, if we find that any argument is not an `int`, then a `TypeError` is raised.

18.14 Applying Multiple Decorators

A function can be decorated by multiple decorators. For example, the following function has been decorated by 3 decorators.

```
@decorator1
@decorator2
@decorator3
def func():
    print('Hello')
```

The decorator that is closest to the function definition is applied first and then other decorators are applied. So, first `decorator3` is applied to this function, then `decorator2` and then `decorator1` will be applied. The decorators are applied from bottom to top, these are also called stacked decorators. If you do the decoration manually, then this code is equivalent to the following code.

```
def func():
    print('Hello')

func = decorator1(decorator2(decorator3(func)))
```

First the function `func` is created and then it is passed to `decorator3`; `decorator3` returns a modified function which is then passed to `decorator2`; `decorator2` returns a modified function which is passed to `decorator1` and `decorator1` returns a modified function which is assigned to the name `func`. There can be cases where order of application of decorators does matter, so you need to know about the order in which they are applied. For example, suppose you need to apply two decorators named `authentication` and `authorisation`. The

authentication decorator checks the user details and after the user has been verified, authorisation decorator checks the access privileges of the user. So, the order of applying these decorators is important in these types of cases.

We had created two decorators named `accepts_ints` and `limit_return_value` in the previous two sections. Let us apply both of them to a function.

```
@accepts_ints
@limit_return_value
def func1(x, n):
    return x ** n
x = func1(2, n=59)
print(x)
```

Both decorators are working on this function. If any of the argument is not of `int` type or if the return value is not in the range 0 to 1000, then an error will be raised. Let us see one more example.

The following function takes in an email id, and extracts and returns the username from it.

```
def username(email_id):
    return email_id[:email_id.index('@')]
u1 = username('john@xmail.com')
u2 = username('jack@zmail.com')
print(u1, u2)
```

Output-

```
john jack
```

We have a decorator named `trace` that decorates a function by printing function name, its arguments and the return value. This type of decorator can be useful while debugging a stack of function calls.

```
def trace(fn):
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
```

```

        print(f'args : {args}   kwargs : {kwargs}'
)
        result = fn(*args, **kwargs)
        print(f'Return value : {result}\n')
        return result
    return wrapper

```

If we apply this decorator to our username function, the documenting code will be run whenever the function is executed.

```

@trace
def username(email_id):
    return email_id[email_id.index('@')]
u1 = username('john@xmail.com')
u2 = username('jack@zmail.com')
print(u1, u2)

```

Output-

```

username called
args : ('john@xmail.com',)   kwargs : {}
Return value : john
username called
args : ('jack@zmail.com',)   kwargs : {}
Return value : jack
john jack

```

Now suppose we have another decorator named capitalizer that capitalizes the return value of the function.

```

def capitalizer(fn):
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        return result.upper()
    return wrapper

```

Here we are returning result in uppercase. Let us apply this decorator to our username function.

```
@capitalizer
def username(email_id):
    return email_id[email_id.index('@')]
u1 = username('john@xmail.com')
u2 = username('jack@zmail.com')
print(u1, u2)
```

Output-

JOHN JACK

Now let us apply both the decorators together.

```
@capitalizer
@trace
def username(email_id):
    return email_id[email_id.index('@')]
u1 = username('john@xmail.com')
u2 = username('jack@zmail.com')
print(u1, u2)
```

Output-

```
username called
args : ('john@xmail.com',)  kwargs : {}
Return value : john
username called
args : ('jack@zmail.com',)  kwargs : {}
Return value : jack
JOHN JACK
```

Both the decorators were run but the documentation is wrong. Value returned by the function is in capital letters, but the documentation is showing it in small letters. Let us reverse the order of the two decorators.

```
@trace
```

```
@capitalizer
def username(email_id):
    return email_id[:email_id.index('@')]
u1 = username('john@xmail.com')
u2 = username('jack@zmail.com')
print(u1, u2)
```

Output-

```
wrapper called
args : ('john@xmail.com',)  kwargs : {}
Return value : JOHN
wrapper called
args : ('jack@zmail.com',)  kwargs : {}
Return value : JACK
JOHN JACK
```

Now we get the correct documentation, so the order of decoration can matter sometimes.

In this case, we can see a small difference in output. The name `wrapper` is printed instead of the function name. It happened because metadata of a function is lost when a function goes through a decorator. We will see how to fix this in the next section.

18.15 Preserving metadata of a function after decoration

When a function is decorated, all the calls to that function are replaced by calls to the wrapper function returned by the decorator. This leads to loss of metadata of the original function which can be useful in introspection. Let us see what we can do to retain the identity of the original function and preserve information about it even after decoration.

We saw the decorator named `trace` and the function `username` in the last section.

```

def trace(fn):
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
        print(f'args : {args}  kwargs : {kwargs}')
    )

    result = fn(*args, **kwargs)
    print(f'Return value : {result}\n')
    return result

    return wrapper

def username(email_id):
    """Returns the username from an email id"""
    return email_id[:email_id.index('@')]

u1 = username('john@xmail.com')

```

Right now, we have not applied the decorator to the function username. After executing this code, let us see some metadata of the function username.

```

>>> username.__name__
'username'
>>> username.__doc__
'Returns the username from an emailid'
>>> help(username)
Help on function username in module __main__:
username(email_id)
    Returns the username from an emailid
>>> username
<function username at 0x000001415BB37E20>

```

Now let us apply the decorator to this function and again check the metadata after executing the same code.

```

@trace
def username(email_id):

```



```

    """Returns the username from an emailid"""
    return email_id[:email_id.index('@')]
>>> username.__name__
'wrapper'
>>> username.__doc__
>>> help(username)
Help on function wrapper in module __main__:
wrapper(*args, **kwargs)
>>> username
<function trace.<locals>.wrapper at
0x0000014D5B5F7EB0>

```

The `__name__` attribute has changed because after decoration, the name `username` refers to the function object created inside the decorator and so it is showing the `__name__` attribute of the `wrapper` function. This `wrapper` has no docstring, so no docstring is displayed when we check the `__doc__` attribute. All this output proves that `username` is not referring to the original function, after decoration it refers to the `wrapper` function. And this is why the metadata of the original function is lost after decoration, what we are seeing is the metadata of the `wrapper` function. This can lead to unwanted behaviour by debugging tools that perform introspection by using this metadata.

If we do not want the original function to lose its name, documentation and other attributes, even after decoration, then we can use the `wraps` function from the `functools` module. This `wraps` function is a decorator that copies the introspection details of a function to another function. We will apply this decorator to the `wrapper` function.

```

from functools import wraps

def trace(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')

```

```

        print(f'args : {args}   kwargs : {kwargs}')
    )

    result = fn(*args, **kwargs)
    print(f'Return value : {result}\n')
    return result

return wrapper

```

This wraps decorator is little different, it is a decorator that takes an argument. We will see such decorators in the next section. We send function `fn` as argument to this wrapper function. So now all the important metadata of `fn` will be copied in the wrapper function.

Now after executing the modified code, if we check the attributes of `username`, we will see that the metadata is not lost after decoration.

```

>>> username.__name__
'username'
>>> username.__doc__
'Returns the username from an emailid'
>>> help(username)
Help on function username in module __main__:
username(email_id)
    Returns the username from an emailid
>>> username
<function username at 0x00000145EAAE7EB0>

```

So, if you want to preserve the original function's metadata, then you can use the `functools.wraps` decorator from the standard library.

18.16 General template for writing a decorator

Here is the general pattern for writing a decorator.

```

from functools import wraps

```

```
def decorator_name(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        #Place code that has to be executed before
function call
        result = fn(*args, **kwargs)
        #Place code that has to be executed after
function call
        return result
    return wrapper
```

The decorator takes a function as argument and defines a wrapper function inside and returns that wrapper. We make the wrapper function take args and kwargs so that we can decorate a function irrespective of the number of arguments it takes. Inside wrapper we call the undecorated function and store its return value and then return this return value from the wrapper.

Before and after the execution of the undecorated function we can place the decoration code.

The wrapper function is decorated with the wraps decorator from functools module so that the metadata of the undecorated function is not lost.

18.17 Decorators with parameters

In [Section 18.12](#), we had created the decorator named limit_return_value to specify the limits of return value. If this decorator is applied to a function and that function returns a value that is not between 0 and 1000, then a ValueError will be raised.

```
from functools import wraps
def limit_return_value(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
```

```

        result = fn(*args, **kwargs)
        if result < 0 or result > 1000:
            raise ValueError('Value returned is
not within limits')
        return result
    return wrapper
@limit_return_value
def func1(x, n):
    return x ** n

```

Now suppose we have two more functions named `func2` and `func3` and we want to put a check on the return value of these function also. For `func1`, we want the return value to be between 100 and 500 and for `func2` we want the return value to be between 0 and 5000.

```

def func2(m, n):
    return m + n
def func3(m, n):
    return m * n

```

If we apply our `limit_return_value` decorator, then it will not work because we have hardcoded the values 0 and 1000 in the decorator. So now what should we do, should we define 2 more decorators with the same code but different lower and upper limits, and what if we want a fourth set of limits. We just cannot keep on defining decorators that have the same code except for the values of limits. We need to add parameters to our decorator so that we can send different limits as arguments. Here is the approach that comes naturally to our mind.

```

from functools import wraps
def limit_return_value(fn, lower_limit,
upper_limit):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)

```

```

        if result < lower_limit or result >
upper_limit:
            raise ValueError('Value returned is
not within limits')
        return result
    return wrapper
def func2(m, n):
    return m + n
func2 = limit_return_value(func2, 100, 500)
func2(2, 90)

```

This approach will work as long as we manually do the decoration using the reassignment technique. In our example, we have called the `limit_return_value` decorator function with the extra two arguments that specify the limits.

However, this approach will not work if we use the automatic decoration syntax. In this syntax, after the `@` sign we need to provide a reference to function object that can be called with a function as argument. So instead of this approach, another approach is used in which we define a decorator factory that will give us different decorators depending on the arguments that we provide. The decorator factory is simply a function that returns a decorator. So now let us make a decorator factory instead of a single decorator.

```

def limit_return_value(lower_limit, upper_limit):
    def actual_decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            result = fn(*args, **kwargs)
            if result < lower_limit or result >
upper_limit:
                raise ValueError('Value returned
is not within limits')
            return result
    return actual_decorator

```

```
        return wrapper
    return actual_decorator
```

Now the function `limit_return_value` is a decorator factory so its work is to create and return a decorator. We can see that this decorator factory function creates the actual decorator using the `def` statement and then returns it. Inside the decorator function, the regular work of defining a wrapper and returning it is done. The code is exactly the same as was in our single decorator, only the numbers 0 and 1000 have been replaced by `lower_limit` and `upper_limit`.

Now we have nested closures. The outermost function is `limit_return_value`, it has the inner function

`actual_decorator` and inside this function we have the innermost function `wrapper`. We know that the parameters of the outer functions can be accessed inside the inner functions. So `lower_limit` and `upper_limit` can be used inside the `wrapper` function.

The decorator factory will give us an actual concrete decorator depending on the values of these limits. Now let us see how to use this factory.

The `limit_return_value` is a function that gives us back a decorator, so we need to call it to get a decorator. The expression `limit_return_value(0, 1000)` will give us a decorator in which `lower_limit` will be 0 and `upper_limit` will be 1000. Similarly the expression `limit_return_value(5, 50)` will give us a decorator in which `lower_limit` will be 5 and `upper_limit` will be 50.

In the automatic decoration syntax, we can place this call after the `@` sign, to apply the returned decorator to a function.

```
@limit_return_value(0, 1000)
def func1(x, n):
    return x ** n
```

So now the decorator returned by the call `limit_return_value(0, 1000)` is applied to this function. For decorating functions `func2` and `func3`, we can call the decorator factory with different arguments.

```

@limit_return_value(100, 500)
def func2(m, n):
    return m + n
@limit_return_value(0, 5000)
def func3(m, n):
    return m * n

```

If we have to apply this decorator manually using the reassignment technique, then we have to write it this way.

```

def func3(x, n):
    return x * n

func3 = limit_return_value(0, 5000)(func3)

```

First we get a decorator with the call `limit_return_value(0, 5000)`, and then we call that decorator with `func3` as argument.

18.18 General template for writing a decorator factory

In the previous section we saw that the decorator factory returns the actual decorator which is applied to the function that has to be decorated. Here is the general pattern for creating a decorator factory.

```

from functools import wraps

def decorator_name(parameter1, parameter2, ...):
    def actual_decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            #Place code that has to be executed before
            function call
            result = fn(*args, **kwargs)
            #Place code that has to be executed after
            function call

```

```
        return result
    return wrapper
return actual_decorator
```

The decorator factory defines and returns the actual decorator, the decorator defines the wrapper function and returns it. Inside the wrapper, the function to be decorated is executed along with the extra statements and the return value of this function is returned from the wrapper. So, the decorator factory returns actual decorator; actual decorator returns wrapper and wrapper returns the return value of the original function. The parameters of the decorator factory function can be used inside the actual decorator function or the wrapper function.

18.19 Decorator factory example

Let us see one more example of creating a decorator factory. The following code is for the decorator named `trace` that we have seen before. It prints the name, arguments and the return value of the function that is being decorated.

```
def trace(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
        print(f'args : {args}   kwargs : {kwargs}')
    )
    result = fn(*args, **kwargs)
    print(f'Return value : {result}\n')
    return result
    return wrapper
```

We can enable or disable this tracing by taking a parameter named `active`, if the parameter `active` is `False`, then the `trace` decorator will not print the tracing information of the call, and if it is `True` then the tracing information will be printed. Here is the code for the decorator factory.


```

def trace(active=True):
    def actual_decorator(fn):
        if active:
            @wraps(fn)
            def wrapper(*args, **kwargs):
                print(f'{fn.__name__} called')
                print(f'args : {args}  kwargs :
{kwargs}')

                result = fn(*args, **kwargs)
                print(f'Return value :
{result}\n')

                return result

            return wrapper
        else:
            return fn

    return actual_decorator

```

This decorator factory takes an argument named `active` with a default value of `True`. If `active` is `True` then we will define and return the `wrapper` function otherwise we will return the original function. Now let us apply the decorator returned by this decorator factory to different functions.

```

@trace(active = False)
def func1(x, n):
    return x ** n

@trace()
def func2(m, n):
    return m * n

@trace(True)

```

```
def func3(m, n):
    return m + n
x = func1(2, 4)
y = func2(10, 3)
z = func3(2, 4)
print(x, y, z)
```

Output-

```
func2 called
args : (10, 3)  kwargs : {}
Return value : 30
func3 called
args : (2, 4)  kwargs : {}
Return value : 6
16 30 6
```

We can define a variable at the top and ask the user if tracing is required or not.

```
tracing_option = input('Want to trace function
calls(yes/no) : ')
tracing_option = True if tracing_option.lower() ==
'yes' else False
@trace(tracing_option)
def func1(x, n):
    return x ** n
```

18.20 Applying decorators to imported functions

We can even apply our own decorators to the functions that we import from standard library or third-party packages. However, we cannot use the `@` syntax for these functions.

```

from math import factorial
from random import randint
def trace(fn):
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
        print(f'args : {args}   kwargs : {kwargs}')
    )
    result = fn(*args, **kwargs)
    print(f'Return value : {result}\n')
    return result
    return wrapper
factorial = trace(factorial)
randint = trace(randint)
factorial(3)
randint(5, 50)

```

Output-

```

factorial called
args : (3,)   kwargs : {}
Return value : 6
randint called
args : (5, 50)   kwargs : {}
Return value : 14

```

18.21 Decorating classes

We have seen how to create decorators that can decorate functions, in this section we will see how to create decorators that can decorate classes. When we talk about decorating classes, we can either decorate individual methods or we can create a decorator to decorate the whole class.

```

class MyClass:

```

```

def __init__(self, a):
    self.a = a

@timer
def method1(self, x, y):
    print('method1 executing')

@timer
def method2(self, x, y):
    print('method2 executing')

@trace
def method3(self, x, y):
    print('method3 executing')

```

Decorating methods is straightforward, it is like decorating functions only. For example, the decorators `trace` and `timer` that we have created in our lectures, can be applied to methods of a class to trace or time the method calls. While applying the decorators to methods of a class, we need to keep in mind that a method always receives the current instance as the first argument. The decorators `trace` and `timer` that we had created, will work properly with methods also because they are not doing anything special with the first argument but suppose you have a decorator that sanitizes the first argument of a function then you cannot simply apply that decorator to a method, because for methods, the current instance is the first argument, and the first argument that you send when you call the method is actually the second argument.

Now, let us see how to create a decorator function to decorate the class as a whole. This type of decorator function will take a class as the argument. It will either modify that class and return it, or it will create a new class and return it. Modifying the class in-place and returning the modified class is more convenient than creating a new class. So, we will see some examples where we will create a decorator that takes a class and returns the modified class.

The syntax for applying the decorator to a class will be the same. You can either use the automatic way or decorate the class manually.

```
@decorator
```

```
class MyClass:
    pass
class MyClass:
    pass
MyClass = decorator(MyClass)
```

First, let us create a very simple decorator, which, when applied to a class, adds a new attribute named `author` and a line to the docstring of the class.

```
def my_decorator(cls):
    if cls.__doc__ is None:
        cls.__doc__ = '\nThis is an important
class\n'
    else:
        cls.__doc__ += '\nThis is an important
class\n'
    cls.author = 'Ryan'
    return cls
@my_decorator
class Person:
    """This is the docstring of Person class"""
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def speak(self):
        print(f'Hello, I am {self.name}')
print(Person.__doc__)
print(Person.__dict__)
class Car:
    def __init__(self, model, max_speed):
        self.model = model
        self.max_speed = max_speed
```

```

        def show(self):
            print(f'{self.model}, {self.max_speed}')
Car = my_decorator(Car)
print(Car.__doc__)
print(Car.__dict__)

```

Output-

This is the docstring of Person class

This is an important class

```
{'__module__': '__main__', ..... , 'author':
'Ryan'}
```

This is an important class

```
{'__module__': '__main__', ..... , 'author':
'Ryan'}
```

Our decorator takes in a class as the argument, so we have named its parameter `cls`. If the docstring of the class is `None`, then we assign a string to the docstring; otherwise, we add the string to the docstring. After this we add a new attribute named `author` to the class. At last, we return the class from the decorator.

We have applied this decorator to the two classes named `Person` and `Car`. The `Person` class has been decorated using the automatic syntax, while the class `Car` has been decorated using the manual decoration syntax. The output clearly shows the effect of decoration.

In our next example, we have created a decorator that adds an attribute named `time_of_creation` to each instance of the class. Note that in the previous example, we added an attribute to the class object, so we had actually created a class variable. Now, we want to add an attribute to each instance object, so we will be creating an instance variable. This attribute, named `time_of_creation`, will store the time when the instance object is created. The decorator will also print a message whenever a new instance object is created.

```

def add_creation_time(cls):
    init = cls.__init__

```

```

    def new_init(self,*args, **kwargs):
        from time import ctime
        self.time_of_creation = ctime()
        print(f'A new object of type {cls.__name__ }
created')
        init(self,*args, **kwargs)
    cls.__init__ = new_init
    return cls
@add_creation_time
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def speak(self):
        print(f'Hello, I am {self.name}')
@add_creation_time
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color
    def show(self):
        print(f'{self.model}, {self.max_speed}')
bob = Person('Bob', 23)
tom = Person('Tom', 66)
x = Car('Audi R8', 'White')
y = Car('Jaguar XJ', 'Black')
print(bob.time_of_creation)
print(tom.time_of_creation)
print(x.time_of_creation)
print(y.time_of_creation)

```

Output-

A new object of type Person created

A new object of type Person created

A new object of type Car created

A new object of type Car created

Tue Aug 22 16:57:32 2023

Tue Aug 22 16:57:32 2023

Tue Aug 22 16:57:32 2023

Tue Aug 22 16:57:32 2023

Let us understand the code that is written inside the decorator. We have to print the message and add the attribute when an instance is created, so we will have to change the `__init__` method. First, we save the `__init__` in a separate variable named `init`. Then we define a function named `new_init`. The first argument is `self` and then we place `args` and `kwargs`.

After that, we imported the `ctime` function from the `time` module. We add a new attribute named `time_of_creation`. After that, we print the message, that a new object has been created. Next, we call this original `__init__` that we have saved in the variable `init`. The extra work has been done before calling the original `init`. Then we assign the `new_init` method to the `__init__` attribute of the class. And at last, we return the class object.

We have applied this decorator to the classes `Person` and `Car`. The output shows that when instance objects of these classes are created, a message is displayed. A new attribute named `time_of_creation` gets attached to each instance, and its value is the time when the instance object is created.

In our next example, we have a decorator function named `counter`, which when applied to a class, creates a class variable named `count` that keeps track of the number of instances created. This decorator also attaches an instance variable named `serial_number` to each instance. For the first

object that is created, `serial_number` is 1, for the second object it is 2 and so on.

```
def counter(cls):
    cls.count = 0
    init = cls.__init__
    def new_init(self,*args, **kwargs):
        cls.count += 1
        self.serial_number = cls.count
        init(self,*args, **kwargs)
    cls.__init__ = new_init
    cls.__lt__ = lambda self, other:
self.serial_number < other.serial_number
    cls.__gt__ = lambda self, other:
self.serial_number > other.serial_number
    return cls

@counter
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def speak(self):
        print(f'Hello, I am {self.name}')

@counter
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color
    def show(self):
        print(f'{self.model}, {self.max_speed}')

bob = Person('Bob',23)
```

```

tom = Person('Tom', 66)
sam = Person('Sam', 45)
jim = Person('Jim', 56)
x = Car('Audi R8', 'White')
y = Car('Jaguar XJ', 'Black')
z = Car('Toyota Glanza', 'Blue')
L = [tom, bob, jim, sam]
print(Person.count, Car.count)
print(sam.serial_number, x.serial_number)
for person in sorted(L):
    print(person.name, end=' ')

```

Output-

```

4 3
3 1
Bob Tom Sam Jim

```

In the decorator, first, we add an attribute named `count` to the class. As in the previous example, here also we have to attach an instance variable so we will have to define a new initializer method. For that, first, we save the original `__init__` and define a new initializer which is assigned to `cls.__init__`. Inside the new initializer, we increase `count` by 1, so whenever a new instance variable is created, this class variable `count` will be increased by 1. After this, we create an instance variable named `serial_number`. Then we call the original `__init__`. The new `__init__` is assigned to the `__init__` attribute of `cls`.

We have also defined the `__lt__` and `__gt__` methods to make the instance objects sortable on `serial_number`. Finally, we return `cls`. We have applied the decorator to the classes `Person` and `Car`. Four instance of `Person` class and three instances of `Car` class are created.

The class variable `count` has been attached to both classes and each instance gets an instance variable `serial_number`. The list of `Person` objects is sorted on `serial_number` attribute.

In our next example, we have created a decorator that adds a new method to each class that it decorates.

```
def introducer(cls):
    def introduce_yourself(self):
        print(f'I am an instance object of
{self.__class__.__name__} ')
        print(f'My id is {hex(id(self))}')
        print(f'Here are my attributes - ')
        for key,value in self.__dict__.items():
            print(f'{key} : {value}')
        cls.introduce_yourself = introduce_yourself
    return cls

@introducer
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def speak(self):
        print(f'Hello, I am {self.name}')

@introducer
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color
    def show(self):
        print(f'{self.model}, {self.max_speed}')

bob = Person('Bob',23)
tom = Person('Tom',66)
sam = Person('Sam',45)
x = Car('Audi R8','White')
```

```
y = Car('Jaguar XJ','Black')
z = Car('Toyota Glanza','Blue')
bob.introduce_yourself()
y.introduce_yourself()
```

Output-

I am an instance object of Person

My id is 0x278023cab00

Here are my attributes -

name : Bob

age : 23

I am an instance object of Car

My id is 0x278023cb790

Here are my attributes -

model : Jaguar XJ

color : Black

Inside the decorator we have defined a function named `introduce_yourself`, and added it as a new attribute to the class. We have defined this function here exactly like we would have defined a method inside the class. We have written `self` as the parameter; inside the body some information about the instance object `self` is printed. At last, as usual, we are returning `cls`. This decorator is applied to the two classes `Person` and `Car`. So now we have an additional instance method attached to these classes.

We can place the function `introduce_yourself` outside the decorator also, so it will not be executed each time this decorator decorates a class.

```
def introduce_yourself(self):
    print(f'I am an instance object of
{self.__class__.__name__} ')
    print(f'My id is {hex(id(self))}')
    print(f'Here are my attributes - ')
    for key,value in self.__dict__.items():
```

```

        print(f'{key} : {value}')
def introducer(cls):
    cls.introduce_yourself = introduce_yourself
    return cls

```

In our next example, we have a decorator named `trace` which when applied to a function prints the trace information.

```

from functools import wraps
def trace(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
        print(f'args : {args}   kwargs : {kwargs}')
    )
        result = fn(*args, **kwargs)
        print(f'Return value : {result}\n')
        return result
    return wrapper

```

Suppose we want to apply this `trace` decorator to all the methods of a class. We can simply do it by placing the `@trace` line before each method. But if the class has many methods, then it is too much of work, and if in future a new method is added and we forget to apply this decorator then that method will not be traced. So, we can write a decorator that will decorate the whole class such that the `trace` decorator is applied to each method.

```

def tracer(cls):
    for attr_name, value in cls.__dict__.items():
        if callable(value):
            setattr(cls, attr_name, trace(value))
    return cls

@tracer
class MyClass:

```

```

def __init__(self, name, a, b):
    self.name = name
    self.a = a
    self.b = b
def method1(self, x):
    return self.a + x
def method2(self, message):
    print(message + self.name)
a = MyClass('ABC', 1, 2)
a.method1(4)
a.method2(message = 'Hello ')

```

The class decorator named `tracer` takes in a class as argument. To decorate the methods of class, you need to find the name of the methods. Methods are callable attributes of a class object, so you can use built in `callable` function which returns `True` if an attribute is callable.

We iterate over all the attributes of the class object, if an attribute refers to a callable object, then we make that attribute refer to the decorated callable. When the class decorator `tracer` is applied to a class, the `trace` decorator will be applied to all the callable attributes of that class. If you do not want to apply this decorator `trace` to dunder methods, then you can place a condition for that.

```

def tracer(cls):
    for attr_name, value in cls.__dict__.items():
        if callable(value) and
not(attr_name.startswith("__")          and
attr_name.endswith("__") ):
            setattr(cls, attr_name, trace(value) )
    return cls

```

So now if an attribute is callable and is not a dunder method, then only the decorator will be applied. For instance, methods, the first argument is

always the instance object. If you do not want the instance object to be displayed, then you can use a slice (`{args[1:]}`) in the `trace` function.

```
def trace(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f'{fn.__name__} called')
        print(f'args : {args[1:]}  kwargs : {kwargs}')
        result = fn(*args, **kwargs)
        print(f'Return value : {result}\n')
        return result
    return wrapper
```

18.22 Class Decorators

So far, we have used only functions as decorators. We have used functions to decorate functions and to decorate classes. Now, we will see how to define a class as a decorator.

At the start of this chapter, in the definition of decorator, we had seen that a decorator is a callable; a callable is any object that can be called a function. If a class has the `__call__` method defined inside it, then instance objects of that class become callable objects. When these instance objects are called like a function using parentheses, the code inside the `__call__` method is executed. Now, let us see how we can use such a class as a decorator.

```
class MyDecorator:
    def __init__(self, fn):
        self.fn = fn
    def __call__(self, *args, **kwargs):
        print('Decoration before execution of function')
        self.fn(*args, **kwargs)
```

```

        print('Decoration after execution of
function\n')
def func(message, name):
    print(message, name)
func = MyDecorator(func)

```

The class `MyDecorator` is going to serve as a decorator. We have a simple function named `func` and we have decorated it using the class `MyDecorator` by using the manual decoration syntax. In the statement `func = MyDecorator(func)`, we are sending the undecorated function to the class and getting back the decorated function. We know that when we call a class object (`MyDecorator(func)`), the `__init__` method of the class is executed. This is why we have made `__init__` method accept a function. The call `MyDecorator(func)` returns an instance object of the class. So, after the statement `func = MyDecorator(func)`, `func` is actually an instance object of the `MyDecorator` class.

```

>>> func

<__main__.MyDecorator object at
0x00000175DAE124D0>

```

After the assignment, `func` no longer refers to the function object created by `def func()`. It refers to the instance object created by the expression `MyDecorator(func)`. So now the name `func` is an instance object, and this instance object is callable, since we have the `__call__` method defined inside its class. We can call this instance object like a function:

```

>>> func('hello', 'bob')
Decoration before execution of function
hello bob
Decoration after execution of function
>>> func('hi', 'tom')
Decoration before execution of function
hi tom
Decoration after execution of function

```


These calls actually execute the code of the `__call__` method. Inside the `__call__` method, we have placed the undecorated function call and the decoration code.

So, this is how we can create a decorator class. The second parameter to `__init__` accepts the function to be decorated, and this function is stored as an instance variable. Inside the `__call__` method the original undecorated function is called and the decoration code is also placed before or after the call. The class is instantiated at the decoration time, and the instance object that is created is assigned to the function name. After decoration, the function name refers to an instance object, and so when the function name is called, the `__call__` method of the class is executed.

Instead of using the manual decoration statement, we can use the automatic decoration syntax to get the same effect.

```
@MyDecorator
def func(message, name):
    print(message, name)
```

You can return the result from the `__call__` method, so that the return value of the original function is not lost.

```
def __call__(self, *args, **kwargs):
    print('Decoration before execution of
function')
    result = self.fn(*args, **kwargs)
    print('Decoration after execution of
function\n')
    return result
```

If you want to preserve the metadata, then you have to call the `wraps` decorator.

```
def __init__(self, fn):
    self.fn = fn
    wraps(fn)(self)
    self.author = 'Jim'    # Add a new attribute
here
```

If you want the decorator to add an attribute to the function, you have to add it inside the `__init__` method, not inside the `__call__` method. This is because we do not want to add the attribute each time the function is called. We want to add the attribute just once when the function is decorated.

18.23 Class Decorators with parameters

Now we will see how to make a class decorator that takes arguments. This is very different from the decorator without arguments that we saw in the previous section. The `__init__` method will receive the arguments that the decorator has to take, it will not receive the undecorated function. The undecorated function is received by the `__call__` method.

```
class MyDecoratorWithArgs:
    def __init__(self, a1, a2):
        self.a1 = a1
        self.a2 = a2
        print(f'Decorator arguments {a1}, {a2}')
    def __call__(self, fn):
        def wrapper(*args, **kwargs):
            print('Executed Before function call')
            result = fn(*args, **kwargs)
            print('Executed After function
call\n')
            return result
        return wrapper

@MyDecoratorWithArgs('Hello', 10)
def func1():
    print('func1 executing')

@MyDecoratorWithArgs(10, 12)
def func2(a, b):
    print('func2 executing')
```

```

def func3():
    print('func3 executing')
func3 = MyDecoratorWithArgs(3,5)(func3)
func1()
func2(1, 2)
func3()

```

Output-

```

Decorator arguments Hello, 10
Decorator arguments 10, 12
Decorator arguments 3, 5
Executed Before function call
func1 executing
Executed After function call
Executed Before function call
func2 executing
Executed After function call
Executed Before function call
func3 executing
Executed After function call

```

The `__call__` method will be executed during the decoration time. So, when the decoration is done, `__init__` will be executed and then `__call__` will be executed. In the `__call__` method, we are passing the function that we want to decorate. Inside `__call__`, we have defined a wrapper function. Inside this wrapper function, we have put the decoration code and the call to the undecorated function. After that, the wrapper function is returned from `__call__`.

So, the decorated function actually refers to the wrapper returned by `__call__`. In the manual decoration assignment statement `func3 =`

MyDecoratorWithArgs(3, 5)(func3), we can see that func3 is being assigned the return value of `__call__`. We have applied the decorator with different arguments to different functions.

Exercise

What will be the output of the code given in questions 1 to 8?

```
1. def func():
    def g():
        print('Hello')
    return g()
g()

2. def my_decorator(fn):
    def wrapper():
        print('Hello', end = ' ')
        fn()
    return wrapper
def func1():
    print('Welcome to Python')
func1 = my_decorator(func1)
func1()

3. def my_decorator(fn):
    def wrapper():
        fn()
        print('@')
    return wrapper
@my_decorator
```

```

def func1():
    print('Learning Decorators',end=' ')
func1()
4. def my_decorator(fn):
    def wrapper(s):
        fn(s)
    return wrapper
@my_decorator
def func1(s):
    return s.upper()
print(func1('abcd'))
5. def my_decorator(fn):
    def wrapper(s):
        return tuple(fn(s))
    return wrapper
@my_decorator
def func1(s):
    return s.split()
print(func1('Welcome to Python'))
6. def my_decorator(fn):
    def wrapper():
        print('Hello', end = ' ')
        fn()
    return wrapper
@my_decorator

```

```
def func1():
    print('Welcome to Python')
func1 = my_decorator(func1)
func1()
```

```
7. def counter(func):
    count = 0
    def wrapper():
        nonlocal count
        count += 1
        print(count, end = ' ')
        func()
    return wrapper

@counter
def func1():
    print('Hello', end= '. ')

func1()
func1()
func1()
```

```
8. def my_decorator(fn):
    def wrapper():
        fn()
    return wrapper

def func1():
    print('Welcome to Python')

i1 = hex(id(func1))
```

```
func1 = my_decorator(func1)
i2 = hex(id(func1))
print(i1 == i2)
```

9. We cannot apply our decorators to functions that are imported from the standard library.
(A) True (B) False
10. Write a decorator which ensures that the first argument received by the function is a string.
11. Write a decorator which ensures that the return value of a function is not zero.
12. Write a decorator that changes all the string arguments to lowercase and all integer arguments to their absolute values.
13. Write a decorator that makes a function accept only keyword arguments. Is there any other way of achieving this?
14. Write a decorator that converts the return value from a list to a string that contains all the values of that list separated by commas. Write another decorator that converts the return value from a comma-separated string of values to a list.
15. Write a decorator that appends a line to the docstring of a function. This will be a simple decorator that does not need to define any inner function. It should just modify the docstring and return the original function back.
16. Write a decorator that adds a new attribute named `author` to any function it decorates.
17. Write a decorator named `timer` to calculate the time taken by a function to execute. Apply this decorator to the built in `sum` function.
18. Write a decorator that executes the function after a delay of 5 seconds.
19. Modify the `add_to_docstring` decorator that you had created in the question 15 so that it can add different strings to docstrings of different functions.

20. Modify the decorator `delay_execution` that you had created in the question 18, such that it delays execution for a specified number of seconds.
21. Write a decorator factory that takes variable number of positional arguments. These arguments denote the allowed values for the first argument of the function that is to be decorated.

```
@first_arg_can_be(1, 2, 3, 6, 9)
def func1(x, y, z):
    print(x, y, z)
func1(6, 67, 34)
func1(8, 67, 34)  # this call gives error
@first_arg_can_be('cut', 'copy', 'paste')
def func2(a, b):
    print(a, b)
func2('cut', 2)
func2('yes', 2)  # this call gives error
@first_arg_can_be('user', 'admin')
def func3(m, n):
    print(m, n)
func3('admin', 2)
func3('new', 2)  # this call gives error
```

22. Modify the following decorator such that it takes as argument the file to which the information is written. The default value for the parameter should be `log.txt`.

```
def logger(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
```



```

        from time import ctime
        with open('log.txt', 'a') as fout:
            fout.write(f'{fn.__name__} called
at {ctime()}\n')
        return fn(*args, **kwargs)
    return wrapper

```

23. In question 16, we had written a decorator that adds a new attribute named `author` to a function. Now write a decorator that can add any number of new attributes to a function.

24. Modify the following decorator so that it can be enabled or disabled.

```

def timer(fn):
    def wrapper(*args, **kwargs):
        from time import time
        start = time()
        result = fn(*args, **kwargs)
        end = time()
        print(f'{fn.__name__} took {end -
start} seconds')
        return result
    return wrapper

```

25. Write a decorator that restricts the return value of a function to a few types.

```

@returns(int, float)      # function can return
an int or float only
def func1():
    return 1.2

@returns(list, tuple)     # function can return
a list or tuple only

```

```

        return [1,3]

@returns(str)    # function can return a string
only
def func3():
    return 'abc'

```

26. In the chapter, we had created this decorator function to count the number of calls.

```

def counter(fn):
    def wrapper():
        wrapper.number_of_calls += 1
        return fn()
    wrapper.number_of_calls = 0
    return wrapper

```

Instead of this function, create a decorator class for the same purpose.

27. In a class decorator, how will you copy the metadata of the original function using the `wraps` decorator.
28. Write a decorator class named `Logger` that logs a function call to a file. Take a class variable named `log_file` and give it the value `'log.txt'`.
29. Change the class decorator `Logger` such that it takes the name of the log file as argument.
30. Change the `Logger` class made in the previous question so that it can be disabled or enabled.
31. Write a class decorator named `Accepts` that takes arguments; these arguments specify the types of the arguments that the decorated function can take.

Lambda Expressions and Functional Programming

19

19.1 Lambda expression

In the chapter on functions, we had learnt that `def` is an executable statement which when executed creates a function object and assigns it to the function name. For example, when the following `def` statement is executed, a function object is created and is assigned to the name `factorial`.

```
def factorial(n):  
    fact = 1  
    while n > 0:  
        fact *= n  
        n -= 1  
    return fact
```

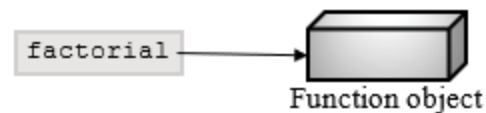


Figure 19.1: Function object created by `def` statement

We know that function objects are first-class objects in Python, they can be sent as an argument to a function or they can be returned from a function. We have seen their different attributes also.

Apart from `def` statement, we have one more tool in Python that gives us a function object. It is called lambda expression. Here is an example of a lambda expression:

```
lambda x, y: x + y
```

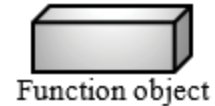


Figure 19.2: Function object created by lambda expression

Do not worry about the syntax now, just understand that when this expression is executed, it gives a function object. It is similar to a list comprehension expression giving a list object or a generator expression giving a generator object.

The function object produced by `lambda` expression is not given any name as it is done in the `def` statement. That is why these are also called anonymous (unnamed) functions. The `lambda` expression, `lambda x, y: x + y` is like a simple function that takes two arguments and returns the expression `x + y`. When we define a function using `def`, we have a name so we can call the function using the name, or we can send the name as an argument to another function, but `lambda` expression gives us a nameless function object, so now the question is how do we call it. Here is a way in which it can be called:

```
>>> (lambda x, y: x + y) (2, 3)
5
```

The expression inside parentheses gives us a function object, and in front of this function object, we have placed a pair of parentheses with 2 values, so effectively, we are calling the function object with the arguments 2 and 3. The arguments 2 and 3 are assigned to parameters `x` and `y` and the value `2 + 3` is returned. So, when we execute the whole expression, we get a value of 5. But this form does not make much sense, `lambda` expression is not used like this.

Another way to use the function object returned by `lambda` is to do what the `def` statement does i.e. assign the function object a name.

```
>>> add = lambda x, y: x + y
```

After this assignment statement, the name `add` refers to the function object produced by this `lambda`, and now you can invoke the function object by using this name.

```
>>> add(2, 3)
```

So now the name `add` is exactly like a function name, you can use it like you use a function name defined by `def`. But again, this is also not the way it is used. The most common use of a `lambda` expression is sending it as an argument to another function.

```
>>> func(5, 6, lambda x,y: x+y)
```

Here we are sending a `lambda` expression as an argument to a function call. Before seeing the details and advantages of this usage of `lambda` expressions, let us first see the syntax and some examples of `lambda` expressions. Here is the syntax of a `lambda` expression:

```
lambda parameter1, parameter2, .....: expression
```

The `lambda` keyword is followed by an optional parameter list; these parameters are not inside parentheses. After the parameters, we have a colon. This colon separates the parameter list from the body of the function. The body of the function here is limited to a single expression. When the function object returned by the `lambda` expression is called, code in the expression is evaluated and the value of the expression is returned. There is no explicit `return` keyword, the value of the expression is returned implicitly.

So, when the `lambda` expression is executed, a function object is created and when that function object is called, the expression after the colon is evaluated and its value is returned.

19.2 Comparing `def` statement and `lambda` expression

Let us compare a `lambda` expression to a `def` statement. The first obvious difference is that `lambda` expression is an expression, and `def` is a statement. The parameters in a `lambda` expression are not enclosed in parentheses, while in a `def`, they are enclosed in parentheses.

The parameter list in a `lambda` expression allows all the type of arguments that we can use in a `def` statement. However, all those things are not used

very often since `lambda` expressions are used for very simple tasks. As in `def` statement, we can have default arguments, variable list of arguments, or keyword arguments but generally only positional arguments are used in `lambda` functions.

The `lambda` function evaluates just a single expression, while the `def` statement is capable of executing multiple statements. Inside the body of the `lambda` function, you cannot use loops, conditional statements or assignment statements. There is just one expression in the function body, and there is no `return` keyword in front of the expression. When you have some complex work to be done, you have to use `def`.

As we have already discussed, `lambda` functions are anonymous, while functions defined by `def` have a name.

Since a `def` statement is a statement and the `lambda` expression is an expression, `lambda` expression can be used in places where `def` cannot be used, like inside a list or a dictionary literal or in the argument list of a function call. In these places, a `def` statement will not work syntactically. So, `lambda` expressions enable us to define small functions inline.

Let us compare the types of objects returned by the `def` statement and `lambda` expression:

```
>>> def add(x, y):  
        return x + y  
  
>>> type(add)  
<class 'function'>  
  
>>> add  
<function add at 0x02CEE738>
```

We have defined a function named `add` by using the `def` statement, and we get a function object with the name `add` at address `0x02CEE738`. Now let us write `lambda` expressions:

```
>>> lambda x, y: x + y  
<function <lambda> at 0x0333E738>  
  
>>> lambda n: n*n  
<function <lambda> at 0x0333E737>
```

A lambda expression gives a function object with the generic name `lambda`.

```
>>> type(lambda x, y: x + y)
<class 'function'>
```

We can see that it has the same type as the name `add`, and this shows that both `def` statements and lambda expressions give us function objects.

We can assign the function object given out by lambda to a name.

```
>>> a = lambda x, y: x + y
>>> type(a)
<class 'function'>
>>> a
<function <lambda> at 0x036AE738>
```

We can check the name attribute of the function object referred to by `a`.

```
>>> a.__name__
'<lambda>'
```

It has a generic name `lambda`

Now we can use the name `a` like a regular function.

```
>>> a(3, 4)
7
```

lambda expressions are best suited for single use short functions as the function objects produced by them can be garbage collected after they have been used, unlike function objects produced by `def` statements.

19.3 Examples of lambda expressions

Here are some examples of lambda functions and their equivalent `def` statements:

```
lambda s: s.strip().upper() + ' !'          def
shout(s):
    return s.strip().upper() + ' !'
```

```

lambda alist: sum(alist)/len(alist)      def
average(alist):
    return sum(alist)/len(alist)
lambda a, b: a if a<b else b      def smaller(a, b):
    return a if a<b else b
x = 5      x = 5
lambda: sum(range(x))      def func():
    return sum(range(x))

```

Although we cannot use a conditional statement inside a lambda expression, a conditional expression is allowed. We can also have lambda expressions without parameters. The last lambda expression example is parameter-less; it uses the global variable `x` in its expression. These examples show us that a lambda expression is equivalent to a function with a single return statement, but it does not use the `return` keyword.

Function calls are expressions, so we can write function calls in the place of expression. Even functions that return `None` can be used. For example, we can use `print` function in the lambda expression.

```
>>> lambda s: print('Hello', s)
```

Here we have a call to `print` function as the expression.

```
>>> (lambda s: print('Hello', s)) ('Sir')
Hello Sir
```

The following lambda expression uses a conditional expression. It will return 0 if the parameter `x` is greater than 10, otherwise it will return 1. We have called it with argument 15, and so it returns 0.

```
>>> (lambda x: 0 if x > 10 else 1)(15)
0
```

In the chapter on functions, we saw that a function can return multiple values by separating them with commas.

```
>>> def max_min_avg(L):
    return max(L), min(L), sum(L)/len(L)
>>> marks = [92, 76, 98, 67, 88, 92, 89]
```



```
>>> maxmarks, minmarks, avgmarks =  
max_min_avg(marks)
```

We know that in the function `max_min_avg`, we are returning a tuple; the comma-separated values are packed into a tuple, and then that tuple is returned. When we call the function, we unpack the returned tuple. This automatic packing of tuple will not work in lambda expressions.

```
>>> lambda L: max(L), min(L), sum(L)/len(L)  
NameError: name 'L' is not defined
```

Here we are trying to do the same thing with a lambda expression, but we get an error. In a lambda expression, if the expression that is being returned is a tuple, then it should be enclosed within parentheses. We have to explicitly form a tuple by enclosing the values inside parentheses.

```
>>> lambda L: (max(L), min(L), sum(L)/len(L))
```

Similarly, you can return other data structures like lists or dictionaries from a lambda.

```
>>> lambda L: [max(L), min(L), sum(L)/len(L)]
```

Now we are returning a list.

Like `def` statements, lambda functions have their own local namespace, and the scope lookup rules apply for lambda expressions also. So, they can access any variable defined in the enclosing functions, modules or built-in names. The expression inside lambda generally uses the parameters but it can use global and built-in names also. Suppose `x` is a global variable:

```
>>> x = 4  
>>> lambda a: x ** a
```

We can access `x` in this lambda expression.

You cannot create new variables or change any of the variables that are accessible to lambda expression because you cannot write an assignment statement inside a lambda expression. For example, you cannot do this-

```
>>> lambda a: x=5  
SyntaxError: cannot assign to lambda
```

Loop statements are not allowed inside `lambda` expressions, but you can use comprehension expressions if you need some looping logic. For example, here, we have used a list comprehension expression:

```
>>> f = lambda x, y: [x*i for i in range(y)]
>>> f(4, 5)
[0, 4, 8, 12, 16]
```

We can also embed the `map` built-in function that we will study later.

So, we have seen the syntax and examples of `lambda` expressions, and we have compared them with the `def` statement. The `lambda` expressions are syntactically limited to just a single expression, and we already have the `def` statement, which can do the same work for us. The obvious question, now, is why do we need these `lambda` expressions. We will get the answer in the remaining chapter, where we will see some common places where the `lambda` expression is generally used.

19.4 Using Lambda expressions

We know that a `lambda` expression, when executed, gives us a function object, so we can use a `lambda` expression wherever a function object is required. Using `lambda` expressions is completely optional. You can always achieve the same result by writing an equivalent `def` statement. But they are handy when you have to write single-expression throw-away functions. A throw-away function means a function that has to be used only once, and after that, it has no use. So, if you need to write a function that has to be used only once, and the only thing the function does is return an expression, then you do not need to write a full-fledged `def` statement. It is better to write a `lambda` expression instead.

Now you would ask why we will ever need to write a function that has to be used at only one place in a program because when we are introduced to the concept of functions, we are told that functions are used to avoid code repetition; we define a function at one place and call it at multiple places in our program. The need for `lambda` expression arises when we have to send functions as arguments to higher-order functions. Let us try to understand all this with the help of examples.

When we define a function, we give it a name so that we can call the function at multiple places using that name. But when we need to use a function at only one place in the program, it can be anonymous; there is no need for a name. We can define the function where it has to be used. For instance, in the following example, we want the function named `square`, only for sending as argument to the higher-order function `func`, so there is no need to define it using `def`. We can define it using a `lambda`, in the argument list itself.

```
def square(n):  
    return n * n
```

```
.....  
.....  
.....  
func(x, y, square)      func(x, y, lambda n: n*n)  
.....  
.....
```

In the argument list of the caller, an inline `lambda` expression is embedded instead of a named function defined somewhere else in the program.

`lambda` expressions are preferred when we need to write single expression throw-away functions. Since the function has to be used only once, we can define it where it is used, and so we do not need to give it a name. `lambda` expressions can be useful in places where you want a single expression function as an argument or a return value.

We have seen earlier that a higher-order function is a function that can receive function objects as arguments or return them. There are several built-in higher-order functions, and many libraries also have higher-order functions that expect a function as an argument. The functions that need to be sent to these higher-order functions are mostly throwaway functions, they have no use other than sending them to these higher functions as arguments. So, in these cases, we can use `lambda` expressions.

We will see all this in detail, first let us see with the help of a dummy example, what happens when we send a `lambda` expression as an

argument in a function call. When we were learning function objects, we had seen the following example:

```
def subtract(a, b):  
    print(a - b)  
def add(a, b):  
    print(a + b)  
def multiply(a, b):  
    print(a * b)  
def divide(a, b):  
    print(a // b)  
def calculate(a1, a2, fn):  
    fn(a1, a2)  
calculate(2, 4, add)
```

The function `calculate` expects a function object as its third argument. While calling this function, we have sent the name `add` as the third argument. The name `add` refers to the function object created when the `def` statement was executed. This argument is assigned to the parameter `fn`, so now `fn` also refers to the same function object, and inside the `calculate` function, that function object is called. This is how it works; we have seen all this before.

Since the `calculate` function takes a function object as its third argument, we can send a `lambda` expression as an argument because a `lambda` expression also gives us a function object.

```
calculate(5, 3, lambda x, y: print(x + y))
```

The `lambda` expression gives a function object so a function object is sent as the third argument and this function object is assigned to the parameter `fn`, so when `fn` is called inside the function, the function object returned by `lambda` is called.

So, in the function `calculate` we can send name of any function that has two parameters or we can send any `lambda` expression with two parameters. Let us call this once more with another `lambda` expression.

```
calculate(5, 3, lambda x, y: print(2*x - y))
```

We sent another `lambda` expression here. If we do not use `lambda` expression, then we will have to write a new `def` statement just for sending a function object here.

```
def func(x, y):  
    print(2*x - y)  
calculate(5, 3, func)
```

So, we would have to write a `def` statement and think of an appropriate name for the function and then send it to the `calculate` function. If you are sure that you will not need this function anywhere else, then there is no need to give it a name and clutter your namespace. You can use a `lambda` expression which is anonymous.

So, `lambda` expressions are mostly sent as arguments to higher-order functions. When a function is passed as an argument to another function, the passed function is called a call-back function. We can say that `lambda` expressions are used in implementing callback functions.

There are some higher-order functions that are built-in like `max`, `min`, `sorted`, `map`, `filter`, and some are there in the standard library like `functools.reduce`. The `lambda` functions are often used in combination with these functions, we will see the details later in this chapter.

`lambda` expressions are handy when you want to provide a small callback function but you do not want to clutter your namespace by creating a separate `def` function. You will see and use `lambda` expressions in code that require users to provide short callback functions. Many GUI frameworks also have higher-order functions that expect function objects as arguments. Therefore, `lambda` expressions are also generally used in GUI programming and network programming.

19.5 Using `lambda` expressions for returning function objects

lambda expressions can also be used to return function objects from a function. Here is an example that we have seen earlier in the functions chapter:

```
def func(x):
    if x < 0:
        def fn():
            print('Hello')
    elif x > 0:
        def fn():
            print('Hi')
    else:
        def fn():
            print('Hey')
    return fn

f1 = func(6)
f1()
f2 = func(0)
f2()
```

Inside the function `func`, a `def` statement is executed depending on the value of `x`. At the end, a function object is returned from the function. We have called the function `func` two times, and we have assigned the returned function objects to the names `f1` and `f2`.

There is no need to define the inner functions using a `def` statement. Instead, we can use lambda expressions.

```
def func(x):
    if x < 0:
        return lambda: print('Hello')
    elif x > 0:
        return lambda: print('Hi')
    else:
```

```
return lambda: print('Hey')
```

19.6 Lambda expressions as closures

A closure is a nested function that can access free variables from an enclosing function even after it has finished its execution. We know that, like nested function definitions, lambda expressions can reference values from the enclosing scope, so lambda expressions are also useful as a closure.

In the previous example that we have seen, suppose there was one more parameter in the function.

```
def func(x, username):
    if x < 0:
        return lambda: print('Hello', username)
    elif x > 0:
        return lambda: print('Hi', username)
    else:
        return lambda: print('Hey', username)

f1 = func(6, 'Sam')
f1()
f2 = func(0, 'Tim')
f2()
```

Output-

```
Hi Sam
Hey Tim
```

We can use the parameter username inside the lambda expression, so here these lambda expressions act as closures. Here is one more example:

```
def func(symbol):
    return lambda message: print(message + symbol)

exclaim = func('!!!!!!')
```

```
question = func('???')
sentence = func('.')
exclaim('OMG')
question('What is this')
sentence('Python is easy')
exclaim('Really')
```

Output-

```
OMG!!!!
What is this???
Python is easy.
Really!!!!
```

The `lambda` expression remembers the value of the symbol from the enclosing scope even after the flow of control is not in that scope; thus, it acts as a closure.

The function `func` returns a function object. That function object is created by the `lambda` expression. `message` is the parameter of the `lambda` expression and `symbol` is the parameter of the function `func`. We have called the function `func` 3 times, first time with argument `'!!!!'`, second time with argument `'???'` and third time with argument `'.'`. These arguments will be assigned to the parameter `symbol` when `func` is called.

When the function `func` is executed, the `lambda` expression creates a function object which is returned by `func`. We have assigned the returned function objects to the names `exclaim`, `question`, and `sentence`. These names now act like functions and we call them with different arguments. The arguments that are sent to these function calls, are assigned to the parameter `message` of `lambda` expression.

We saw examples of how we can use `lambda` expressions to send function objects as arguments to a function and how to return function objects from a function. `lambda` expressions give you the ability to create a function on the fly in situations where you do not need a full-fledged function written using `def`. By using `lambda` expressions, you can inline a function

definition where it has to be used. When you need a short and simple function that will be used only where it is defined, use a `lambda` expression.

19.7 Creating jump tables using `lambda` functions

We can place `lambda` function inside list and dictionary literals. This way we can use `lambda` expressions to create jump tables.

```
>>> L = [lambda s: s.strip().lower(),
...      lambda s: s.strip().upper(),
...      lambda s: s.lstrip().title(),
...      lambda s: s[::-1].lower(),
...      ]
```

Here, we have stored these `lambda` expressions in a list:

```
>>> L[1]('Python')
PYTHON
>>> L[3]('Python')
nohtyp
```

Here, we have used `lambda` expressions as values of a dictionary:

```
>>> d = {'add': lambda x, y: x + y,
...      'subtract': lambda x, y: x - y,
...      'multiply': lambda x, y: x * y,
...      'divide': lambda x, y: x // y,
...      'power': lambda x, y: x ** y,
...      'double': lambda x: x * 2,
...      'square': lambda x: x ** 2,
...      'table': lambda x: [x * i for i in
range(1, 11)]},
```

```

...     'summation': lambda x: sum(range(1, x +
1)),
...     }
>>> d['summation'](4)
10
>>> d['power'](3,2)
9

```

So, when you have to write a lot of small functions that are used only once, you can use `lambda` expressions instead of defining lots of one-off `def` statements.

19.8 Using `lambda` expressions in sorted built-in function

We have seen that the `lambda` expressions are often used as an argument to functions that expect a function object. The `sorted` built-in function has a parameter named `key` that accepts a function object.

```
>>> help(sorted)
```

```

Help on built-in function sorted in module
builtins:

```

```
sorted(iterable, /, *, key=None, reverse=False)
```

```
    Return a new list containing all items from
    the iterable in ascending order.
```

```
    A custom key function can be supplied to
    customize the sort order, and the reverse flag can
    be set to request the result in descending order.
```

In the `sorted` function, the first parameter is the iterable on which the sorting is done. It can be a list, tuple, or dictionary; it sorts in ascending order based on the natural ordering of the items present in the iterable. It does not perform in-place sort, which means that it will not in any way change the iterable; it will only return a list that represents the sorted

version of the iterable. The parameter named `key` accepts a function object, and it has a default argument of `None`.

The argument that we send for this `key` parameter should be a function that takes a single argument and returns a single value. This function will be applied to each element of the iterable to get the sorting keys. When this argument is `None`, the sorting is performed by comparing the items inside the iterable. When this is a function, that function is applied to each item of the iterable, and the sorting is performed on those values instead of the items of the iterable. Let us understand this with a few examples:

```
>>> L = ['elephant', 'bear', 'duck', 'fox', 'giraffe']
>>> sorted(L)
['bear', 'duck', 'elephant', 'fox', 'giraffe']
```

When we sorted this list of strings, the sorting is performed in lexicographical order. Now suppose in our list, there are some strings that have letters in upper case.

```
>>> L = ['elephant', 'bear', 'Duck', 'Fox', 'Giraffe']
>>> sorted(L)
['Duck', 'Fox', 'Giraffe', 'bear', 'elephant']
```

After sorting, the string `'bear'` comes after the 3 strings because upper case and lower case have different codes. We can see the codes using `ord` function.

```
>>> ord('D')
68
>>> ord('b')
98
```

Upper case letters have lower codes, so we got the strings `'Duck'`, `'Fox'`, `'Giraffe'` before `'bear'`.

Now suppose we want to sort the strings based on their length instead of the default lexicographical order. For this we can send the built-in `len` function for the `key` parameter.

```
>>> L = ['elephant', 'bear', 'Duck', 'Fox',  
'Giraffe']  
>>> sorted(L, key=len)  
['Fox', 'bear', 'Duck', 'Giraffe', 'elephant']
```

The `len` function is applied to each string of the list, and a key for each element will be obtained. The keys for the strings in the list would be – 8 for 'elephant', 4 for 'bear', 4 for 'Duck', 3 for 'Fox' and 7 for 'Giraffe'. So now, the length of the string is used as a key for sorting. The strings 'bear' and 'duck' have the same length, but 'bear' comes before 'duck' because the sorting performed by the `sorted` function is stable, which means that if two elements have the same level, then whichever is first, will appear first in the result.

As we have seen, the sorting performed by the `sorted` function is case-sensitive, so the upper-case letters came before the lower ones.

```
>>> L = ['elephant', 'bear', 'Duck', 'Fox',  
'Giraffe']  
>>> sorted(L)  
['Duck', 'Fox', 'Giraffe', 'bear', 'elephant']
```

Now, suppose we want to perform a case-insensitive sort i.e. we want to ignore the case of the strings while sorting. We can do this by sending the `str.lower` method as the argument for the `key` parameter.

```
>>> sorted(L, key=str.lower)  
['bear', 'Duck', 'elephant', 'Fox', 'Giraffe']
```

The `str.lower` method is applied to each string. It returns the equivalent lowercase string for each string and the lowercase forms of all the strings will be used as the keys for sorting. So, the keys in this case would be 'elephant', 'bear', 'duck', 'fox', 'giraffe' and sorting will be done on these keys.

If we send the method `str.upper` for the `key` parameter, then also we will get case insensitive sort because, in this case, the uppercase forms of all the strings will be used as the keys.

```
>>> sorted(L, key=str.upper)
```

```
['bear', 'Duck', 'elephant', 'Fox', 'Giraffe']
```

In this case, the keys are 'ELEPHANT', 'BEAR', 'DUCK', 'FOX', and 'GIRAFFE', and sorting will be done on these keys instead of the original strings of the list.

Suppose we have to sort based on the reversed spelling of the string and case insensitively. We don't have any built-in function or method for that. We can define our own function and send it.

```
>>> def rev(s):  
    return s.lower()[::-1]
```

```
>>> sorted(L, key=rev)
```

```
['Giraffe', 'Duck', 'bear', 'elephant', 'Fox']
```

This function that we have written has to be used nowhere else, we have to write it because we had to send the function object to the `sorted` function. The function returns just a single expression so instead of creating a new function by using a `def` statement, we can send a `lambda` expression.

```
>>> sorted(L, key=lambda s: s.lower()[::-1])  
['Giraffe', 'Duck', 'bear', 'elephant', 'Fox']
```

Let us see one more example:

```
>>> employees = [ ('Rajendra', 'Kumar', 32, 6000),  
...               ('Sam', 'Saxena', 43, 8000),  
...               ('Shyamchandra', 'Verma', 23, 3000),  
...               ('Sam', 'Gupta', 33, 7000),  
...               ('Sam', 'Sung', 31, 5000)  
...               ]
```

Here, we have a list that contains tuples; each tuple contains the first name, last name, age, and salary of the employee. If we call the `sorted` function without any key, then regular sort order is first name, last name, age, and salary.

```
>>> sorted(employees)
```

```
[('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000), ('Sam', 'Sung', 31, 5000), ('Shyamchandra', 'Verma', 23, 3000)]
```

Sorting is done on first name; if first names are the same, then those tuples are sorted based on last name; if last name is also the same, then those tuples will be sorted using age, and if age is also the same, then those tuples will be sorted on salary. If all 4 elements are the same for some tuples, then the tuple that appears earlier in the list appears earlier in the result also. If we want to change this sort order, then we can send the `key` argument.

Suppose we want to sort the tuples based on the age of the employee. Age is present at index 2 of the tuples, so we can send a `lambda` function that returns the element at index 2 of a sequence.

```
>>> sorted(employees, key=lambda x: x[2])
[('Shyamchandra', 'Verma', 23, 3000), ('Sam', 'Sung', 31, 5000), ('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000)]
```

For each tuple of the list, this `lambda` function returns the element at index 2 which is used as the key for sorting. Similarly, if we want to sort on the salary then we can make the `lambda` function return element at index 3.

Now suppose we want to sort on the combined length of first name and last name. We will need a `lambda` function that gives us the sum of lengths of first name and last name.

```
>>> sorted(employees, key=lambda x: len(x[0]) + len(x[1]))
[('Sam', 'Sung', 31, 5000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000), ('Rajendra', 'Kumar', 32, 6000), ('Shyamchandra', 'Verma', 23, 3000)]
```

The following call to the `sorted` function shows the regular sort order without any key.

```
>>> sorted(employees)
```

```
[('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000), ('Sam', 'Sung', 31, 5000), ('Shyamchandra', 'Verma', 23, 3000)]
```

Here, if the first names are same then the sorting is done on the second name. Suppose we want the sorting to be done on the first name and then age, then we can do this:

```
>>> sorted(employees, key=lambda t: (t[0], t[2]))  
[('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Sung', 31, 5000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000), ('Shyamchandra', 'Verma', 23, 3000)]
```

If you want case insensitive sort here, then you can write this:

```
>>> sorted(employees, key=lambda t: (t[0].upper(), t[2]))  
[('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Sung', 31, 5000), ('Sam', 'Gupta', 33, 7000), ('Sam', 'Saxena', 43, 8000), ('Shyamchandra', 'Verma', 23, 3000)]
```

So, by sending different lambda functions, we can change the sort order in any way. Writing lots of named tiny functions just to change the sort order is inconvenient, so lambda functions are generally used for the key parameter.

The key parameter is also present in the sort method of list type and in max and min built-in functions. Let us sort our employees list using the sort method, and for the key parameter, we will send a lambda function that returns the element at index 1, so sorting will be performed on last names.

```
>>> employees.sort(key=lambda x: x[1])  
>>> employees  
[('Sam', 'Gupta', 33, 7000), ('Rajendra', 'Kumar', 32, 6000), ('Sam', 'Saxena', 43, 8000), ('Sam',
```

```
'Sung', 31, 5000), ('Shyamchandra', 'Verma', 23, 3000)]
```

This makes in-place changes and our list is changed. Let us bring our original list back:

```
>>> employees = [ ('Rajendra', 'Kumar', 32, 6000),  
...               ('Sam', 'Saxena', 43, 8000),  
...               ('Shyamchandra', 'Verma', 23,  
3000),  
...               ('Sam', 'Gupta', 33, 7000),  
...               ('Sam', 'Sung', 31, 5000)  
...           ]
```

To find out who gets the maximum salary, we can use the max function with appropriate lambda function for the key parameter.

```
>>> max(employees, key=lambda x: x[3])  
('Sam', 'Saxena', 43, 8000)
```

If we need to find out who is the youngest employee, we have to use age as the key.

```
>>> min(employees, key=lambda x: x[2])  
('Shyamchandra', 'Verma', 23, 3000)
```

The sorted function works on any iterable, and therefore will work on dictionaries also. Suppose we have the following dictionary and we sort it by using the sorted built-in function:

```
>>> d = {'pen': 23, 'eraser': 5, 'book': 30}  
>>> sorted(d)  
['book', 'eraser', 'pen']
```

We get a list that is sorted on keys of the dictionary. We will get the same result if we send d.keys() as the argument.

```
>>> sorted(d.keys())  
['book', 'eraser', 'pen']
```


To get a sorted list of values, we can write this:

```
>>> sorted(d.values())  
[5, 23, 30]
```

If we use the `items` method, we get a list of tuples that is sorted on keys of the dictionary.

```
>>> sorted(d.items())  
[('book', 30), ('eraser', 5), ('pen', 23)]
```

If we want a list of tuples that is sorted on values of the dictionary, we can send a lambda function.

```
>>> sorted(d.items(), key=lambda t: t[1])  
[('eraser', 5), ('pen', 23), ('book', 30)]
```

If a built-in class or a user-defined class does not have methods that support the sorting then the `sort` method or `sorted` function will not work for iterables containing the objects of that class. We can, however, use lambda expressions to sort them. For example, suppose we have a class named `Student` and we have a list that contains some objects of `Student` class.

```
class Student:  
    def __init__(self, name, marks, birthYear):  
        self.name = name  
        self.marks = marks  
        self.birthYear = birthYear  
    def __str__(self):  
        return f'{self.name} {self.marks}  
{self.birthYear}'  
s1 = Student('John', 97, 1988)  
s2 = Student('Sam', 89, 1987)  
s3 = Student('Pam', 99, 1982)  
L = [s1, s2, s3]  
L.sort() # Will give TypeError: '<' not supported  
between instances of 'Student'
```

If we try to sort a list of `Student` objects using the `sort` method, we get an error. To make the objects of this class sortable, we need to define the magic method for less than operator for this class, but we may not have access to the code of the class if it is a built-in class or is written by someone else. We can make this `sort` method work by sending a `lambda` expression for the `key` parameter.

```
L.sort(key=lambda s: s.marks)
```

Here, we are specifying that the sorting should be based on the `marks` as the key, and now the `sort` method works correctly.

```
for i in L:
    print(i)
```

Output-

```
Sam 89 1987
John 97 1988
Pam 99 1982
```

So, this way we can sort objects that do not have an ordering defined.

Now, suppose we have an `Employee` class, which has defined a `__lt__` method. The presence of this method enables us to sort the objects of this class.

```
class Employee:
    def __init__(self, name, phone, basic, ta,
da):
        self.name = name
        self.phone = phone
        self.basic = basic
        self.ta = ta
        self.da = da
    def __lt__(self, other):
        return self.name < other.name
    def __str__(self):
```

```

        return f'{self.name} {self.phone}
{self.basic} {self.ta} {self.da}'
    def __repr__(self):
        return f'{self.name} {self.phone}
{self.basic} {self.ta} {self.da}'
e1 = Employee('Zeba', 89889444, 3000, 500, 200)
e2 = Employee('Amit', 99883994, 4000, 300, 500)
e3 = Employee('Neema', 83988399, 3000, 1000, 500)
e4 = Employee('Rini', 99878784, 3500, 0, 500)
L = [e1, e2, e3, e4]
print(L)
L.sort()
print(L)

```

Output-

```

[Zeba 89889444 3000 500 200, Amit 99883994 4000
300 500, Neema 83988399 3000 1000 500, Rini
99878784 3500 0 500]
[Amit 99883994 4000 300 500, Neema 83988399 3000
1000 500, Rini 99878784 3500 0 500, Zeba 89889444
3000 500 200]

```

When we sort a list containing objects of this class, the objects will be sorted by names, but if we want to sort in some other way, then we can change the key while sorting. For that, we have to send a lambda expression for the key parameter. Suppose we want to sort based on the sum of basic, ta and da. We can send this lambda expression:

```
L.sort(key=lambda e: e.basic + e.ta + e.da)
```

In this section, we saw the use of lambda expressions in sorting. It is a very common use case for lambda expressions since in these cases, functions are very simple ones that just return an expression. So instead of writing separate def statements, we just inline the function using lambda expressions. In the coming sections, we will see how the lambda functions

are used in the built-in higher-order functions `map` and `filter` and in the function `reduce`.

19.9 Functional programming

Functional programming is a programming paradigm in which most of the work in a program is done using pure functions. A pure function is a function without any side effects; its return value is always determined by its input arguments, so it always returns the same output, given the same input. In Python, functional programming is supported by the higher-order functions `map`, `filter`, and `reduce`.

```
map(func, iterable)
filter(func, iterable)
reduce(func, iterable)
```

`map` and `filter` are built-in functions while `reduce` is present in `functools` module. These functions can replace for loops and if statements and hence can make the code shorter. So, using these functions makes the code concise as the control flow statements can be replaced by expressions, and the programmer can focus on solving the actual problem instead of going into the details of looping and branching.

These functions are not used much because now Python has better alternatives in the form of comprehensions and generators which we have already seen before. If you are writing your code and you can do the same thing with a comprehension, then it is better to use a comprehension since they are considered more Pythonic and are preferred by the Python community. Although these functions are not used very often, you might encounter them in some code that you are using so it is good to be familiar with them. All these functions have a parameter that accepts a function as an argument. Generally, the functions that are to be sent as arguments to these functions are single-expression throw-away functions. So, we mostly use `lambda` expressions as arguments to these functions instead of defining separate `def` statements. In the next three sections, we will discuss these three functions.

19.10 map

```
map(func, iterable)
```

The `map` function takes in a function and an iterable as argument. The function that is sent, should be such that it takes in a single argument and returns a single value. The second argument can be any iterable like list or a tuple; the values inside the iterable should be acceptable as arguments to the first function.

This `map` function returns a map object which is an iterator in which each item is obtained by applying the argument function to each element of the iterable. An iterator is returned instead of a list for efficiency reasons.

If the input iterable contains values `v1`, `v2`, `v3`, `v4` and `v5`, then the `map` function will return an iterator that produces values `func(v1)`, `func(v2)`, `func(v3)`, `func(v4)`, `func(v5)`. We can iterate over the iterator as in a loop, or we can convert the iterator to iterables like list, tuple, etc. Let us understand this with the help of an example.

Suppose we have a list and a function named `square` that takes a single argument and returns the square of that argument.

```
>>> numbers = [4, 6, 7, 9]
>>> def square(n):
...     return n * n
```

We call the `map` function with the function `square` and the list `numbers` as the two arguments.

```
>>> map(square, numbers)
<map object at 0x000002677E9FA170>
```

This will return an iterator that produces the values `square(4)`, `square(6)`, `square(7)`, `square(9)`. So, the returned iterator will produce values 16, 36, 49 and 81. If we put the call to `map` inside the `list` function then we will get a list.

```
>>> list(map(square, numbers))
[16, 36, 49, 81]
```

The function `square` is a short function so you can use a `lambda` function instead, if you do not need the `square` function anywhere else.

```
>>> list(map(lambda n: n*n, numbers))  
[16, 36, 49, 81]
```

If we send the `str` function, then we get an iterator that produces the values `str(4)`, `str(6)`, `str(7)`, `str(9)`.

```
>>> list(map(str, numbers))  
['4', '6', '7', '9']
```

Let us see some more examples:

```
>>> L = [4, 6, 8, 9, 3]
```

We have this list and we want to create another list by multiplying each element of this list by 10. So, the new list should have elements 40, 60, 80, 90 and 30. We can do this by iterating over this list in a `for` loop and then appending items to the new list, but in functional programming we use functions to avoid loops. So, we will call the `map` function, with a `lambda` function as the first argument and list `L` as the second argument.

```
>>> map(lambda x: x*10, L)  
<map object at 0x000001CCFA83BDC0>
```

This returned iterator can be converted to a list.

```
>>> list(map(lambda x: x*10, L))  
[40, 60, 80, 90, 30]
```

The `lambda` function was called for each element of the list and the values that were returned by the `lambda` function are the values produced by the iterator.

In our next example we have used the `map` function to create a list that contains cubes of all the numbers from 5 to 10.

```
>>> list(map(lambda x: x**3, range(5,11)))  
[125, 216, 343, 512, 729, 1000]
```

The `lambda` function returns the cube of its parameter, and the `range` function will give us numbers from 5 to 10.

In our next example, we have a dictionary named `students` that we have seen in earlier chapters. The student IDs are used as keys of the dictionary and the corresponding values are dictionaries that contain the student's details.

```
students = { 105416: { 'name': 'John',
                       'gender': 'M',
                       'city': 'Paris',
                       'age': 21,
                       'marks': { 'Maths': 89,
                                   'Physics':
78,
                                   'Chemistry': 91 },
                       'is_sporty': True },
             144547: { 'name': 'Dev',
                       'gender': 'M',
                       'city': 'London',
                       'age': 23,
                       'marks': { 'Maths': 88,
                                   'Physics':
77,
                                   'Chemistry': 98 },
                       'is_sporty': False },
             132399: { 'name': 'Mary',
                       'gender': 'F',
                       'city': 'Paris',
                       'age': 22,
                       'marks': { 'Maths': 99,
```

```

87,
'Physics':
'Chemistry':88 },
'is_sporty': True }
}

```

We need to create a list of tuples where each tuple contains the name and age of the student. So, we need the following output:

```
[('John', 21), ('Dev', 23), ('Mary', 22)]
```

Let us write a map function call for it. For the second argument, we will send `students.values()`, so it will give us the student dictionaries. For the first argument, we will write a lambda function. For each dictionary, we want the name and age, so if the parameter is `d`, then it will return `(d['name'], d['age'])`. Since we want to return a tuple, we need to explicitly enclose it in parentheses.

```

>>> list(map(lambda d: (d['name'], d['age']),
students.values()))
[('John', 21), ('Dev', 23), ('Mary', 22)]

```

19.11 map with multiple iterables

```
map(func, iterable1, iterable2, ..... )
```

It is possible to send more than one iterable in the map function, but the condition is that the number of arguments in the function `func` should be equal to the number of iterables that are sent to the map function. For example, if we send three iterables, then the function `func` should be such that it takes 3 arguments and returns a single value. The arguments to the function are received from the corresponding iterables. Here is an example-

```

>>> def multiply(x, y, z):
...     return x * y * z
...

```



```
>>> map(multiply, [1,2,3,4], (4,5,6,7),  
[10,20,30,40])
```

```
<map object at 0x000002A61232AE30>
```

The function `multiply` takes three arguments and returns their product. We are calling the `map` function with this function as the first argument, and then we are sending three iterables, one is a tuple, and two are lists. `map` will call the `multiply` function for the corresponding elements of the iterables one by one.

Initially, it applies the function to the first elements of the iterables, so it calls `multiply` function with arguments 1,4 and 10. The return value becomes the first value given out by the iterator. Then it applies the function to the second element of the iterables, so it calls `multiply` with arguments 2, 5, and 20, and the return value becomes the second value given out by the iterator. Similarly, `multiply` is called for other elements of the iterables, and the return values become the values produced by the iterator.

So we get an iterator that produces values `multiply(1,4,10)`, `multiply(2,5,20)`, `multiply(3,6,30)`, `multiply(4,7,40)`. We can see that in each call of `multiply`, the value for parameter `x` comes from the first list, the value for parameter `y` comes from the tuple, and the value for parameter `z` comes from the second list. That is why the number of iterables sent should be equal to the number of arguments that the function takes.

So, the call of `map` will return an iterator that will produce values 40, 200, 540 and 1120. If we enclose the call in the `list` function, then we get our result in a list.

```
>>> list(map(multiply, [1,2,3,4], (4,5,6,7),  
[10,20,30,40]))  
[40, 200, 540, 1120]
```

The iterables that are sent here need not be of the same length. If they have different lengths then the `map` function will stop applying the function when the shortest iterable is exhausted.

In our next example, we have 3 lists of unequal length, and we want to create a new list in which each element is the sum of the corresponding elements of these lists.

```
>>> list1 = [1, 9, 3, 4]
>>> list2 = [4, 3, 6, 1, 7, 8]
>>> list3 = [8, 9, 3, 5, 6]
>>> list(map(lambda x, y, z: x + y + z, list1,
list2, list3))
[13, 21, 12, 10]
```

The three lists were of unequal length, so the map function stopped when the shortest list that is of length 3, got exhausted.

In our next example, we have three lists, and we want to create a new list named `ids`.

```
>>> names = ['Sophia', 'Michael', 'Benedict',
'Anthony']
>>> cities = ['Paris', 'London', 'Bareilly',
'Tokyo']
>>> phones = [676858939, 223878965, 856937891,
676757913]
```

In the `ids` list, each element should be created by joining the last 2 characters from the name, the first 2 characters from the city and the first 3 characters from the phone. We will call the map function, in which we will send the three lists and before that we send a lambda function.

```
>>> ids = list(map(lambda x,y,z: x[-2:] + y[:2] +
str(z)[:3], names, cities, phones))
>>> ids
['iaPa676', 'elLo223', 'ctBa856', 'nyTo676']
```

In the lambda function, we have three parameters. From the name, we want the last two characters, so we have written `x[-2:]`, then from city we want first two characters so we have written `y[:2]` and from phone we want first 3 characters so have written `str(z)[:3]`.

19.12 filter

The built-in function `filter` is used to filter out elements of an iterable depending on the result of another function.

```
filter(func, iterable)
```

This function takes two arguments, a function and an iterable. The argument function `func` should be such that it accepts a single argument and returns a Boolean value, which means that it should return either `True` or `False`. The second argument can be any iterable like list, set or tuple.

The `filter` function returns a filter object which is an iterator that produces only those items of iterable for which the function `func` returns `True`. The argument function is applied to each element of the iterable; if the function returns `True` for an element then that element will be produced by the iterator. So, we can say that it filters out all the elements for which the function returns `False`. Let us understand this with the help of an example.

We have the following list of numbers; from this list, we want only those elements that are divisible by 3; this means that we want to filter out the rest of the elements.

```
>>> numbers = [24, 34, 0, 4, 12, 45, 67, 15]
```

We will define a function that takes a number as the argument and returns `True` if the number is divisible by 3 and returns `False` otherwise.

```
>>> def func(x):  
...     return x % 3 == 0
```

Next, we will call the `filter` function with the function `func` and the `numbers` list as the arguments.

```
>>> filter(func, numbers)  
<filter object at 0x000002A61232B9D0>
```

The function `func` is applied to each element of the list and only those numbers are produced by the iterator for which this function returns `True`. The function returns `True` for 24, 0, 12, 45 and 15 so these numbers will be produced by the iterator that is returned by this call of `filter`. We can convert it to a list to see the results.

```
>>> list(filter(func, numbers))  
[24, 0, 12, 45, 15]
```

So, the `filter` function includes only those elements for which the argument function returns `True` and rejects those elements for which the function returns `False`. We can send a `lambda` expression as an argument instead of a named function and we will get the same result.

```
>>> list(filter(lambda x: x % 3 == 0, numbers))  
[24, 0, 12, 45, 15]
```

The first argument to the `filter` function can be a function or it can be `None`. If it is `None`, then it returns the iterator that will produce those elements that are `truthy`.

```
>>> list(filter(None, numbers))  
[24, 34, 4, 12, 45, 67, 15]
```

Here we have sent `None`, so we get all non-zero integers which are considered `truthy`, and `0` which is considered `falsy` is filtered out.

In our next example, we have a list of numbers and from this list we want only those numbers that are less than 100.

```
>>> numbers = [12, 109, 67, 34, 390, 65, 990, 87, 52]  
>>> list(filter(lambda x: x < 100, numbers))  
[12, 67, 34, 65, 87, 52]
```

We sent a `lambda` function that returns `True` for numbers that are less than 100, and we sent the `numbers` list as the second argument. On enclosing the returned iterator inside the `list` function, we get the list of numbers less than 100.

In the next example, we have a list of tuples and we want only those tuples for which the second element is less than 100.

```
>>> L = [('A', 101), ('X', 89), ('C', 209), ('F', 39)]  
>>> list(filter(lambda t: t[1] < 100, L))  
[('X', 89), ('F', 39)]
```

Here the `lambda` function accepts a tuple and checks its index 1 element and if it is less than 100, then it returns `True`. The two tuples `('A', 101)` and `('C', 209)` were not included since the values at second index are more than 100.

In the next example, we have a list of strings and we want only those strings that end with `.doc`.

```
>>> L = ['names.doc', 'file2.xls', '', 'info.doc',  
'help.doc', '', 'show.ppt']  
>>> list(filter(lambda s: s.endswith('.doc'), L))  
['names.doc', 'info.doc', 'help.doc']
```

We called the `endswith` method on the string `s`, so only those strings were included that end with `.doc`.

If we want to filter out empty strings from the above list `L`, we can use `None` as the first argument. If `None` is used as the first argument, then only truthy values are included and since empty strings are falsy, they are rejected.

```
>>> list(filter(None, L))  
['names.doc', 'file2.xls', 'info.doc', 'help.doc',  
'show.ppt']
```

Now, here instead of the `lambda` function we sent `None`, and we get all the non-empty strings.

In the next example, we have a list of strings, and we want only those strings that are alphanumeric. We can use the `isalnum` method of string class, it returns `True` if the string is alphanumeric otherwise it returns `False`.

```
>>> L = ['ab12', '23', '22%cd', 'cv', 'xy@z']  
>>> list(filter(str.isalnum, L))  
['ab12', '23', 'cv']
```

In our next example, we have a class named `Student` and a list of objects of `Student` type.

```
class Student:  
    def __init__(self, name, marks, birthYear):
```

```

        self.name = name
        self.marks = marks
        self.birthYear = birthYear
    def __str__(self):
        return f'{self.name} {self.marks}
{self.birthYear}'
    def __repr__(self):
        return f'{self.name} {self.marks}
{self.birthYear}'
s1 = Student('John', 97, 1988)
s2 = Student('Sam', 89, 1987)
s3 = Student('Pam', 99, 1985)
s4 = Student('Tim', 91, 1983)
s5 = Student('Jim', 80, 1987)
L = [s1, s2, s3, s4, s5]

```

We want to create a new list of students who have got more than 90 marks. We can use the `filter` function for getting this new list.

```

>>> new_list = list(filter(lambda x: x.marks > 90,
L))
>>> print(new_list)
[John 97 1988, Pam 99 1985, Tim 91 1983]

```

The `lambda` function used here will return `True` for only those `Student` objects whose marks are more than 90.

In the next example, we will combine `map` and `filter` to get our required list. We have the following list `L` and we want to get the cubes of all positive numbers in this list.

```

>>> L = [-3, 7, -9, 6, 3, 5, -8]

```

For filtering out the negative numbers we can use `filter` and for getting the cubes we can use `map`.

```

>>> map(lambda x: x*x*x, filter(lambda x: x>0, L))

```

For the first argument to the `map` function, we have written a `lambda` function that returns the cube of a number. For the second argument, we have written a call to `filter` function that includes only negative numbers. We can enclose the whole thing inside the `list` function to get a list.

```
>>> list(map(lambda x: x*x*x, filter(lambda x:
x>0, L)))
[343, 216, 27, 125]
```

Here is another example where we have used both `map` and `filter` functions. From the following list of strings, we want to get all the strings that are alphabetic and change them to uppercase.

```
>>> L1 = ['??', 'abc', 'Efg', '123', 'A1', 'pqr']
>>> list(map(str.upper, filter(lambda s:
s.isalpha(), L1)))
['ABC', 'EFG', 'PQR']
```

All the `map` and `filter` operations can be replaced by generator expressions or list comprehensions. For example, for getting the previous 2 lists, we can write these list comprehensions.

```
>>> [x * x * x for x in L if x > 0]
[343, 216, 27, 125]
>>> [s.upper() for s in L1 if s.isalpha()]
['ABC', 'EFG', 'PQR']
```

So, we can always replace `map` and `filter` expressions with list comprehensions or generator expressions. List comprehensions and generator expressions are better alternatives, and so their use is recommended wherever possible. They provide a more readable solution as can be seen in these two cases. Therefore, since the introduction of list comprehensions and generator expressions, the functions `map` and `filter` have lost their importance and have very limited use. But, if sometimes the filtering criteria is complex, it is better to use a separate function for the filtering process and then use it in the `filter` function.

19.13 Reducing an iterable

The function `reduce` reduces an iterable to a single value, which means that it returns a single value for the whole iterable. Here are some examples:

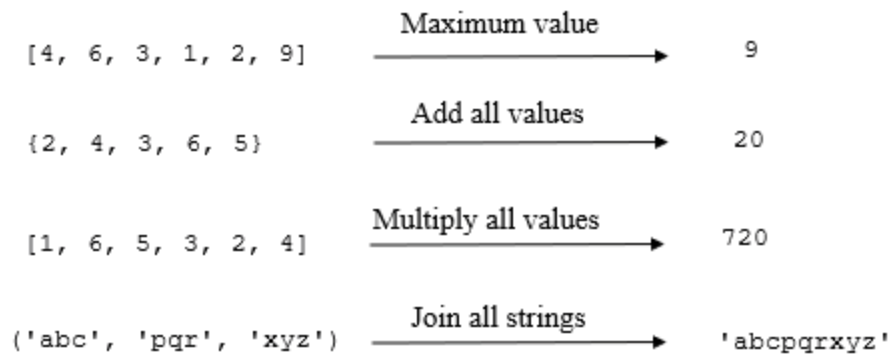


Figure 19.3: Reducing iterables

In the first example we have reduced the list to a single value such that the single value is the maximum value. Then we have reduced a set to a single value which is the sum of all values in the set. In the third example, the list is reduced to the product of all the values in the list. In the last example, a tuple of strings is reduced to a single string by joining all the strings inside the tuple.

Now, let us see how the `reduce` function works and reduces an iterable to a single value. This function is not a built-in function, we need to import it from the `functools` module.

```
reduce(func, iterable)
```

This function takes two arguments, first is a function and second is an iterable. The argument function should be such that it takes two arguments and returns a single value. The function `reduce` works by continually calling the argument function for the successive elements of the iterable, computing and accumulating the results, till the iterable is reduced to a single value.

The argument function is invoked with the first and second values of the iterable, followed by computation of the result. Subsequently, the function is invoked with this result and the third value, with the process repeating for

the fourth value and beyond. This continues till all the values in the iterable are used.

Suppose we have a list and we define a function `add` that takes two arguments and returns their sum. We call the function `reduce` and send the function `add` as the first argument and the list as the second argument.

```
>>> L = [4, 6, 3, 1, 2]
>>> def add(x, y):
...     return x + y
>>> from functools import reduce
>>> reduce(add, L)
16
```

Here is how this `reduce` function works:

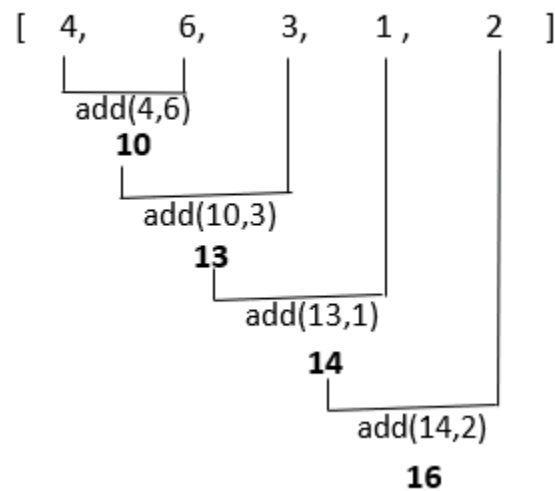


Figure 19.4: Working of *reduce* function

First, the function `add` is called for the first and second element of the list, then the function `add` is called with the result of the call `add(4, 6)` and the next element of the list. After that `add` is called with result of the call `add(10, 3)` and the next element. This process continues till there are no more elements left in the list so at last a single value is returned from `reduce`. The call to `reduce`, that we have written, is equivalent to:

```
add(add(add(add(4, 6), 3), 1), 2)
```

So, this is how the function `reduce` reduces a sequence of values to a single value. The argument function is applied to successive items in the iterable, until the iterable is exhausted.

We can also give an initial value in the `reduce` function.

```
reduce(func, iterable, initialvalue)
```

If we provide this value, then `reduce` does not start working from the first value of the iterable, it starts from the initial value. If we take the previous example and send an initial value, then this is how the `reduce` function will work:

```
>>> reduce(add, L, 1000)
1016
```

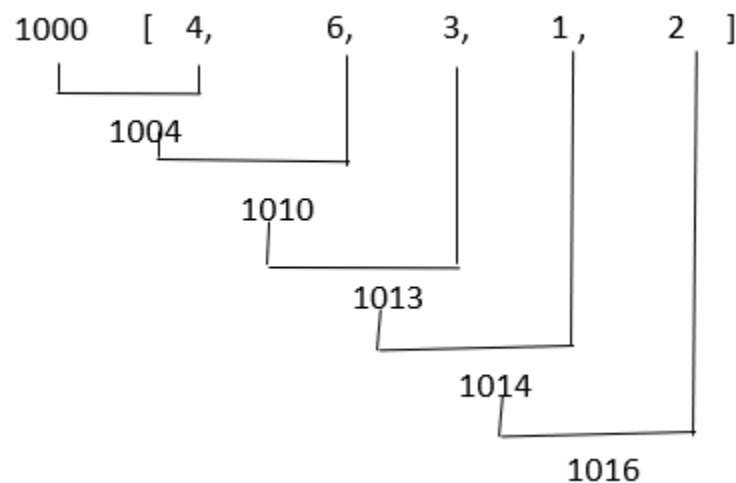


Figure 19.5: Working of `reduce` function with an initial value

First the function is applied to initial value and first element and then to the result and second element and so on. The initial value is generally used to handle the case when the iterable is empty.

Now, suppose we have the following list of strings:

```
>>> words = ['apple', 'boy', 'cat']
>>> reduce(add, words)
'appleboycat'
```

We sent the `add` function and so in the result all the strings are joined together.

If we want to find the maximum value of all the values in a list, we can send a `lambda` function.

```
>>> L = [4, 6, 3, 1, 2]
>>> reduce(lambda x, y: x if x > y else y, L)
6
```

The list is reduced to a single value which is the maximum value. To get the minimum value, we can change the greater than sign to less than sign.

When an iterable has only one element, then `reduce` returns that element without applying the function. When the iterable is empty, it will give an error. To guard against such cases, you can pass an initial value, this will guarantee that `reduce` returns a value even when the iterable is empty. For example, suppose our list `L` becomes empty.

```
>>> L.clear()
```

Now if we call `reduce`, we will get an error.

```
>>> reduce(add, L)
```

```
TypeError: reduce() of empty iterable with no
initial value
```

To avoid this error, we can use an initial value; here we can use 0 as the initial value.

```
>>> reduce(add, L, 0)
0
```

Now we do not get any error for our empty list and the result will not be affected if the list is not empty. So, the initial value acts as the default result when the iterable is empty.

19.14 Built-in reducing functions

Most of the common use cases of the `reduce` function like adding elements of an iterable or finding maximum and minimum values in an

iterable are provided as built in functions in Python (`sum()`, `max()` and `min()`). So, you do not need to write them using `reduce`. We have used these built-in functions before. There are more reducing functions that return a single value for a sequence of values.

`all()` Returns True if all elements of the iterable are truthy or if the iterable is empty

`any()` Returns True if any item is Truthy, if iterable is empty it returns False

These two are also built-in functions, both take an iterable as argument.

```
>>> all([1, 2, 3, 0, 5, 8])
```

```
False
```

We sent a list to the function and got False because all the values in the list are not Truthy, there is a 0 which is Falsy. If we remove this zero, we get True because now all the elements are Truthy.

```
>>> all([1, 2, 3, 5, 8])
```

```
True
```

```
>>> names = ['Raj', 'Dev', '', 'Sam']
```

```
>>> all(names)
```

```
False
```

We get False because there is an empty string in the list. If we remove it, we will get True.

We can use this function in an `if` statement like, `if all(names)` then perform an action.

```
if all(names):
```

```
    pass
```

We have the following list named `marks` and suppose we want to perform some action, only if all the numbers in this list are greater than 50.

```
>>> marks = [65, 67, 89, 48, 90, 56]
```

If we write the following generator expression, it will give us values, True, True, True, False, True, True.

```
>>> (m > 50 for m in marks)
```

We can send this generator expression as argument to `all`, to test whether all values are greater than 50.

```
>>> all(m > 50 for m in marks)
```

```
False
```

The generator does not give all Truthy values, so we get `False`. If we delete the element 48 from the list and call the `all` function again, then we will get `True` because the generator expression will give all `True` values.

```
>>> marks.remove(48)
```

```
>>> all(m > 50 for m in marks)
```

```
True
```

Now suppose we want to know whether any number in the `marks` list is greater than 100. For this, we can send a generator expression to the built-in `any` function.

```
>>> any(m > 100 for m in marks)
```

```
False
```

The generator expression does not give out any `True` value, so we get `False` here. If we insert a value more than 100 in this list, and then call the function again, then we will get `True`.

```
>>> marks.insert(2, 150)
```

```
>>> any(m > 100 for m in marks)
```

```
True
```

Now, suppose we have this list of dictionaries. Each dictionary has three keys- `name`, `city`, and `marks`.

```
>>> students = [ {'name': 'John',  
...               'city': 'Paris',  
...               'marks': 21,  
...               },  
...               {'name': 'Dev',  
...               'city': 'London',
```

```

...         'marks': 23,
...     },
...     {'name': 'Mary',
...      'city': 'Paris',
...      'marks': 22,
...     }
... ]

```

We want to find out the maximum marks obtained by a student, for that we can use the `max` function.

```

>>> max(student['marks'] for student in students)
23

```

This gives us the maximum marks, if we want the whole record of the student who got the maximum marks, then we can send a value for the `key` parameter of the function `max`.

```

>>> max(students, key=lambda d: d['marks'])
{'name': 'Dev', 'city': 'London', 'marks': 23}

```

If we want only the name of the student, we can write this:

```

>>> topper = max(students, key=lambda d:
d['marks'])
>>> topper['name']
Dev

```

The following expression will also give us the same result:

```

>>> max(students, key=lambda d: d['marks'])
['name']
Dev

```

Now suppose we want to find out if there is any student from Paris. We can use the `any` function for this.

```

>>> any(student['city'] == 'Paris' for student in
students)

```

True

19.15 operator module

Some common lambda expressions are used quite often when using higher order functions. These lambda expressions use some common operators like addition, subtraction or multiplication. There is a module named `operator` that gives us the functional equivalents of the common operators.

```
>>> import operator
>>> help(operator)
```

You can call `help` on this module to view all the functions available. Instead of writing your own lambda expressions for common operations like addition or multiplication, you can use the named functions from this module. For example, you can use `operator.add` instead of `lambda x, y: x + y`. These functions come in handy when using higher-order functions like `sorted`, `map`, or `filter`. The functions in this module are highly optimized, so your code's efficiency will be better if you use functions from the `operator` module instead of lambda expressions or `def` functions, wherever possible. And the code becomes more readable also. Let us see an example:

```
>>> list(map(lambda x, y: x * y, [1,2,3,4],
[5,6,7,8]))
[5, 12, 21, 32]
```

Here we have called the `map` function and sent a lambda expression as first argument. Instead of the lambda function, you can send `operator.mul`.

```
>>> list(map(operator.mul, [1,2,3,4], [5,6,7,8]))
[5, 12, 21, 32]
```

The `operator` module has two more useful functions that we can use instead of lambda functions while doing functional programming. These are `itemgetter` and `attrgetter` functions. The `itemgetter` function can be used to get items from sequences and `attrgetter` can be

used to extract attributes from objects. Let us see how to use these functions:

```
>>> from operator import itemgetter
>>> employees = [ ('Rajendra', 'Kumar', 32, 6000),
...               ('Sam', 'Saxena', 43,
8000),
...               ('Shyamchandra', 'Verma', 23,
3000),
...               ('Sam', 'Gupta', 33, 7000),
...               ('Sam', 'Sung', 31, 5000)
...             ]
```

We imported the `itemgetter` function from the `operator` module. We have a list of tuples, which we used earlier when learning about `sorted` function. To sort this list of tuples based on element at the second index, we would write this:

```
>>> sorted(employees, key=lambda t: t[2])
[('Shyamchandra', 'Verma', 23, 3000), ('Sam',
'Sung', 31, 5000), ('Rajendra', 'Kumar', 32,
6000), ('Sam', 'Gupta', 33, 7000), ('Sam',
'Saxena', 43, 8000)]
```

Instead of the `lambda` function, we can use the `itemgetter` function which is more readable and faster and gives us the same result.

```
>>> sorted(employees, key=itemgetter(2))
[('Shyamchandra', 'Verma', 23, 3000), ('Sam',
'Sung', 31, 5000), ('Rajendra', 'Kumar', 32,
6000), ('Sam', 'Gupta', 33, 7000), ('Sam',
'Saxena', 43, 8000)]
```

We can do multiple levels of sorting by sending more than one index values to the `itemgetter` function.

```
>>> sorted(employees, key=itemgetter (1,2))
[('Sam', 'Gupta', 33, 7000), ('Rajendra', 'Kumar',
32, 6000), ('Sam', 'Saxena', 43, 8000), ('Sam',
```



```
'Sung', 31, 5000), ('Shyamchandra', 'Verma', 23, 3000)]
```

The sorting is done first by index 1 element and then by index 2 element.

The next example shows how to use the attrgetter function.

```
from operator import attrgetter
class Student:
    def __init__(self, name, marks, birthYear):
        self.name = name
        self.marks = marks
        self.birthYear = birthYear
    def __str__(self):
        return f'{self.name} {self.marks} {self.birthYear}'
s1 = Student('John', 97, 1988)
s2 = Student('Sam', 89, 1987)
s3 = Student('Pam', 99, 1982)
s4 = Student('Pam', 99, 1978)
L = [s1, s2, s3, s4]
L.sort(key=attrgetter('marks'))
for i in L:
    print(i)
print()
L.sort(key=attrgetter('marks', 'birthYear'))
for i in L:
    print(i)
```

Output-

```
Sam 89 1987
John 97 1988
Pam 99 1982
```

```
Pam 99 1978
Sam 89 1987
John 97 1988
Pam 99 1978
Pam 99 1982
```

For the key parameter of the `sort` method, we have called the `attrgetter` function with the `marks` attribute. So, the list will be sorted based on the `marks` attribute. We can have multiple levels of sorting; in the next call we have sent two strings, so first it will sort by `marks` and then by `birthYear`.

Exercise

1. Which of the following will return a function object?
(A) `lambda x, y: x + y`
(B) `(lambda x, y: x + y) (2, 3)`
2. Is it possible to write a `lambda` expression without any parameters?
(A) Yes (B) No
3. `lambda` expressions can access variables only in their local scope.
(A) True (B) False
4. Which built-in function is created here using `reduce`?
`reduce(lambda x, y: bool(x) or bool(y), L)`
(A) `any(L)`
(B) `all(L)`
(C) `max(L)`
5. In the `map` function, if the argument function accepts `n` arguments then ____ iterables should be sent after that function.
(A) `n-1`

(B) n

(C) $n+1$

6. Which built-in function can be used in place of `reduce` in the following expression?

```
reduce(lambda x, y: bool(x) and bool(y), L)
```

(A) `any(L)`

(B) `all(L)`

(C) `min(L)`

7. What is the value of the following expression?

```
' & '.join(map(str, [23, 45, 67, 12]))
```

(A) `'23456712'`

(B) `'23&45&67&12'`

(C) `'23 & 45 & 67 & 12'`

(D) Shows error

8. What will this function call return?

```
any([0, 0, 0])
```

(A) `True`

(B) `False`

9. Will these two expressions give the same list?

```
list(x for x in [6, 3, -1, -7, 9] if x < 0)
```

```
list(filter(lambda x: x < 0, [6, 3, -1, -7, 9]))
```

10. Will the output of these two calls be same?

```
sorted(L, key=lambda s: len(s))
```

```
sorted(L, key=len)
```

What will be the output of the code given in questions 11 to 20?

11. `x = 5`

```

    f = lambda: sum(range(1, x + 1))
    s = f()
    print(s)
12. f = lambda x, y: [x * i for i in range(y)]
    r = f(4, 5)
    print(r)
13. (lambda x, y: x + y, x - y)(7, 3)
14. print(sorted([-22, 3, 4, -44, 32, 2], key=abs))
15. L = [(1, 'one'), (2, 'two'), (3, 'three'), (4,
    'four'), (5, 'five')]
    print(sorted(L, key=lambda t: len(t[1])))
16. L = ['spam', 'ten', 'run']
    print(list(map(tuple, L)))
17. def func(f1, f2, x, y):
    return f1(x) + f2(x, y)

    n = func(lambda x: x ** 2, lambda x, y: max(x,
    y), 4, 6)
    print(n)
18. items = {'x': 100, 'y': 50, 'z': 90, 'd': 67}
    if any(p > 100 for p in items.values()):
        print('Not everything is affordable')
    else:
        print('Everything is affordable')
19. import functools
    import operator

    L = [('X', 4), ('Y', 5), ('H', 9), ('O', 6),
    ('L', 2), ('P', 7)]

```

```

r = functools.reduce(operator.mul,
filter(lambda x: x % 2 == 0, map(lambda x:
x[1], L)))

print(r)

```

20. `def f(s1, s2):`

```

    return lambda text: s1 + ', ' + text + ' '
+ s2

```

```

f1 = f('Dear Sir', 'Thankyou')

```

```

f2 = f('Sir/Madam', 'Thanks')

```

```

f3 = f('Hi', 'Bye')

```

```

print(f1('Please contact me.'))

```

```

print(f2('Please respond.'))

```

```

print(f3('How are you?'))

```

```

print(f3('Where are you?'))

```

21. Which list will you get by writing the following expression?

```

list(filter(lambda x: x % 3, [2, 3, 6, 8, 9,
10]))

```

(A) [3, 6, 9]

(B) [2, 8, 10]

(C) [2, 6, 8, 10]

22. Which built-in functions can you use instead of the following reduce calls?

```

reduce(lambda a, b: a if a > b else b, [4, 3,
2, 7, 6])

```

```

reduce(lambda a, b: a if a < b else b, [4, 3,
2, 7, 6])

```

23. `names = [ ('Aman', 'Kumar'), ('Kamal', 'Kapoor'), ('Kamal', 'Gupta'), ('Raj',`

```
'Kumar')]
```

Get a sorted list by using the `sorted` built-in function; the sorting should be done on the second name, in reverse order of the spelling.

24. The following function `bubble_sort` is written for sorting a list.

```
def bubble_sort(a, compare=lambda x, y: x >
y):
    for x in range(len(a) - 1, 0, -1):
        swaps = 0
        for j in range(x):
            if compare(a[j], a[j + 1]):
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
        if swaps == 0:
            break
list1 = [6, 3, 1, 5, 9, 8]
bubble_sort(list1)
print(list1)
```

The second parameter `compare` takes a default argument which is a lambda function. This function will sort the list in ascending order because of the default lambda function.

(i) What will you send as the second argument if you want the sorting to be done in descending order?

(ii) What will you send as the second argument if you want to sort the following list of tuples based on the first element of each tuple in ascending order?

```
L = [('Tom', 14), ('Sam', 12), ('Ron', 19),
      ('Ken', 13)]
```

25.

```

class Teacher:
    def __init__(self, name, subject, salary):
        self.name = name
        self.subject = subject
        self.salary = salary
    def __str__(self):
        return f'{self.name} {self.subject} {self.salary}'
t1 = Teacher('Ken', 'Physics', 3800)
t2 = Teacher('Sam', 'Maths', 4000)
t3 = Teacher('Tim', 'Maths', 3500)
L = [t1, t2, t3]

```

(i) Create a new list L1 which contains the elements of list L sorted by name.

(ii) Create a new list L2 which contains the elements of list L sorted by salary.

26. Write code to replace the following code with (i) map (ii) List comprehension

```

L = []
for x in [2, 3, 4, 5]:
    L.append(x * 2)
print(L)

```

27. Write code to replace the following code with (i) filter (ii) List comprehension

```

L = []
for x in [2, 3, 4, 5]:

```

```

    if x % 2 == 0:
        L.append(x)

```

28. Write code to replace the following code with (i) map and filter (ii) List comprehension

```

L = []
for x in [2, 3, 4, 5, 6, 7]:
    if x % 2 == 0:
        L.append(x ** 3)

```

29. Use the function `reduce` to convert the following list of strings to a single string that contains all the strings of this list separated by commas.

```

L = ['pen', 'pencil', 'book', 'eraser']
'pen, pencil, book, eraser'

```

Is there any other way of doing this in Python?

30. Create a tuple from the following list; the tuple should contain only non-empty strings from the list L.

```

L = ['spam', '', 'ten', '', 'run']

```

31. L = [[1, 'Agra', 3], [4, 'Delhi', 6], [7, 'Belmont', 9], [6, 'Bareilly', 3]]

Write an expression to get the following string from the above list.

Agra-Delhi-Belmont-Bareilly

32. Create a list of those numbers from 1 to 100 that have 3, 6, or 9 as their last digit.
33. The following list contains tuples that have item names and their prices. Write an expression to calculate the sum of the prices of all the items.

```

prices = [('pen', 10), ('pencil', 3),
          ('eraser', 6), ('book', 60)]

```


34. `L = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]`

Write an expression that concatenates all the lists in the list `L` to a single list given below.

`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`

35. Rewrite the following using the `reduce` function.

`func(func(func(1, 2), 3), 4)`

36.

```
class Teacher:
    def __init__(self, name, subject, salary):
        self.name = name
        self.subject = subject
        self.salary = salary
    def __str__(self):
        return f'{self.name} {self.subject} {self.salary}'

t1 = Teacher('Ken', 'Physics', 3800)
t2 = Teacher('Sam', 'Maths', 4000)
t3 = Teacher('Tim', 'Maths', 3500)
L = [t1, t2, t3]
```

Create a list `L2` that contains the names of all the teachers sorted by salary in descending order.

37. The following dictionary contains name as key and height in inches as the value.

`d = {'Ram': 67, 'Sam': 60, 'Tom': 62}`

Using `map` and `dict` functions, create another dictionary that contains the name as key and height in centimeters.

1 inch = 2.54 cm

38. In Python 3.8, a new function named `prod` was added to the `math` module. This function calculates and returns the product of all the elements in the input *iterable*. Rewrite the following `reduce` call by using the `prod` function from the `math` module.

```
L = [('X', 4), ('Y', 5), ('H', 9), ('O', 6),  
      ('L', 2), ('P', 7)]
```

```
functools.reduce(operator.mul, filter(lambda  
x: x % 2 == 0, map(lambda x: x[1], L)))
```

39. Write a call to `reduce` function that finds out the factorial of a number. Make sure that the call works when the number is 0.
40. Find a set of all non-alphanumeric characters used in a string.

```
s = 'abc, (1,2,3), [9,10], (5,3) '          {',', '  
'}', '(', '['}
```

41. Write the following expression using an appropriate built-in function.

```
functools.reduce(operator.add, L, 322)
```

42. How will you find the longest word in a list of words?

```
words = ['it', 'that', 'paper', 'won']
```

Exception handling code gives your program the ability to handle any failures or problems that occur during execution time. This chapter will show you how to make your software robust by handling exceptions in Python. Students tend to ignore this topic, because their programs work perfectly without any exception handling also. But if you want to write any serious code, then you need a good understanding of exceptions. Let us see why exception handling is so important.

When everything is normal, the code without exception handling and the code with exception handling will both run smoothly. But in case of unusual circumstances, the code that does not know how to deal with errors will crash abruptly, while the code that includes exception handling code will be able to handle any problems that occur during execution time. Even if the problem cannot be handled, the program will have the ability to terminate gracefully.

Abnormal crashes at execution time can lead to loss of user's data and resources, and this can make your client lose trust in your application. If you do not want this, you need to write code that knows how to deal with errors. Exception handling code makes your software robust so that it can withstand any unusual circumstances and the code does not break down easily.

20.1 Types of Errors

In the first chapter, we had seen what happens when we execute our Python code. The code we write in our source files is converted into an intermediate form called bytecode by the Python compiler. The bytecode passes through

the Python Virtual Machine (PVM), which interprets this bytecode. This means that it converts the bytecode instructions to machine code instructions one by one and sends these machine code instructions to the processor for execution, and we get the output. The intermediate compilation step is hidden from the programmer; we can just type and run our program immediately. So, the source code is compiled to byte code by using the compiler and executed by using the PVM. Now, let us see how things can go wrong in this whole process.

During the compilation step, the compiler checks the syntax of each instruction and translates it to byte code. When it finds anything written in the wrong syntax, it stops the translation and displays an error message. These errors are called syntax errors or parsing errors. Syntax, as you know, is a set of rules that define how the code instructions should be written in a language. These errors occur due to the incorrect syntax of the code. For example, you might miss a colon at the end of an if statement or a def statement, miss a quote in a string literal, or use an unbalanced pair of parentheses. You must have encountered lots of syntax errors, especially when you were a beginner. When there is a syntax error in your program, and you try to run the program, IDLE shows a dialog box and it also highlights the location where the syntax error is detected in your program. You need to fix the error by making changes in your code and run the program again. Identifying and removing these errors from your program is generally not very difficult. Some IDEs can detect and highlight syntax errors as you type the program.

When all the syntax errors have been removed, the byte code is generated, and your program enters run time. This byte code goes through the Python Virtual Machine, which executes it by converting it to machine code.

So, run time is the time when your program is executing. During this time, your program will interact with the user and might be connected with multiple external resources. It will take in user input, perform processing, open files, or connect to the database or network; it will produce output to files or screen; this all happens at run time, i.e., when your program is running. Things can go wrong during the run time, also. If an error occurs during this time, then the execution of the program stops immediately, and it is terminated with an error message. Any error that occurs at this run time is called a run time error or an exception.

Run time errors can occur due to incorrect user input, problems with external resources, or any mistake in your program. For example, there will be a run time error when your program is trying to open a file that does not exist or it is trying to add an integer to a string, which is not a valid operation. The interpreter does not know what to do in these cases. It is unable to handle the situation, so it just stops the execution at the point where the error occurred. The program terminates abnormally without executing the rest of the code. This abnormal termination or crashing is not good if you have a big application running.

So, we have seen what are syntax errors and what are run time errors. If the byte code is generated successfully, it means that there are no syntax errors, and if the generated bytecode executes fully, it means that there were no run time errors. However, the absence of syntax errors and run-time errors does not mean that your program is perfect. Your program can still have errors, and these errors are called logical errors. Logical errors occur when your program runs smoothly and gives you the output, but the output that it gives is not what was intended, so your program works, but it does not do what was expected. These errors occur due to the wrong logic of the code that you have written. The problem is not with the code; the program does exactly what it has been told to do, the problem is that the programmer was not able to communicate properly the solution in the form of code, or maybe the solution that the programmer has come up with, is not correct. It could be due to small things like a missing assignment or using the wrong operator. These errors cannot be caught and reported by the compiler or the interpreter. The programmer has to identify them, and so these errors are the most difficult ones to detect and remove. You have to examine your code and debug the program, and at times, may take the help of a debugger. Debugger is a tool that helps you analyze your program while it runs.

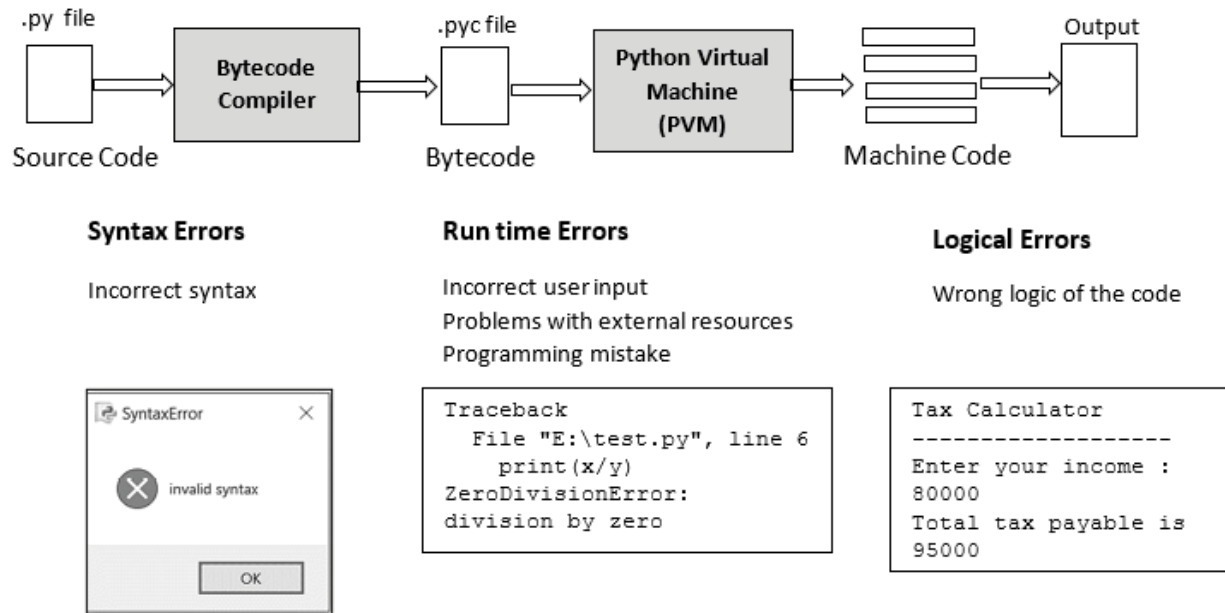


Figure 20.1: Errors

So, we have three types of errors. Syntax errors can be removed by making changes in your code. Similarly, logical errors are also removed by modifying your code. The client of your application will never see syntax errors, and if you have properly tested your application, there should be no logical errors also in your application. Run time errors are something that can cause your application to crash when your client is running it. In this chapter, we will see in detail how to deal with run time errors or exceptions. First, let us see some examples of all three kinds of errors.

In the following program, we ask the user to enter two numbers. If the first is less than the second, we multiply them; otherwise, we divide them. After this, we calculate the value of `x` and print it.

```
first = int(input('Enter a number : '))
second = int(input('Enter another number : '))
print(f'{first = }, {second = }')
if first < second:
    print(f'{first * second = }')
else:
    print(f'{first / second = }')
```

```
x = 40 / first + second
print(f'{x = }')
```

When we run this, we get a syntax error because, in the second line of the program, we missed closing the right parenthesis. After providing the parenthesis when we run, we again get a syntax error as we have missed the colon after `else`. After putting the colon, when we run the program, it executes, which means that it is syntactically correct, and we have removed all the syntax errors. When our program will run, it will ask for input.

```
Enter a number : 4
```

```
Enter another number : 5
```

```
first = 4, second = 5
```

```
Traceback (most recent call last):
```

```
File
```

```
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandling\P20_1.py", line 6, in <module>
```

```
    if first < secnd:
```

```
        ^^^^^
```

```
NameError: name 'secnd' is not defined. Did you
mean: 'second'?
```

We got a run time error, because the problem occurred when our program was running. When an error occurs at run time, Python raises an exception. `NameError` is the name of the exception that is raised, and it has an error message associated with it. This `NameError` is a built-in exception in Python, there are many more exceptions which will be seen in detail later on. We will also learn about traceback in detail later. So, Python raised the exception and terminated the program immediately and the remaining code of the program was not executed. The reason for this exception was that while executing the program, the interpreter found a name that was not defined. When we correct the spelling and run it again, there will be no run time error.

```
Enter a number : 4
```

```
Enter another number : 5
```

```
first = 4, second = 5
```

```
first * second = 20
x = 15.0
```

If we run this program with numbers 4 and 0, then again, the program will be abnormally terminated due to a run time error.

```
Enter a number : 4
Enter another number : 0
first = 4, second = 0
Traceback (most recent call last):
  File
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandli
ng\P20_1.py", line 9, in <module>
    print(f'{first / second = }')
          ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

The interpreter raised the exception `ZeroDivisionError`, and the program was abnormally terminated. This exception occurred because we entered 0 for the second number and division by zero is undefined. This exception will not occur every time the program is run, and it cannot be removed by modifying our code because it is not in control of our program, it depends on what the user inputs during the run time. For these types of exceptions, we need to include exception handling code that we will see in detail later in this chapter.

We get the following output when we run our program with numbers 2 and 3:

```
Enter a number : 2
Enter another number : 3
first = 2, second = 3
first * second = 6
x = 23.0
```

The value of `x` is not what we had expected. The formula we implemented said that you have to get the value of `x` by dividing 40 by the sum of `first` and `second`. Here, 40 divided by 2+3 should be 8 but we get 23.

The code we wrote for getting the value of `x` is `x = 40 / first + second`. The interpreter did not have any problem executing this, so it did not complain; it just did what we told it to do; the problem was that we told it the wrong thing. We should have placed the expression `first + second` inside parentheses since division has a higher precedence than addition. This is an example of a logical error in the program. This error occurs when the programmer is not able to understand and communicate the solution properly in the form of code. After enclosing `first + second` inside parentheses, when we run the program again, we will get 8 as the value of `x`.

So, we have seen examples of syntax error, run time error and logical error. Both syntax errors and logical errors can be removed by modifying the code. Users of your application will normally never see the syntax errors because they all are eliminated during the development time. For logical errors you need test driven development. A well written and thoroughly tested program will not have any logical errors.

Some run time errors are under the control of the program, these can be removed by modifying the code, while some run time errors are not under the control of the program, they have to be handled by your program. We have already seen this in our example. Let us see some more situations when run time errors can occur. Run time errors in Python occur mainly due to two reasons. Some occur due to mistakes in the code while others are caused when something unusual occurs at run time.

Here are some statements that will cause run time errors due to mistakes in the code:

```
marks = [34, 32, 45]
for i in range(4):    # IndexError: list index out of
    print(marks[i])   range
print(mark[0])        # NameError: name 'mark' is not
                      defined
marks.popitem(2)      # AttributeError: 'list' object
                      has no attribute 'popitem'
```

```

x = marks[0] / (marks[1] - 32)  #
ZeroDivisionError: division by zero
print(int('ten'))  # ValueError: invalid literal
for int() with base 10: 'ten'
print(len(12345))  # TypeError: object of type
'int' has no len()
marks2 = marks + 2  # TypeError: can only
concatenate list (not "int") to list
pow(2)  # TypeError: pow() missing required
argument 'exp' (pos 2)

```

In our first example, we have a list that contains elements up to index 2 and we are trying to access element at index 3 in the for loop and this will make the interpreter raise an `IndexError` at run time. `IndexError` is the name of the exception that is raised by Python when an attempt is made to access an out-of-range index. These exception names are built in Python and we will learn about them in detail later. An error message is also displayed with the exception name.

In the next statement, we have misspelled the variable name; we wrote `mark` instead of `marks`, so here we are trying to access a variable that does not exist and, in this case, the interpreter will raise a `NameError` exception. The next statement will make the interpreter raise an `AttributeError` exception since the list object does not have any method named `popitem`. In the next statement, we will get `ZeroDivisionError` because the denominator becomes zero. The expression `int('ten')` will give a `ValueError` exception since `int` function cannot convert the given string to `int`. In the next three statements we get `TypeError` due to different reasons.

All these errors can be removed by understanding the error message displayed by Python and modifying the code accordingly.

Now, let us see some run time errors that occur due to unusual events at run time:

```

import stack  # ModuleNotFoundError: No module
named 'stack'

```

```

f = open('data.txt', 'r') # FileNotFoundError:
[Errno 2] No such file or directory:
s = f.read()
print(s)
boys = int(input('Enter number of boys '))
girls = int(input('Enter number of girls '))
# ValueError: invalid literal for int() with base
10: 'fifteen'
print('Ratio of boys to girls is', boys / girls) #
ZeroDivisionError

```

These statements are syntactically correct, and they will not result in any run time error most of the time. But in some unusual cases they might result in run time errors. For example, `import stack` can result in `ModuleNotFoundError` if Python is not able to locate the module named `stack`. In the next few statements, we are opening the file `data.txt` and reading data from it. This will work correctly if the file exists, but it can result in a run time error if the file is not found or the user does not have permission to read the file.

In the next two statements, we are sending the text entered by the user to the `int` function. These statements will work correctly if the user enters data that can be converted to `int`, but if user enters some invalid data (like `'fifteen'`), then these statements can result in a run time error.

The next statement will result in `ZeroDivision` error if the user enters zero as input for a number of girls.

These errors are not due to any mistake in your code, and they do not always show up when the program is run. They occur only in rare situations. These errors can be due to bad user input or some problems related to external sources your program uses while running. For example, your program might be connected to a network or a database, and an error occurs in that external source, or there could be insufficient memory at run time, or some hardware failure or you might be opening a file that does not exist or importing a

module that is not installed. Any of these issues can cause run time errors in your program.

Whatever the reason, a run time error will terminate your program, and it will be an abnormal termination, which could be harmful. So, as a programmer, you need to make sure that your program knows how to handle these run time errors that occur due to unusual events. These types of errors cannot be removed by modifying the code that you have written because there is nothing wrong with your code. They were not caused due to any problem in your code, they were caused by events that are not under the control of your program.

Your program should know what to do when these unexpected events occur. For that you need to add some code in your program, and that code is called error handling code or exception handling code. You have to anticipate events that can go wrong at run time, and include code to handle those events.

So, we saw that run time errors can be caused due to mistakes in the code or due to unusual events at run time. If a mistake in the code causes the exception, then the programmer just needs to figure out what is wrong in the code and can modify the code to fix the error. If the exception is caused by some unusual event at run time, then the programmer needs to write exception handling code so that the program does not crash abruptly at run time. In the next section, we will learn about the two strategies that can be used to write the exception handling code.

20.2 Strategies to handle exceptions in your code

There are two approaches that can be followed when we want to deal with exceptions that occur due to unusual events:

1. LBYL - Look Before You Leap
2. EAFP - Easier to Ask for Forgiveness than Permission

In the LBYL approach, we avoid exceptions, while in the EAFP approach, we handle exceptions.

First, let us see the “Look before You Leap” approach. In this approach, we use conditional checks to eliminate any possibility of error. Whenever we have to perform any error-prone operation, first, we make sure that all conditions are favorable for the execution of that operation. We check all the situations that can make the operation give errors and we do this by placing conditional checks in the form of `if` statements. When the conditions are favourable, then only we execute the operation. If there is any possibility of error, we do not execute the operation.

Let us understand this with the help of an example that we saw in the previous section:

```
print('Ratio of boys to girls is', boys / girls)
```

This operation is error-prone because if number of girls is zero then it gives us `ZeroDivisionError` which is a run time error and it will terminate our program. If we follow the LBYL coding style, then before executing this operation we will put a check; we perform this division only when number of girls is not equal to zero.

```
if girls == 0:
    print('No girls, Ratio not defined')
else:
    print('Ratio of boys to girls is', boys / girls)
```

So, we prevented the error from occurring by putting an `if` statement. In the LBYL approach, there will be a lot of checking done before the operation actually executes because we have to make sure that nothing goes wrong while performing that operation. That is why there will be lot of `if` statements when we write our error handling code using this coding style.

The other approach is EAFP, “Easier to ask for forgiveness than permission.” This is based on a famous quote by Grace Hooper. In this approach, we try to execute the code assuming that everything will work correctly, but if our assumption proves false and something goes wrong, we deal with it. Python supports this approach with the help of `try...except` statement. We will see the details of this statement in the coming sections.

```
try:
```

```
    print('Ratio of boys to girls is ', boys /
girls)
except ZeroDivisionError:
    print('No girls, Ratio not defined')
```

We are performing the error prone operation, and if an exception is raised during execution of this operation, we handle it. Since the raised exception is handled, the program will not be abnormally terminated.

In the LBYL approach, we are very cautious; we do not give Python a chance to raise the exception, while in EAFP, we just go ahead and execute the operation, and if anything unusual occurs and Python raises an exception, then we just handle that exception. From these two approaches, EAFP is more Pythonic; it is commonly used by Python programmers, while LBYL is common in other languages like C, which don't have any exception-handling mechanism.

Let us compare the two approaches and see some benefits of using the EAFP approach.

Since exceptions are rare, the code will work correctly most of the time. For example, the user input will be correct most of the time, only occasionally it will be wrong. Similarly memory overflow or network errors will also not occur very often. In LBYL, all the if statements are processed every time the code is run, even when everything is okay. This increases the execution time. The extra conditions put extra overhead on code processing.

In EAFP, the error handling mechanism is processed only when an exception occurs, not every time the code runs. So, this approach results in efficient running in usual cases when the error does not occur. We know that most of the time, errors do not occur, so using this approach results in better performance of our program.

In LBYL, sometimes you must duplicate a part of the operation in the conditional check, which again results in extra processing.

Another advantage of the EAFP approach is that the error handling code does not get mixed with your mainstream code. The error handling code is separate from the mainstream code; the mainstream code is in one block, and the error handling code is in a separate block. This separation makes the program more readable and easier to debug or modify. In LBYL, the if

statements are mixed with the main logic of the program. This reduces program clarity and readability and makes it difficult to understand or modify the program thus, code management becomes difficult. In EAFP, the emphasis is on the mainstream code, while in LBYL, the emphasis is on the conditions, and the main code gets rather hidden at the end, which is not good for program readability.

Sometimes it is not possible to predict exactly what errors can occur in an operation, or the programmer may miss checking some conditions. In those cases, it is better to attempt the operation and then catch the error.

Another problem with LBYL approach is that sometimes the circumstances can change between the looking and the leaping step, so when you checked the conditions, everything was fine but while attempting the operation, something goes wrong; this could happen in a multithreaded environment. If we use EAFP approach, there will be no such problem.

20.3 Error Handling by Python (Default exception handling)

In the previous section, we saw that we can write the exception handling code by using the `try...except` statement. Before seeing how to write the exception handling code using this statement, let us first see what exactly happens when a run time error occurs.

We have seen that when the program is executing, and an error occurs at any point, Python stops the program execution at that point and raises an exception. The program is terminated and the exception name with a message is displayed. Now the question is: what is an exception and what does it mean when we say that Python raised an exception? We know that everything in Python is an object, and so exceptions are also objects.

There are many built-in classes in Python that are dedicated to errors. Here are some of them:

<code>IndexError</code>	<code>NameError</code>	<code>AttributeError</code>
<code>ZeroDivisionError</code>		
<code>ValueError</code>	<code>TypeError</code>	<code>ModuleNotFoundError</code>
<code>FileNotFoundError</code>		

An exception is an instance of one of these classes. So, whenever a run time error occurs, the interpreter stops the normal control flow of the program and creates an instance of the appropriate class. For example, if there is an attempt to divide by zero, then the interpreter creates an instance of the `ZeroDivisionError` class. This is called exception object or simply exception and it contains information about the error that has occurred. After creating the exception object, the run time system starts searching for exception handling code in the program. If it finds the code to handle the particular type of exception that was created, then it executes that code and continues with the rest of the program. If it does not find a suitable exception handling code, then it terminates the program. This search for an appropriate exception handler for a raised exception is called exception propagation. Let us understand how this search happens.

Python creates a stack of function calls during the program execution. This stack is made up of frames; there is a frame for each function call. When a function is called, a frame is pushed on the stack, and when a function returns, a frame is popped from the stack.

Let us take an example program and see how the call stack is maintained and what happens when an exception occurs:

```
def f1():
    print('function f1 statement 1')
    f2()
    print('function f1 statement 3')

def f2():
    print('function f2 statement 1')
    f3()
    print('function f2 statement 3')
def f3():
    print('function f3 statement 1')
    x = int(input('Enter a number : '))
    print('function f3 statement 3')
print('Program begins')
```



```
f1()
print('Program ends')
```

Output-

```
Program begins
function f1 statement 1
function f2 statement 1
function f3 statement 1
Enter a number : one
Traceback (most recent call last):
  File
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandli
ng\P20_4.py", line 17, in <module>
    f1()
  File
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandli
ng\P20_4.py", line 3, in f1
    f2()
  File
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandli
ng\P20_4.py", line 8, in f2
    f3()
  File
"E:\Deepali\BOOK_Python\Programs\20_ExceptionHandli
ng\P20_4.py", line 13, in f3
    x = int(input('Enter a number : '))
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10:
'one'
```

Initially, a global frame is pushed on the stack, then function `f1` is called so a frame for this call is pushed. Inside function `f1`, function `f2` is called, so a frame for this call is pushed, and then inside the function `f2`, function `f3` is called so a frame for this is also pushed on the stack. While executing

function `f3`, for the input call, the user entered something that could not be converted to `int`, so a `ValueError` exception will be raised.

Python will stop the execution at this point and create an instance object of type `ValueError`. It will then look for some exception-handling code in the current function `f3`. If it does not find any such code there so it terminates the current call and goes to the next calling function, so it comes to `f2`. The remaining statements of the function `f3` will not be executed. Inside function `f2` also, it could not find any exception handling code, so it terminates this call and goes to next function which is function `f1`. The remaining statements of function `f2` will not be executed. In `f1` also, there is no exception handling code, so this call also terminates, and control goes to the module level. Here also, it does not find any exception handling code, so the program is terminated here. The remaining statements of the program will not be executed. This exception propagation mechanism involves the unwinding of the call stack until a suitable exception handling code is found in any function.

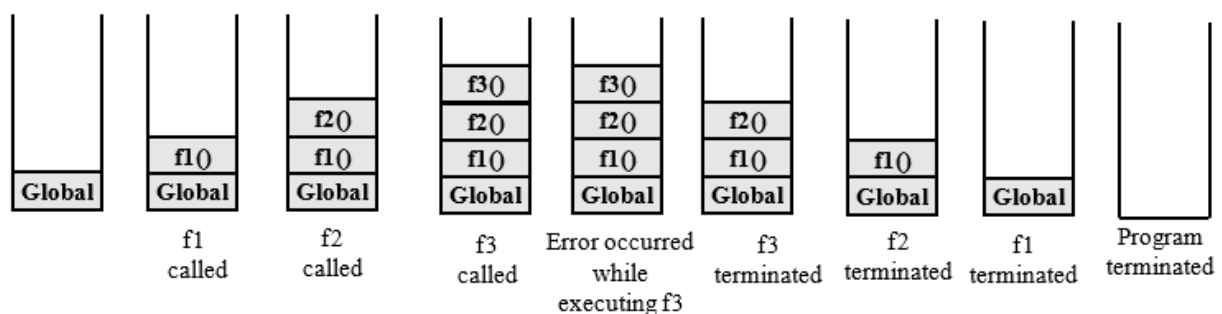


Figure 20.2: Call stack

If an exception occurs in a function and is not handled there, then it bubbles up or propagates to the calling function; if not handled there also, then it goes up to the next calling function, and this propagation continues till it reaches the main module. If the programmer has not provided any exception handling code in the main module also, then the program terminates, and an error message is shown. So, the exception bubbles up to the top level of the program and the default exception handler comes into action, it terminates the program and prints the error message.

From the output of the program, we can see that along with the exception name and message, a traceback of function calls is also shown. It contains

details of all the function calls that were terminated when the exception was propagated. We need to understand how to read the traceback to locate unhandled exceptions.

The traceback prints the names of the files and the functions that were executed till the exception occurred. For each call, we have 2 lines; the first line contains the filename, line number, and the function name, and the second line shows the line of code. The function calls are listed from least recent to most recent, so traceback should be read from the last line to the first line.

In the last line, the name of the exception and a brief error message about it are displayed; the message generally tells you the reason for the exception. Before the last line, we can see the function call where the error occurred and then we have the function that invoked it, and so on. So, the traceback gives us the details of the call stack; in our example, it tells us that the `ValueError` exception occurred in function `f3`, and then it propagated to function `f2`, and then to function `f1`, and then to the main module. If any function is in a different file, then that filename will be shown in the traceback. In complicated programs, seeing just the most recent call will not help identify the cause of the problem, so in those cases reading this traceback helps clearly identify the origin of the problem.

So, when an exception is raised and there is no exception handling code, then the default behaviour is abnormal termination of the program. This default behaviour is not ideal and it is definitely not what you want most of the times, as this can lead to loss of data and resources. For example, suppose in a program you are opening a file, reading and writing to the file and closing the file.

```
Open file
```

```
Read from the file
```

```
Write to file
```

```
Close the file
```

If some error occurs during the writing of the file, the program will terminate abruptly and the subsequent statements will not be executed, and so, the file will not be closed. This can lead to loss of data in the file. It is important to close any opened file to avoid any problems with the file. Similarly, if you

are reading from a database, and some error occurs then that database connection will not be closed. So abnormal termination can lead to waste of resources. If we do not want the default Python handler to terminate the program, then we have to write code to catch and handle the exceptions in our program.

So, for run time errors we have two options, we can either let Python handle these errors which means that the program will be abnormally terminated or we can catch them and handle them in our program by writing exception handling code. The exception handling code can either fix the error at run time and continue execution or it can terminate the program gracefully.

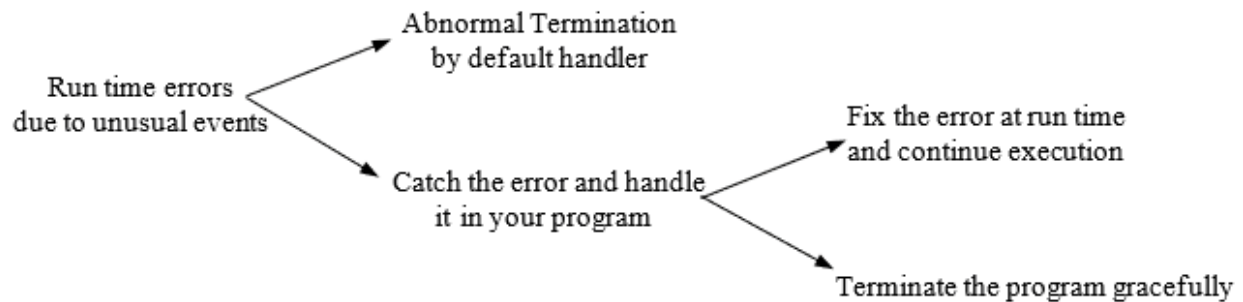


Figure 20.3: Handling run time errors

In the next section, we will have a brief look at the built-in exception classes and then in the subsequent section, we will learn how to write the exception handling code using the `try...except` statement.

20.4 Built-in Exceptions: Python Exceptions Class Hierarchy

In the previous section, we saw that when an error occurs at run time, Python creates an exception object which is an instance object of one of the built-in exception classes. Exception classes in Python can be of two types: built-in classes or custom classes. Built-in classes are predefined in Python; they are also called standard exceptions. Custom exception classes are defined by the programmer for their special needs. We will see how to define our own exception classes and raise our own exceptions later. In this section, we will discuss the built-in exception classes.

The built-in exception classes are organized in a hierarchy using inheritance. This inheritance structure of exception classes categorizes exceptions into different types. Python raises them in many different situations as we have seen, and mostly all the modules in the standard library also use these exceptions.

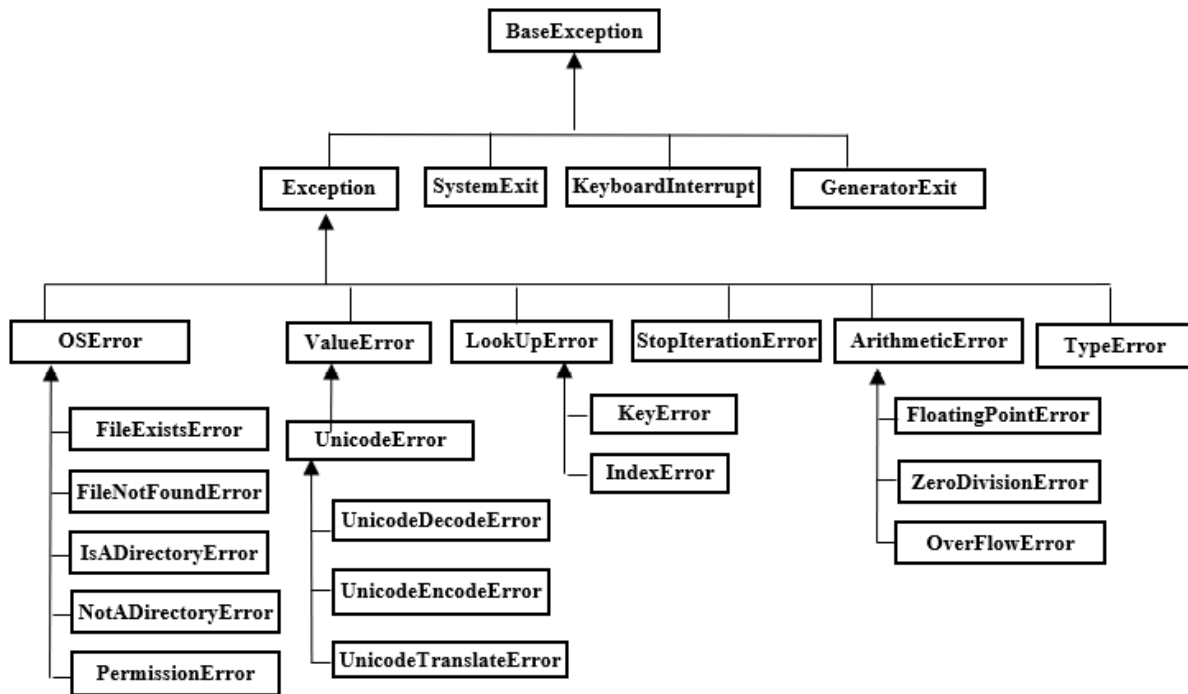


Figure 20.4: Built-in exceptions

The class `BaseException` is the base class of all the built-in exception classes. From `BaseException`, four classes named `Exception`, `SystemExit`, `KeyboardInterrupt` and `GeneratorExit` are derived. All the remaining built-in exception classes are derived directly or indirectly from the `Exception` class. The figure shows some of the classes derived from `Exception` class; there are many other classes also.

These classes are all present in `builtins` module, you can import this module and use `dir` function on it to see all the exception class names.

```

>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException',

```

```
'BaseExceptionGroup', 'BlockingIOError', .....  
tuple', 'type', 'vars', 'zip']
```

To see the built-in exceptions inheritance tree, you can use help.

```
>>> help(builtins)
```

To see the class hierarchy of an exception class, we can see its `__mro__` attribute .

```
>>> IndexError.__mro__  
(<class 'IndexError'>, <class 'LookupError'>,  
<class 'Exception'>, <class 'BaseException'>,  
<class 'object'>)
```

We can see that `IndexError` class is derived from `LookupError`, which is derived from the `Exception` class, and the `Exception` class is derived from the `BaseException` which is derived from the `object` class.

To see the method resolution order, you can also use help on that particular exception.

```
>>> help(ZeroDivisionError)
```

Help on class `ZeroDivisionError` in module `builtins`:

```
class ZeroDivisionError(ArithmeticError)  
    | Second argument to a division or modulo  
operation was zero.
```

```
    |  
    | Method resolution order:  
    |     ZeroDivisionError  
    |     ArithmeticError  
    |     Exception  
    |     BaseException  
    |     object
```

```
.....  
.....
```

In Python 2, it was possible to write string-based exceptions, but in Python 3, we have only class-based exceptions. Let us see the benefits of defining

the exceptions as class instance objects.

The object-oriented way to represent exceptions helps to pack more information about the exception inside the object because an instance object can store both state and behavior. The extra information can be used inside the handler for handling the exception.

Classes support inheritance, so we can categorize the exceptions and arrange them in a hierarchy. The inheritance structure helps you to write handlers that can catch a wide range of related errors. This is because exceptions written in handlers, are matched by inheritance relationship. Let us see what this means. If an exception is mentioned in a handler, then it will handle that particular exception and it will also handle any subclass of that exception. For example, in your code, if you write a handler to handle the `ArithmeticError` then it will handle `ArithmeticError` as well as `FloatingPointError`, `ZeroDivisionError` and `OverflowError` as they are subclasses of `ArithmeticError`. Similarly, a handler for `LookupError` can handle `LookupError` and both `KeyError` and `IndexError` as they are derived from `LookupError`. So, if we want to catch a whole category of errors, we can just specify the superclass in the handler. If we write a handler for the `Exception` class, then it catches almost all exceptions except `BaseException`, `SystemExit`, `KeyboardInterrupt`, and `GeneratorExit`. This is a very broad catch and is written only in a few situations.

So, the inheritance hierarchy helps us write general or specific handlers for handling exceptions. Another advantage of object-oriented exceptions is that we can easily define our own exceptions by deriving from built-in exceptions, we will see later how to do that. Due to this inheritance hierarchy of exceptions, we can add new exceptions in the future without breaking the existing error-handling code.

Inheritance helps to achieve common behavior for exception classes. The default behavior is defined in the superclasses and is inherited by the subclasses.

Let us see the details of a few exception classes.

`BaseException`: This is the root class for all the exception classes. The default display and state retention behavior is defined in this class and is inherited by all the subclasses.

`Exception`: This is the base class for most of the built-in exception classes. All user-defined exceptions are also supposed to be inherited from this class, not from the `BaseException` class. This is because when you mention `Exception` in a handler, then it will not catch the 3 exceptions that signal system exit events.

`KeyboardInterrupt`: This exception is raised when the user interrupts the execution of the program, generally by typing Ctrl-C or Delete. It sends the interrupt signal to the Python interpreter. When the user interrupts the program using Ctrl+C for whatever reason, he expects the interpreter to exit, so this exception is never handled in the program. It inherits from the `BaseException` instead of the `Exception` so that it is not caught by the general handler, which tries to catch all exceptions by mentioning `Exception` in the handler. If this exception is caught, then the interpreter will not exit, which is not the expected behavior when the interrupt key is pressed. So, this is one of the exceptions that is not supposed to be caught.

`SystemExit`: Another exception that should not be handled is the `SystemExit` exception. It is raised by the `sys.exit()` function. When this function is called, the interpreter should exit. So, like `KeyboardInterrupt`, this one should also be allowed to propagate up and cause the interpreter to exit. This is not meant to be handled, and that is why this also inherits from the `BaseException` instead of `Exception`. So, it will not be caught by any code that catches `Exception`.

`GeneratorExit`: This is also derived from `BaseException` and is raised when a generator or a coroutine is closed. We do not need to write handlers for this also as it is technically not an error.

So, every exception does not necessarily denote an error. Some exceptions are raised by Python to indicate some special events that are not errors.

`StopIteration`: One more non-error exception is the `StopIteration` exception. It is raised when the `next()` function is called for an exhausted iterator. The exhaustion of an iterator is not an error or an abnormal situation. It just signals the condition that the iterator has no

more items to produce. When an iterator is used in a for loop, the exception is handled by the for loop to end the loop iterations.

Now, let us see some exceptions that indicate errors.

`AttributeError` is raised when you try to access an attribute using a dot notation, but the attribute does not exist.

`NameError` is raised when an identifier that has not been defined is used.

`IndexError` is raised when the index of a sequence is out of range.

`KeyError` is raised when a key is not found in the set of dictionary keys.

`TypeError` is raised when an operation or function is applied to an object of inappropriate type. For example, when you add an `int` and a `str` value (like `1 + 'x'`) or call `min()` without any argument, it expects at least one argument.

`ValueError` is raised when a function receives an argument that has an inappropriate value. For example, the calls `int('on')`, `sqrt(-9)` will raise this error.

`ZeroDivisionError` is raised when there is an attempt to divide by zero.

`ImportError` is raised when the import statement fails, this can happen when a module is not found or a name in a module is not found.

`MemoryError` is raised when an operation runs out of memory.

`AssertionError` is raised when an assert statement fails.

`IOError` is raised when an input or output operation fails.

`SyntaxError` is raised when the parser encounters a syntax error. For example, when you import a module, and there is a syntax error in the code of that module, then this exception is raised.

`RuntimeError` is raised when the error does not fall into any category.

20.5 Customized Exception Handling by using `try...except`

An exception that is raised can be caught and handled by the exception handling code. We have seen that if a raised exception is not caught, then the Python interpreter terminates the program and reports an error message and a traceback to the console. If you do not want the program to terminate abnormally, then you need to catch and handle the exception that is raised by Python. You need to write your error handling code using the `try` statement. `try` statement is a compound statement with different clauses, it can take one of these two forms.

<pre>try: except ExceptionA: except ExceptionB: except ExceptionC: else: finally: </pre>	<pre>try: finally: </pre>

Figure 20.5: Two forms of try statement

In the first form, we have a `try` block with one or more `except` clauses followed by an optional `else` clause and an optional `finally` clause. In the second form, we have a `try` block with a `finally` clause. We will learn about different clauses and their details later in this chapter. Let us start with the most basic form of the `try` statement in which we have a single `except` block following the `try` block.

```
try:
    .....
    .....
    .....
except ExceptionA:
    .....
    .....
```

We have the keyword `try` followed by a colon, and inside the `try` block, we will have those statements that we think can cause exceptions. After that, we have the keyword `except` followed by an exception name and a block of statements. If an exception occurs at any statement inside the `try` block, then the interpreter stops executing the `try` block. The remaining statements in the `try` block are not executed, and the control jumps to the `except` clause. If the exception raised inside the `try` block matches the exception written in the `except` clause, then the code in the `except` block is executed, and after that, the control is transferred to the next statement after the whole `try...except` statement.

The `except` block is also known as the exception handler since it includes the code to handle the exception.

If the raised exception does not match the exception mentioned in this `except` clause, then it is propagated up, and if it does not find any suitable handler, then the program is terminated. In any case, whether the exception matches or not, the remaining statements of the `try` block are skipped. They are never executed. So, that is why you should keep your `try` block as short as possible; it should contain only the error-prone code. It is not good to enclose your whole program or a big part of your program inside the `try` block.

While writing the code, you must identify the statements that can cause exceptions. For example, suppose you have the following code, and you suspect that statements 6, 7 and 8 can cause `ValueError` at run time.

```
statement1
statement2
statement3
statement4
statement5
statement6
statement7
statement8
statement9
statement10
```

```
statement11
statement12
```

You can put the error-prone statements inside a `try` block, and immediately after the `try` block, you have to write the `except` block, with the exception name `ValueError`. Inside the `except` block, you can write the code to handle `ValueError`.

```
statement1
statement2
statement3
statement4
statement5
try:
    statement6
    statement7
    statement8
except ValueError:
    statementX
    statementY
statement9
statement10
statement11
statement12
```

Now, let us see how this code will behave in different scenarios.

If no exception occurs, then all the statements from 1 to 12 will be executed. First, statements 1 to 5 will be executed, then statements 6 to 8, and then statements 9 to 12. The `except` block will be skipped.

Now, let us see what happens when inside the `try` block, a `ValueError` exception is raised at statement 7. First, statements 1 to 6 are executed. At statement 7, the `ValueError` exception is raised, so the control jumps to the `except` block. The rest of the `try` block is not executed, so statement 8 is not executed. The raised exception matches the one mentioned in the

`except` clause, so the code inside the `except` block is executed, and it handles the `ValueError` exception in whichever way it can. After executing the statements X and Y, the rest of the program continues, so statements 9 to 12 will be executed.

If inside the `try` block, an exception is raised which is a subclass of `ValueError` exception then also the control flow will remain same because exception match occurs for the subclasses also. For example, suppose `UnicodeDecodeError` or `UnicodeEncodeError` is raised in `try` block, then also the `except` block will be executed since these exceptions are derived from `ValueError`.

If any exception other than `ValueError` or its subclasses is raised in the `try` block, then it will not match the exception in the `except` clause and so in that case the error will be propagated up, if this code is part of a function or is enclosed in another `try` block. If the exception is not handled anywhere then the program is abnormally terminated.

If any exception is raised outside the `try` block, whether it is `ValueError` or any other exception, then it will be propagated up if it can be.

So, in the code that we had written without the `try` statement, we were telling Python to execute the statements from 1 to 12. When we include the `try` statement in our code, we are telling the same thing to Python, we want it to execute the statements 1 to 12, but now we are also telling that in case a `ValueError` exception occurs while executing statements 6,7 or 8 then execute the statements X and Y and continue.

Let us see an example program where we can include the `try...except` statement. In the following code, we enter the number of boys and girls and then find the ratio and the total number of students.

```
boys = int(input('Enter number of boys '))
girls = int(input('Enter number of girls '))
r = boys / girls
print(f'Ratio of boys to girls is {r}')
total = boys + girls
print(f'Total number of students = {total}')
```

Here are two sample runs of this program:

Sample Run 1-

```
Enter number of boys 50
Enter number of girls 25
Ratio of boys to girls is 2.0
Total number of students = 75
```

Sample Run 2-

```
Enter number of boys 50
Enter number of girls 0
Traceback (most recent call last):
  File "C:\Users\deepali\test.py", line 4, in
<module>
```

```
    r = boys/girls
```

```
ZeroDivisionError: division by zero
```

In the first run, there was no problem, but in the second run, we got a `ZeroDivisionError` and the program was abnormally terminated. To avoid this, let us write a `try...except` statement.

```
boys = int(input('Enter number of boys '))
girls = int(input('Enter number of girls '))
try:
    r = boys / girls
    print(f'Ratio of boys to girls is {r}')
except ZeroDivisionError:
    print('No girls, Ratio not defined')
total = boys + girls
print(f'Total number of students = {total}')
```

Sample Run 1-

```
Enter number of boys 50
Enter number of girls 25
Ratio of boys to girls is 2.0
Total number of students = 75
```

Sample Run 2-

```
Enter number of boys 50
Enter number of girls 0
No girls, Ratio not defined
Total number of students = 50
```

Now, when the `ZeroDivisonError` occurred, the `except` block was executed, and after that, the program continued normally.

So, while writing the code, if you suspect that some lines of code can raise exceptions then you should put them inside `try` blocks and write appropriate `except` handlers.

20.6 Catching multiple exceptions using multiple except handlers and single except handler

We saw the following example in the previous section. If the statements 6, 7, 8 raised a `ValueError` exception, then the code handled it by executing statements X and Y.

```
statement1
statement2
statement3
statement4
statement5
try:
    statement6
```

```

        statement7
        statement8
except ValueError:
    statementX
    statementY
statement9
statement10
statement11
statement12

```

It is possible that the statements in the `try` block can raise several different types of exceptions at run time. For example, suppose you anticipate that the statements 6, 7, and 8 could raise a `ValueError` exception, a `TypeError` exception, or a `ZeroDivisionError` Exception. You want to respond to each exception in a different way, so you need to write a separate handler for each exception. You can easily do this because a `try` statement can have multiple `except` clauses. So now let us add two more `except` clauses in our `try` statement:

```

.....
try:
    statement6
    statement7
    statement8
except ValueError:
    statementX
    statementY
except TypeError:
    statementP
    statementQ
except ZeroDivisionError:
    statementR
.....

```


When an exception occurs in the `try` block, that exception is compared with the exceptions mentioned in the `except` clauses. The `except` clauses are checked in order starting from the first one. When a match occurs, the corresponding `except` block executes, and the program continues after the `try` statement, which means that the control goes to the statement after the last `except` block.

We know that an exception match occurs if the exception raised is of the same type as the one specified in the `except` clause or if it is of a type derived from the one mentioned in the clause. So, in this code, if `ValueError` or any of its subclass exceptions occur in `try` block then the statements X and Y will be executed and if `TypeError` occurs then statements P and Q will be executed and if `ZeroDivisionError` occurs then statement R will be executed.

If the raised exception does not match any of the exceptions mentioned in the clauses, then the exception propagates up and the program will terminate if no handler is found.

So, if more than one type of exception is anticipated in the `try` block, then you can write a separate `except` block for handling each type of exception.

When you write multiple `except` blocks, you can end up writing them in such a way that more than one block matches an exception that is raised. When an exception is raised, only one `except` block can execute, so the first one that is matched will be executed. Let us see an example of this:

```
try:
    .....
    .....
except LookupError:
    print('Lookup Error')
except IndexError:
    print('IndexError')
```

Suppose an `IndexError` is raised in the `try` block, first it will be compared with the exception in the first `except` clause. The match is successful since the `IndexError` is a subclass of the `LookupError`. So,

this first `except` block will execute and the `except` block with `IndexError` as exception will not be executed.

Thus, if you have multiple `except` blocks then their order is important. The blocks with derived exceptions should be placed before the blocks with base exceptions. The correct order of `except` clauses for the previous `try` statement is this:

```
try:
    .....
    .....
except IndexError:
    print('IndexError')
except LookupError:
    print('Lookup Error')
```

So, you should write the `except` clauses such that the more specific exceptions come before the generic ones.

If same actions need to be performed in case of different exceptions, then you can specify multiple exceptions in a single `except` block. For example, in the following `try` statement, we have to execute statements X and Y in case of `ValueError` and `TypeError`.

Instead of writing separate `except` blocks and duplicating the code, we can write both the exceptions in a single `except` block.

<pre>try: statement6 statement7 statement8 except ValueError: statementX statementY except TypeError: statementX statementY except ZeroDivisionError: statementR</pre>	<pre>try: statement6 statement7 statement8 except (ValueError, TypeError): statementX statementY except ZeroDivisionError: statementR</pre>

Figure 20.6: *try statements with multiple except blocks*

We have enclosed both the exceptions inside a tuple, so the statements X and Y will be executed if any of these exceptions `ValueError` or `TypeError` occurs. In this case, we have specified two exceptions. We can have more than two also if required. So, if different exceptions can be handled by using the same statements, then we can group together the exceptions in a tuple, and specify the tuple in the `except` clause.

We know that we have an inheritance hierarchy for built-in exception classes, and if we specify the base class in an `except` clause, we can catch all the derived exceptions. In the following code, we are handling both `KeyError` and `IndexError` using a single `except` clause. These errors are derived from `LookupError`, so instead of specifying both the errors in the `except` clause, we can just write `LookupError`.

<pre>try: except (KeyError, IndexError): statementX statementY except ZeroDivisionError: statementR</pre>	<pre>try: except LookupError: statementX statementY except ZeroDivisionError: statementR</pre>

Figure 20.7: try statements with multiple except blocks

If you want to catch all the exceptions derived from the `Exception` class, then you can specify `Exception` in the `except` clause.

```
try:
    .....
    .....
except Exception:
    statementP
```

This `except` block will catch all the built-in exceptions except `BaseException` and the other three classes derived from `BaseException`. If you specify `BaseException` in the `except` clause, then it will catch all the exceptions, including `KeyboardInterrupt`, `SystemExit`, and `GeneratorExit`. We have seen earlier that these three exceptions are not supposed to be caught,

they should always be allowed to propagate up. So, whenever you want to write a broad exception handler, it is better to specify `Exception` instead of `BaseException`. If the broad exception handler is used, then it should be placed at the end. If placed anywhere else, then all the `except` blocks after it will never be executed.

It is not considered a good practice to use `except Exception` since it catches almost all exceptions and so it can mask many errors in our code. It will silently catch errors that we have not anticipated and which we do not intend to handle. If you use this handler, then it is important to print the error message or log it somewhere so that you can debug the application and know the reason for the error.

It is a good practice to catch and handle specifically each exception. But if you have to catch all, then you need to show the error information or propagate it further, so that the error information is not lost. In a subsequent section, we will see how to propagate an exception explicitly by using the `raise` statement.

The `except` clause can be written even without any exception name. It is called the default `except` clause and it should always be placed at the last, if it is placed somewhere else then we will get a syntax error.

```
try:
    statement6
    statement7
    statement8
except LookupError:
    statementX
    statementY
except ZeroDivisionError:
    statementR
except:
    statementP
```

This `except` block is a catch-all block. It catches all the exceptions, including `BaseException`, and all those derived from it. So, if no name is

provided in the `except` clause, then it handles all exceptions.

This clause will match any exception that is raised, and that is why it is placed at the end. Any error other than those specified in the `except` clauses will be caught by this default `except` clause placed at the end. It is not a good idea to use this bare `except` since it will catch all errors and thus can hide bugs in your program. It does not help you to figure out the exact cause of the problem as it masks the real error. There is not much that you can do in this block except print some generic message that an error has occurred and log some information. So, this bare `except` clause is not recommended, except in cases where we just need to catch an exception and raise it further.

20.7 How to handle an exception

In this section, we will see different ways to handle an exception. We know that when an exception is raised and is not handled anywhere in the program, then the program will abnormally terminate which is bad, as it may lead to loss of user's data or resources.

An exception generally denotes that an error has occurred and there is a problem. All problems are not fatal and in many cases, our program can continue in spite of the problem. Even if the program cannot continue, we can at least control how it terminates.

To deal with the problem that has occurred, we write exception handlers which are the `except` blocks that we have seen. Now, let us see what are the things that we can possibly do inside the exception handlers.

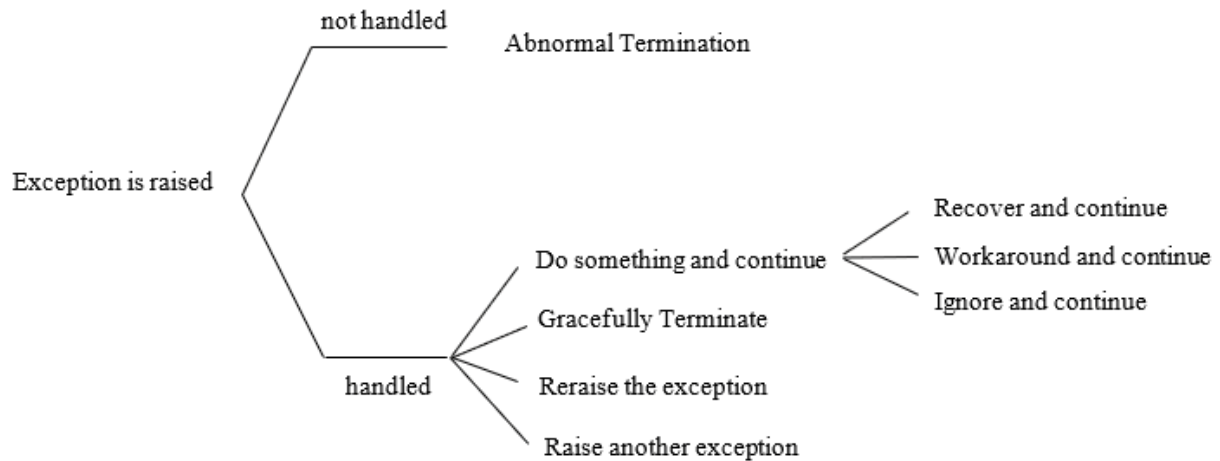


Figure 20.8: How to handle an exception

When we catch the exception, we can do something inside the handler and continue rest of the program normally. Let us see what all can we do.

We can fix the error if possible, so we can fully recover from the error and then let the program continue normally. For example, if the user is trying to open a non-existent file, then we can ask the user to enter another filename, the program need not crash because of the problem. This is a recovery-oriented approach.

If it is not possible to recover from the error, then we provide a workaround and let the program continue. So, in this case, we follow an alternative path, and the remaining code executes normally. For example, if a file that the user tries to open is not available, then we could provide a local file for use and let the program continue. You could use the exception to change some value. For example, in the following code if we get an `IndexError`, we set `x` to `-1`.

```
try:
    x = lst.index[i]
except IndexError:
    x = -1
```

In some cases, we can just ignore the exception and continue. For example, if we are reading many files and collecting data from them, and an exception occurs in reading a particular file, then we can just ignore that exception and

continue reading the rest of the files. So, in spite of the error the application will continue to execute.

Another option could be to gracefully terminate the program. There are some exceptions that are not recoverable, and it is not possible to ignore them or find a workaround. We cannot continue the program, so in these cases our goal is to just terminate the program as elegantly as possible. Graceful termination means proper error reporting and releasing all the resources allocated by the program before terminating the application. We basically try to minimize the harm that is caused by an abnormal crash.

If we catch the error but are not able to continue executing the program, we can show some friendly error message to the user instead of the long and confusing traceback that is shown on abnormal termination. Tracebacks can be really long in big applications that have lots of function calls. If the client using your application is non-technical, then he will be intimidated by the whole traceback thing, and the abnormal program crashes will make him lose trust in your application. Showing the whole traceback also exposes the internal details of your program, which is sometimes not desirable. So, instead of the long traceback, you can show your user some simple error message. If you want the user to take some corrective action in case of an error, then in the error message, you can show some error details, which will help the user know what the problem is, and you can also tell the user what to do to avoid the problem. For example, if the user does not have permission to access a file, then we can alert the user about it. We can perform the required clean-up operations and then exit using the `exit()` function from the `sys` module. The user will then make sure that the file is available with proper permission and will run the application again.

If you want, you can log all the useful debugging information provided in the traceback to some log file. A log file is a file that is used to record the events that occur during the execution of a program. So later, you can see what went wrong in the application, and you can let the user know about the problem in a simpler way. For server programs that run continuously without any user, it is important to write the error message and traceback information to a log file.

So, the option of graceful termination can include the release of resources, proper error reporting, and logging information.

Our next option could be to re-raise the exception. If you do not have enough information to fully handle the exception from where you are currently in the program, then you can send the exception to be handled in a higher context. We know that if we do not catch an exception, then automatically it will propagate. But sometimes, we may want to report the error or take some partial action, and then we want it to propagate up. This is called re-raising the exception. It is done using the `raise` statement that we will see in detail later in this chapter. Let us briefly understand what is meant by re-raising an exception. Suppose an error occurred and Python raised the exception; we caught it and, in the handler, did something and then again raised it; now, in the higher context, it can be again caught and handled. If it is not handled in a higher context, then it will remain an unhandled exception and will become the cause of abnormal termination.

We can also raise a different type of exception. First, Python will raise an exception, we will catch it, and, in the handler, we will probably do something and then raise an exception of another type, which will now propagate up and will have to be handled in the higher context. This is also done using the `raise` statement. So, we catch one type of exception and effectively change it to another type by raising a different type of exception.

So, we saw what we could do with the exception handlers. The `except` blocks can use any of these approaches or a combination of them to handle the error.

20.8 Guaranteed execution of finally block

Before seeing the details of the `finally` block, let us first understand why we need one. Suppose you have opened a database connection, and you are performing some operations on that database. After these operations, finally the connection has to be closed, which is very important otherwise it can lead to loss of data and resource leakage problems.

```
Open Connection
statement1
statement2
statement3
```


Close Connection

statement4

statement5

Statements 1, 2, and 3 represent the code in which we are working on the database. Suppose this code is suspicious, which means that it can raise some exceptions. So, we will protect it inside a `try` block and write `except` blocks to handle any exceptions that it might raise.

Open Connection

```
try:
```

```
    statement1
```

```
    statement2
```

```
    statement3
```

```
    Close Connection
```

```
except LookupError:
```

```
    statementX
```

```
    statementY
```

```
except ValueError:
```

```
    statementZ
```

```
statement4
```

```
statement5
```

The database connection must be closed, in any case, irrespective of any exceptions that occur while performing these operations. You want guaranteed execution of the Close Connection code, regardless of whether an exception occurs or not and whether it is handled or not. You want to ensure that it is executed in all the following three cases:

- when no exception occurs in the `try` block,
- when an exception occurs in the `try` block and is handled by one of the `except` clauses
- when an exception occurs but it does not match any of the `except` clauses following the `try` block and so is propagated up.

Let us see whether the Close Connection code will be executed in all these three cases, if we let it be there inside the `try` block.

If no exception occurs, the whole `try` block will be executed, and the Close Connection code will also be executed.

If an exception occurs in the `try` block, then the control leaves the `try` block and never comes there again, the remaining statements of the `try` block never execute. Thus, in case of any exception inside the `try` block, the Close Connection will not be executed.

So, now we write the Close Connection code inside all the `except` blocks:

```
Open Connection
try:
    statement1
    statement2
    statement3
    Close Connection
except LookupError:
    statementX
    statementY
    Close Connection
except ValueError:
    statementZ
    Close Connection
statement4
statement5
```

Now, the Close Connection will be executed when no exception is raised or if an exception is raised and handled here. But, if an exception is not handled here, then none of these `except` blocks will execute, the exception will be propagated up and the Close Connection will not be executed. So, this approach also doesn't guarantee execution of Close Connection in all situations.

Let us see what happens if we put the Close Connection code after the whole `try` statement.

```
OpenConnection
```

```
try:
```

```
    statement1
```

```
    statement2
```

```
    statement3
```

```
except LookupError:
```

```
    statementX
```

```
    statementY
```

```
except ValueError:
```

```
    statementZ
```

```
Close Connection
```

```
statement5
```

```
statement6
```

If there is no exception in the `try` block, then the Close Connection code will be executed. If an exception occurs and is handled here, then also the control will come to the Close Connection code, and it will be executed. If an exception occurs and is not handled here, then in that case exception propagates up, and the control jumps to a previously entered `try` block. So in that case, the Close Connection will not execute.

So, we could not find an appropriate place for the Close Connection code, which is the cleanup code that needs to be executed on the way out of the `try` block. The `finally` clause of the `try` statement gives us the solution for this problem.

```
OpenConnection
```

```
try:
```

```
    statement1
```

```
    statement2
```

```
    statement3
```

```
except LookupError:
```

```
        statementX
        statementY
except ValueError:
    statementZ
finally:
    Close Connection
statement4
statement5
```

The `finally` block is placed after all the `except` blocks and is always executed before leaving the `try` block. It is executed irrespective of whether an exception occurs or not and whether it is handled or not. So, the code in the `finally` block is executed in all the three situations that we saw.

When no exception occurs in the `try` block, the `finally` block will be executed after the execution of the `try` block. When an exception occurs in the `try` block and is handled by one of the `except` blocks, then the `finally` block is executed after executing the corresponding `except` block.

When an exception occurs but is not handled and is propagated up, then the `finally` block is executed before the exception propagation occurs. The `finally` block is also executed if the control leaves the `try` block because of a `break` or a `return` statement. So, the `finally` block is the best place for our `Close Connection` code.

When we have some external resources that were allocated by the program and were being used in the `try` block, and we want to ensure that they are properly released, then we can place the code for releasing them in the `finally` block, since it is the block which is guaranteed to be executed in any case. So, the `finally` block is used for placing any final processing code that needs to be executed under all circumstances. The final processing code could include closing files, closing database or network connections, logging out the user, releasing locks, or writing final log messages. The `finally` block ensures proper termination of any processes that are running.

You can place the `finally` clause after all the `except` blocks or you can have a `try` statement with only a `try` clause and a `finally` clause.

<pre>try: except LookupError: except ValueError: finally: </pre>	<pre>try: finally: </pre>

Figure 20.9: Two forms of try statement

The `try...finally` form is useful when you want your cleanup code to be run, even when you do not handle any exceptions that occur and let them propagate up. Any exceptions that occur in the `try` block will be propagated up since we do not have any `except` block in this form, but the cleanup code will run before the propagation of exception. The purpose of this form of `try` statement is not exception handling. It is there to ensure execution of the clean-up code.

For example, suppose you are reading from a file, and you want to ensure that the file is closed properly whether or not an exception occurred.

```
f = open('somefile.txt')
try:
    text = f.read()
finally:
    f.close()
```

If the `read` operation raises an exception, then it will be propagated up, however the `close` method of the file will be called before the control leaves the `try` block.

We have seen in the previous section that in case of some errors, it is not possible to continue the program, and in those cases, we aim for graceful termination of the program, which involves running all cleanup code before the program terminates. Now, after learning about the `finally` clause, we

can easily understand that this block plays a major role in our graceful termination planning. We can let the exception propagate up and not handle it anywhere, resulting in termination, but during the exception propagation, all the `finally` blocks on the way will be executed and this makes the termination graceful.

Now, let us see a simple program to verify that `finally` block is always executed on the way out from the `try` block.

```
try:
    x = int(input('Enter a number : '))
    print(10 / x)
except ValueError:
    print('ValueError exception handled')
finally:
    print('Running clean up code')
print('End .....')
```

Sample Run 1-

```
Enter a number : 3
3.3333333333333335
Running clean up code
End .....
```

Sample Run 2-

```
Enter a number : one
ValueError exception handled
Running clean up code
End .....
```

Sample Run 3-

```
Enter a number : 0
Running clean up code
Traceback (most recent call last):
```

```
File "C:\Users\deepali\test.py", line 3, in
<module>
```

```
    print(10/x)
```

```
ZeroDivisionError: division by zero
```

When we entered 3, no exception occurred in the try block, and the finally block was executed. When we entered one, ValueError exception occurred in try block, it was handled, and after execution of the except block, finally block was run. When we entered 0, ZeroDivisionError occurred which was not handled, the program terminated abnormally but the finally code was run. So, we can see that the finally block is run in all the cases.

Now, let us see what happens if the code in the except block raises an exception. In the except block, we have added a statement that will raise an exception.

```
try:
```

```
    x = int(input('Enter a number : '))
```

```
    print(10 / x)
```

```
except ValueError:
```

```
    print('ValueError exception handled')
```

```
    print(3 + 'x')
```

```
finally:
```

```
    print('Running clean up code')
```

```
print('End .....')
```

Sample Run-

```
Enter a number : one
```

```
ValueError exception handled
```

```
Running clean up code
```

```
Traceback (most recent call last):
```

```
File "C:\Users\deepali\test.py", line 2, in
<module>
```

```
    x = int(input('Enter a number : '))
```

```
ValueError: invalid literal for int() with base 10:  
'one'
```

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

```
File "C:\Users\deepali\test.py", line 6, in  
<module>
```

```
    print(3+'x')
```

```
TypeError: unsupported operand type(s) for +: 'int'  
and 'str'
```

We entered one, ValueError occurred in the try block, the except block was executed, and while executing the except block, a TypeError occurred, but the finally code was executed.

The finally block will be executed even if we exit the program using the exit function from the sys module.

```
import sys
```

```
try:
```

```
    x = int(input('Enter a number : '))
```

```
    print(10 / x)
```

```
except ValueError:
```

```
    print('ValueError exception handled')
```

```
    sys.exit()
```

```
finally:
```

```
    print('Running clean up code')
```

```
print('End .....')
```

Sample Run-

```
Enter a number : one
```

```
ValueError exception handled
```

```
Running clean up code
```

We entered one, ValueError occurred in the try block, and the except block was executed in which the exit function is called and the

program is exited, but before that, the `finally` code was executed.

We have seen before that we can use the `finally` block even if there is no `except` block. In the following code, we have only a `finally` block following the `try` block.

```
try:
    x = int(input('Enter a number : '))
    print(10 / x)
finally:
    print('Running clean up code')
print('End .....')
```

Sample Run-

```
Enter a number : one
```

```
Running clean up code
```

```
Traceback (most recent call last):
```

```
  File "C:\Users\deepali\test.py", line 2, in
<module>
```

```
    x = int(input('Enter a number : '))
```

```
ValueError: invalid literal for int() with base 10:
'one'
```

The `ValueError` exception that was raised was not handled, but the `finally` code was executed.

20.9 else Block

The `try` statement can have an optional `else` clause. The `else` block should be placed after all the `except` blocks. It executes only when the `try` block terminates normally. It will not be executed if any exception is raised in `try` block or if the `try` block terminates due to a `break`, `continue` or `return` statement. If `finally` block is present in the `try` statement, then the `else` block should be placed before the `finally` block.

<pre> try: except ExceptionA: except ExceptionB: else: </pre>	<pre> try: except ExceptionA: except ExceptionB: else: finally: </pre>

Figure 20.10: *try statements with else blocks*

If you want to write the `else` clause in your `try` statement, then there should be at least one `except` block present. You cannot use it in the `try...finally` form we saw in the previous section. The following `try` statement will give syntax error because there is no `except` block, and we have written an `else` clause.

```

try:
    .....
else:
    .....
finally:
    .....

```

The `else` block is executed only if no exception occurs during the execution of the `try` block. Let us see the control flow in different situations:

<code>try:</code>	- No exception raised in try block
<code>statement1</code>	1,2,3, 4,5, C,D, 6,7
<code>statement2</code>	
<code>statement3</code>	
<code>except LookupError:</code>	- Exception raised and is handled here
<code>statementX</code>	(ValueError at statement2)
<code>statementY</code>	1, A,B, C,D, 6,7
<code>except ValueError:</code>	
<code>statementA</code>	
<code>statementB</code>	- Exception raised but is not handled here
<code>else:</code>	(ArithmeticError at statement2)
<code>statement4</code>	1, C,D
<code>statement5</code>	
<code>finally:</code>	
<code>statementC</code>	
<code>statementD</code>	
<code>statement6</code>	
<code>statement7</code>	

Figure 20.11: Control flow in a try statement with else block

When no exception occurs in the `try` block, the three statements inside the `try` block are executed, then the `else` block executes, and then the `finally` block executes.

The second situation could be when an exception occurs in the `try` block and is handled in one of the `except` blocks. For example, suppose `ValueError` is raised at `statement2`. In this case, first `statement1` executes, then the `try` block is suspended due to `ValueError` at `statement2`, and the corresponding `except` block executes. The `else` block will not execute in this case as it executes only when there is no exception in the `try` block. After the `except` block, the `finally` block executes, and then control comes out of the `try` statement, and the statements 6 and 7 are executed.

The third situation could be when an exception is raised in the `try` block but is not handled here. For example, suppose an `ArithmeticError` occurs at `statement2`. In this case, first the `statement1` executes, and then the `try` block is terminated due to an exception at `statement2`. `ArithmeticError` cannot be handled here so it will be propagated up. The `else` block will not be executed, the `finally` block executes before propagation of exception and the control in this case will not reach to `statement 6`, it will be transferred to the previously entered `try` block.

So, we saw that the `else` block executes only in the case when no exception is raised in `try` block, if an exception is raised whether handled or not, the `else` block does not execute.

Let us see a simple program to understand this `else` block.

```
schools = [('XYZ', 1, 2), ('PQR', 9, 8), ('ABC', 9, 0), ('LMN', 8, 7)]

for school_name, boys, girls in schools:
    ratio = boys / girls
    print(f'Ratio of boys to girls for school {school_name} is {ratio}')
    if ratio > 1:
        print('Boys in majority\n')
    else:
        print('Girls in majority\n')
```

We have a list of tuples, where each tuple contains the school's name, number of boys and number of girls in the school. In the `for` loop, we find the ratio, print it and use it in the `if` statement. Here is the output of this program:

```
Ratio of boys to girls for school XYZ is 0.5
Girls in majority
Ratio of boys to girls for school PQR is 1.125
Boys in majority
Traceback (most recent call last):
  File "C:\Users\deepali\test.py", line 4, in
<module>
    ratio = boys/girls
ZeroDivisionError: division by zero
```

There was no problem in the first two schools. For the third school, the number of girls is zero, so `ZeroDivisionError` occurred, and the program was terminated. The statement where we are finding the ratio is the suspicious code that can raise exception, so let us give it the protection of `try` block and write an `except` block .

```
schools = [('XYZ', 1,2), ('PQR', 9,8), ('ABC',
9,0), ('LMN', 8,7)]

for school_name, boys, girls in schools:
    try:
        ratio = boys/girls
    except:
        print(f'Ratio not defined for school
{school_name}\n')

        print(f'Ratio of boys to girls for school
{school_name} is {ratio}')
        if ratio > 1:
            print('Boys in majority\n')
        else:
            print('Girls in majority\n')
```

Output-

```
Ratio of boys to girls for school XYZ is 0.5
Girls in majority
Ratio of boys to girls for school PQR is 1.125
Boys in majority
Ratio not defined for school ABC
Ratio of boys to girls for school ABC is 1.125
Boys in majority
```

```
Ratio of boys to girls for school LMN is  
1.1428571428571428
```

```
Boys in majority
```

For the first two schools and the last school, the output is correct. For the third school, it is showing the ratio not defined, but after that, it is printing the two statements showing the ratio and majority for that school, which is wrong. Let us see what is happening. For the first two schools, the ratio is successfully calculated, and no exception occurs, so the `except` block is not executed. For the third school, `ZeroDivisionError` occurs, the ratio is not assigned any value, its value just continues to be what it was in the last iteration.

The `except` block is executed, so we get the message printed, and then the last two statements are also executed which actually use the value of ratio from the previous iteration. In the output, we can see that the ratio of school PQR is actually printed for the school ABC. In the fourth iteration, for the school LMN, the output is correct.

Our requirement is that we want the last two statements to be executed only when the `try` block is successful. If any exception occurs, we do not want these statements to be executed. So now we put them inside the `else` block. The `else` block will be executed only when there is no exception in the `try` block.

```
schools = [('XYZ', 1,2), ('PQR', 9,8), ('ABC',  
9,0), ('LMN', 8,7)]
```

```
for school_name, boys, girls in schools:
```

```
    try:
```

```
        ratio = boys/girls
```

```
    except:
```

```
        print(f'Ratio not defined for school  
{school_name}\n')
```

```
    else:
```

```

        print(f'Ratio of boys to girls for school
{school_name} is {ratio}')
    if ratio > 1:
        print('Boys in majority\n')
    else:
        print('Girls in majority\n')

```

Output-

```

Ratio of boys to girls for school XYZ is 0.5
Girls in majority
Ratio of boys to girls for school PQR is 1.125
Boys in majority
Ratio not defined for school ABC
Ratio of boys to girls for school LMN is
1.1428571428571428
Boys in majority

```

Now we get the desired output. Our problem could be solved even if we had put the statements inside the try block.

```

schools = [('XYZ', 1, 2), ('PQR', 9, 8), ('ABC', 9,
0), ('LMN', 8, 7)]
for school_name, boys, girls in schools:
    try:
        ratio = boys / girls
        print(f'Ratio of boys to girls for school
{school_name} is {ratio}')
        if ratio > 1:
            print('Boys in majority\n')
        else:
            print('Girls in majority\n')
    except:

```

```
print(f'Ratio not defined for school
{school_name}\n')
```

This also gives us the desired output because when there is an exception, the control leaves the `try` block and the rest of the `try` block is not executed. So, then what is the need of the `else` block. In the next section, we will try to understand why it is not a good idea to put this code inside the `try` block and why do we need an `else` block. You can explore the next section if you are curious to know about the need of `else` block, otherwise you can skip it.

20.10 Why do we need an else block

In the first example of the previous section, we had seen that the two statements that are there in the `else` block (statements 4 and 5) will execute only when no exception occurs in the `try` block. Let us delete the `else` block from that example and put the two statements (statement 4 and 5) inside the `try` block.

<code>try:</code>	
<code>statement1</code>	- No exception raised in try block
<code>statement2</code>	1,2,3, 4,5, C,D, 6,7
<code>statement3</code>	
<code>statement4</code>	- Exception raised and is handled here
<code>statement5</code>	(ValueError at statement2)
<code>except LookupError:</code>	1, A,B, C,D, 6,7
<code>statementX</code>	
<code>statementY</code>	
<code>except ValueError:</code>	- Exception raised but is not handled here
<code>statementA</code>	(ArithmeticError at statement2)
<code>statementB</code>	1, C,D
<code>finally:</code>	
<code>statementC</code>	
<code>statementD</code>	
 <code>statement6</code>	
<code>statement7</code>	

Figure 20.12: Control flow in a try statement without the else block

When there is no exception, the first 5 statements will be executed, then the two statements of `finally` block, and then statements 6 and 7 will be

executed. When a `ValueError` occurs in the `try` block at `statement2`, `statement 1` executes, then `statements A and B` of `except` block, and then `statements C and D` of `finally` block, and then `statements 6 and 7` are executed. When an exception occurs at `statement2`, and is not handled here, `statement 1` executes, then `statements C and D` are executed.

If we compare this with the example in the previous section, we can see that the same statements are executed in all three cases. Exactly the same code is being executed in all three cases, whether we put the `statements 4 and 5` inside the `else` block or inside the `try` block. So, then, what is the advantage of putting this code inside the `else` block?

Code Sample 1

```
try:
    statement1
    statement2
    statement3
except LookupError:
    statementX
    statementY
except ValueError:
    statementA
    statementB
else:
    statement4
    statement5
finally:
    statementC
    statementD
statement6
statement7
```

Code Sample 2

```
try:
    statement1
    statement2
    statement3
    statement4
    statement5
except LookupError:
    statementX
    statementY
except ValueError:
    statementA
    statementB
finally:
    statementC
    statementD
statement6
statement7
```

Figure 20.13: try statements with and without the else block

There is actually a difference between these two approaches, and it is due to the fact that the `except` blocks of the `try` statement do not handle any exceptions that are raised in the `else` block.

In sample 1, if `statements 4 and 5` raise an exception of type `LookupError` or `ValueError`, then it will not be handled by the `except` blocks of this `try` statement.

In sample 2, if `statements 4 and 5` raise an exception of type `LookupError` or `ValueError`, then it will be handled by the `except` blocks of this

`try` statement. This is a problem; we wanted to protect only statements 1, 2, 3 from `LookupError` and `ValueError`, and we had written the code in `except` blocks accordingly. But if somehow statement 4 or 5 raises `LookupError` or `ValueError`, then it will also be handled here, which we do not want.

So, the `else` block is useful in avoiding any accidental handling of unexpected exceptions. In sample 1, if statements 4 or 5 raise a `ValueError` or `LookupError` then it will be propagated up. It will not be handled here.

The second benefit of writing an `else` block is that it increases readability. In sample 1, we have visually separated the code that needs protection and the code that will be executed on the success of this code. Your `else` block makes it clear which code you want to protect and which not, which code's exceptions you want to handle here, and which code's exceptions you want to be propagated up. It is clearly visible that any exceptions in the code present inside `else` block will be propagated up.

In sample 1, the `except` blocks are closer to the protected code, so this makes it clearer what is being protected and how is the problem being handled. It is a good idea to keep the code that can cause exceptions near the code that handles those exceptions. The code in the `try` block is reduced and it definitely improves readability.

Another place where someone can think of putting statements 4 and 5 is outside the whole `try` statement. The following figure shows the comparison of control flow when these statements are placed inside the `else` block and when they are placed outside the `try` statement.

Code Sample 1

```

try:
    statement1
    statement2
    statement3
except LookUpError:
    statementX
    statementY
except ValueError:
    statementA
    statementB
else:
    statement4
    statement5
finally:
    statementC
    statementD
statement6
statement7

```

- No exception raised in try block
1,2,3, 4,5, C,D, 6,7

- Exception raised and is handled here
(ValueError at st2)
1, A,B, C,D, 6,7

- Exception raised but is not handled here
(ArithmeticError at st2)
1, C,D

Code Sample 3

```

try:
    statement1
    statement2
    statement3
except LookUpError:
    statementX
    statementY
except ValueError:
    statementA
    statementB
finally:
    statementC
    statementD
statement4
statement5
statement6
statement7

```

- No exception raised in try block
1,2,3, C,D, 4,5 6,7

- Exception raised and is handled here
(ValueError at st2)
1, A,B, C,D, 4,5, 6,7

- Exception raised but is not handled here
(ArithmeticError at st2)
1, C,D

Figure 20.14: Control flow in try statements with and without the else blocks

Let us discuss the control flow in all the three cases in code sample 3.

When there is no exception in the `try` block, the statements 1, 2, and 3 execute, then the `finally` block and then the statements 4, 5, 6, and 7 execute. Now, here, note that the `finally` block will be executed before statements 4 and 5, so if these statements are using some resources that were cleaned up in the `finally` block, then there can be a problem. Suppose we had established a database connection, and all the statements 1,2,3,4,5 use that connection, and in the `finally` block the connection is closed. Then, you cannot put the statements 4 and 5 outside `try`; you have to execute them before the `finally` block executes. We have to put them either inside the `try` block or inside the `else` block and we have already seen that putting inside `try` block is not a good idea so we are left with option of `else` block.

Now let us see what happens when an exception is raised and handled here. Suppose a `ValueError` exception occurs at statement 2; first statement1 executes, then statements A and B of `except` block, then statements C and D of `finally` block and then the statements 4,5,6,7.

If `ArithmeticError` is raised at statement 2 then, statement1 executes, `finally` block executes and then the exception propagates up and so the statements 4 and 5 are not executed.

When we compare the two approaches, we can clearly see the difference. In code sample 1, statements 4 and 5 will be executed only in the case when no exception occurs in `try` block.

In code sample 3, statements 4 and 5 will be executed when no exception occurs in `try` and also when exception occurs and is handled here.

Suppose statements 4 and 5 are something that we want to execute only when the `try` block is successful, maybe we are initializing some variable in `try` block and then using it in these statements. So, in case the `try` block does not succeed, then that variable will not be initialized, and we will get an error while executing statements 4 and 5. We have seen this in our example program where we were assigning to `ratio`. To avoid this problem, you will have to use a flag. The flag will help you put a check so that you execute the two statements only when the `try` block succeeds.

All this work you would have to do if Python had not given you the facility of `else` clause. But since we have an `else` clause, we do not need to do all this, placing the code in the `else` block ensures that these statements will be executed only on success of `try` block. If you remember the `else` block of loops, you will realize this that the `else` block of `try` statement serves the same purpose as the `else` block in a loop.

This whole detailed explanation was there to make you understand why we need an `else` block.

Now, we have seen all the clauses of the `try` statement, let us quickly review them once.

```
try:
    Code that can raise exceptions
except Exception1:
    Code that is to be executed in case of
    Exception1
except Exception2:
    Code that is to be executed in case of
    Exception2
else:
```

Code that is to be executed when no exception occurs in `try` block

`finally:`

Code that needs to be executed in any case

The order of these clauses is important, first we have the `except` blocks, then the `else` block, and then the `finally` block. In the `try` block, we place the mainstream code that is likely to raise exceptions, then we have the `except` clauses, which catch and handle the raised exceptions. The `else` block is executed when there is no exception in the `try` block and the `finally` block is always executed whether there is any exception or not and whether it is handled or not.

20.11 How to get exception details

We know that when an exception is raised, an exception object is created, which contains information associated with the exception. Usually, it contains an error message, but it might contain other information. This information can be useful in handling the exception. To retrieve this information, you will need a reference to the exception object. You can get access to this object by using the `as` keyword in the `except` clause.

In the following `try` statement, we have used the `as` keyword with the identifier `e` in the first `except` clause. The identifier `e` will be a reference to the exception object that would be created when `LookupError` exception occurs in the `try` block. By using this reference, you can access any of the attributes or methods of the raised exception object.

`try:`

`statement1`

`statement2`

`statement3`

`except LookupError as e:`

`statementX`

`statementY`

```
except ValueError:
    statementA
    statementB
else:
    statement4
    statement5
finally:
    statementC
    statementD
```

Let us see an example.

```
numbers = [23, 45, 67]
try:
    print(numbers[3])
except LookupError as e:
    print(type(e))
```

Output-

```
<class 'IndexError'>
```

The statement inside the `try` block will raise an `IndexError` since we are trying to access an out of bound index. In the `except` clause, we have written `LookupError`, which is a parent of `IndexError`, so it will catch `IndexError` also. We have also specified the `as` keyword followed by the identifier `e`. Inside the `except` block, we are printing the type of exception that is caught. The identifier `e` refers to the exception object that was created when the exception occurred, and the type of the exception object is `IndexError` which is printed.

Exception objects can have different attributes attached to them; the number and type of attributes, depends on the type of the exception. All exception classes are derived from `BaseException`, and the `BaseException` type provides some basic attributes and methods that can be used by all the

inherited classes. So, all exception classes have some common functionality provided by the `BaseException` class. The `BaseException` has an attribute named `args`, so all exceptions have the `args` attribute.

When Python creates a new exception instance, it passes some arguments to the exception constructor, these arguments indicate information related to the error. The argument values that it provides to the constructor are stored in the attribute `args` which is actually a tuple. So, every exception instance has an attribute `args` which is a tuple, and it contains arguments that are passed when the exception object is created. Let us create some exception objects (instances of exception classes) and see the `args` attributes for them.

```
>>> e = Exception('msg', 1, 2)
>>> ex = LookupError(2, 4, 'A', 'B')
>>> exc = ZeroDivisionError('XX')
>>> excp = IndexError()
>>> e.args
('msg', 1, 2)
>>> ex.args
(2, 4, 'A', 'B')
>>> exc.args
('XX',)
>>> excp.args
()
```

From these examples, we can see that the arguments that are sent while creating the exception instance are stored in the exception object's `args` tuple attribute.

So, we can see that every exception instance has the `args` attribute which we can access in the `except` block if we get a reference to the instance using the `as` keyword. Another thing that is inherited by all classes is the string representation. String representation of an exception is provided in the

BaseException class, so all other exception classes inherit it. When `str()` function is called on an instance of any exception class, we get the string representation.

```
>>> str(e)
>('msg', 1, 2)
>>> str(ex)
(2, 4, 'A', 'B')
>>> str(exc)
'XX'
>>> str(excp)
''
```

We can see that the string representation is nothing but the tuple of arguments. So, when the instance is printed, the `args` tuple is automatically printed. If there is only a single argument, then it displays as a single value, not as a tuple. If no arguments were passed during creation of the instance, then we get an empty string. When you print the exception object using the `print` function, then also the string representation is used.

```
>>> print(e)
('msg', 1, 2)
```

Most of the exception objects have only an error message as the argument, this error message is printed when we print the object. This all works because the `BaseException` class defines the `__str__()` method, and so the arguments can be printed directly without referencing the `args` attribute. Similarly, the `repr` function is also available for any exception class.

```
>>> print(repr(e))
Exception('msg', 1, 2)
>>> e
Exception('msg', 1, 2)
```


In both these cases, the `repr` function is called.

Now, let us print the `args` tuple and the string representation in the example that we have seen before.

```
numbers = [23, 45, 67]

try:
    print(numbers[3])
except LookupError as e:
    print(type(e))
    print(e.args)
    print(e)
```

Output-

```
<class 'IndexError'>
('list index out of range',)
list index out of range
```

In the `except` block, we are printing type of the exception object, the `args` attribute of the exception object and the exception object itself. In the output, we can see a tuple for the `args` attribute. It contains only one element, which is the string that represents the error message. When Python would have created this instance, it must have passed this argument and so we have this string in the `args` tuple. When we print the exception object, this message is printed because the `__str__` method is called.

Most of the exceptions are called with a single string argument representing the error message, but some exceptions like `OSError` and `SyntaxError` have other arguments also which provide more details that can be used while fixing the error.

In the following example, we are trying to open a non-existent file, so the `FileNotFoundError` exception will occur, which is derived from `OSError` so it will be caught by the `except` clause.

```
try:
```

```

        file = open('ff')
except OSError as err:
    print(type(err))
    print(err.args)

```

Output-

```

<class 'FileNotFoundError'>
(2, 'No such file or directory')

```

We get access to 2 arguments; one is the error number and other is a string that contains error information. Another exception that uses multiple arguments is `SyntaxError`.

```

try:
    import test
except SyntaxError as e:
    print(e.args)

```

Output-

```

("expected ':'", (' C:\\Users\\deepali\\test.py',
4, 16, 'if first > last\n', 4, 16))

```

We have imported a file in the `try` block and since there is some wrong syntax in this file, the `SyntaxError` exception is raised. In the output, we can see that the `args` tuple here contains multiple items. By looking at the values, we are not sure about what they denote. We can use the `dir` function to see all the attributes.

```

>>> dir(SyntaxError)
['__cause__', '__class__', '__context__',
 '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__',
 '__getattr__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__',

```

```
'__setattr__', '__setstate__', '__sizeof__',
'__str__', '__subclasshook__',
'__suppress_context__', '__traceback__',
'add_note', 'args', 'end_lineno', 'end_offset',
'filename', 'lineno', 'msg', 'offset',
'print_file_and_line', 'text', 'with_traceback']
```

We can see these attributes, now the question is how to get the names of the attributes whose values are shown in the tuple. For that we can write a for loop. In this loop, we iterate over the list of attributes, and if the attribute does not start with an underscore, we print the attribute and its value.

```
try:
    import test
except SyntaxError as e:
    print(e.args)
    for attr in dir(e):
        if not attr.startswith('_'):
            print(attr, "=",
e.__getattribute__(attr))
```

Output-

```
("expected ':'", ('C:\\Users\\deepali\\test.py', 4,
16, 'if first > last\\n', 4, 16))
add_note = <built-in method add_note of SyntaxError
object at 0x000001A1F6B1A480>
args = ("expected ':'",
('C:\\Users\\deepali\\test.py', 4, 16, 'if first >
last\\n', 4, 16))
end_lineno = 4
end_offset = 16
filename =
E:\\Deepali\\BOOK_Python\\Programs\\20_ExceptionHandlin
g\\test.py
```

```

lineno = 4
msg = expected ':'
offset = 16
print_file_and_line = None
text = if first > last
with_traceback = <built-in method with_traceback of
SyntaxError object at 0x000001A1F6B1A480>

```

We can see the attributes with the values here, attributes that have None as the value are not a part of the args tuple.

Let us write the same loop for the previous example.

```

try:
    file = open('datafile.txt')
except OSError as err:
    for entry in dir(err):
        if not entry.startswith('_'):
            print(entry, '=',
err.__getattr__(entry))

```

Output-

```

add_note = <built-in method add_note of
FileNotFoundError object at 0x0000027F19C5A950>
args = (2, 'No such file or directory')
Traceback (most recent call last):
  File "C:\Users\deepali\myprogram.py", line 2, in
<module>
    file = open('datafile.txt')
FileNotFoundError: [Errno 2] No such file or
directory: 'datafile.txt'

```

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

```
File "C:\Users\deepali\myprogram.py", line 7, in
<module>
```

```
    print(entry, "=", err.__getattribute__(entry))
```

AttributeError: characters_written

We got two attributes printed, and then we got the message ‘During handling of the above exception, another exception occurred’. The

AttributeError occurred because some attributes are not accessible.

We can handle this exception by ignoring it. We will enclose the error-prone statement inside the try block, and in the except block, we will simply write pass to ignore the exception.

```
try:
```

```
    file = open('datafile.txt')
```

```
except OSError as err:
```

```
    for entry in dir(err):
```

```
        if not entry.startswith('_):
```

```
            try:
```

```
                print(entry, '=',
```

```
err.__getattribute__(entry))
```

```
            except AttributeError:
```

```
                pass
```

Output-

```
add_note = <built-in method add_note of
FileNotFoundError object at 0x0000023721F2A950>
```

```
args = (2, 'No such file or directory')
```

```
errno = 2
```

```
filename = datafile.txt
filename2 = None
strerror = No such file or directory
winerror = None

with_traceback = <built-in method with_traceback of
FileNotFoundError object at 0x0000023721F2A950>
```

So, we have seen that when Python raises an exception, it can associate some values with it. These values are in the form of arguments that are passed while constructing the exception object. By using the `as` keyword and an identifier in the `except` block, we get a reference to the exception object. Then we can access different attributes of the object using that reference.

When you write the bare `except` clause, you cannot use the `as` keyword, so in that case, you can see the details by using the `exc_info()` function of the `sys` module. It will return information about the current exception that has been raised.

```
import sys

try:
    f = open('xyz')
except:
    exc_type, exc_value, exc_traceback =
sys.exc_info()
    print(exc_type)
    print(exc_value)
    print(exc_traceback)
```

Output-

```
<class 'FileNotFoundError'>
[Errno 2] No such file or directory: 'xyz'
<traceback object at 0x000001B3E66E3B40>
```

The function `sys.exc_info()` returns a tuple that contains the exception type, the exception value, and a traceback object that tells the exception propagation path. In the program, we unpack the tuple and then print these values.

In the `BaseException` class, there is a `__traceback__` attribute, this attribute gives us a traceback object. It is the same object returned as the third item of `sys.exc_info()`.

```
>>> dir(BaseException)
[ ..... ' __traceback__ ', .....]
```

There is a `traceback` module that uses this traceback object. We can use this module to visualise the traceback information. It can also be useful for logging purposes. You can import this module and use `help` on it to get more information.

20.12 Nested try statements

Nesting of a `try` statement can happen in two ways:

- (i) when we write a `try` statement inside the `try` block of another `try` statement.
- (ii) when we have a function call inside a `try` block and inside that function's definition, there is another `try` statement.

We will see both of these in detail. First, let us see the first one, which has a `try` statement physically embedded inside another `try` block.

```
try:
    .....
    try:
        .....
    except LookupError:
        .....
```

```
except ValueError:
```

```
.....
```

```
else:
```

```
.....
```

```
finally:
```

```
.....
```

Here, we have a `try` statement inside the `try` block of another `try` statement; we have only 2 levels, but there can be more levels also. If an exception occurs inside the inner `try` block, and if it does not match any of the `except` blocks listed in the inner `try` statement, then the `except` blocks of the outer `try` statement are tried for a match. So, Python first tries to find a matching `except` clause in the inner nested levels and then it moves on to the outer `try` levels. Let us see an example of this:

```
try:
```

```
    try:
```

```
        x = int(input('Enter a number : '))
```

```
        print(10 / x)
```

```
    except ValueError as e:
```

```
        print('ValueError exception occurred : ', e)
```

```
except ZeroDivisionError:
```

```
    print('Cannot divide by zero')
```

Output-

```
Enter a number : 0
```

```
Cannot divide by zero
```

The `except` clause of the inner `try` statement catches `ValueError` and the `except` clause of the outer `try` statement catches `ZeroDivisionError`. When we executed the program, we entered 0, so a `ZeroDivisionError` occurred. No matching `except` clause was found

in the inner `try` statement, so Python tries to find a match in the outer level of `try` and finds a match, and thus, the `except` clause of outer `try` gets executed.

Now, suppose in the `except` clause of inner `try` statement, we write `Exception` instead of `ValueError`.

```
try:
    try:
        x = int(input('Enter a number : '))
        print(10 / x)
    except Exception as e:
        print('An exception occurred : ', e)
except ZeroDivisionError as e:
    print(e)
```

Output-

```
Enter a number : 0
```

```
An exception occurred :  division by zero
```

The `ZeroDivisionError` occurs, but now the exception is caught at the inner level only because we have written the broad exception handler there. The exception dies after it has been handled in the inner `try`, and it does not propagate out of the inner `try`. So even though the outer `try` has a more specific `except` clause, the exception is caught by the inner `try` only. Here is another example:

```
try:
    file = open('data.txt')
    try:
        x = file.readlines()
        print(x)
    except Exception as e:
```

```

        print(type(e))
        print(e)
    finally:
        file.close()
except FileNotFoundError as e:
    print(e)

```

In the `try` block of the outer `try` statement, first, we open the file, and then we have another `try` statement. If the file is not found, then `FileNotFoundError` exception will occur while opening the file, and the `except` clause of the outer `try` statement will execute.

If the file opens successfully, then the inner `try` statement will execute. In the `try` block of this statement, we are reading the file and if an exception occurs during reading of the file, it will be caught and handled and the `finally` block will close the file.

We saw two examples of the nesting that happens when we write a `try` statement explicitly inside a `try` block. In the start of the section, we had mentioned that another way nesting can happen is when we use different functions in our program. Now, let us `try` to understand this one.

First, let us see what happens when we call a function within a `try` block and an exception occurs inside the function call.

```

def func():
    .....
    .....
try:
    .....
    func()
except ValueError:
    .....

```

We have a function call inside the `try` block and while executing the function, suppose an exception is raised. The code that is inside the function definition is not enclosed in a `try` block, so Python goes to the previously entered `try` block and looks for a matching `except` clause. If a match occurs there, then the exception is caught and handled.

Now, suppose there is another function `g` that is called inside the function `func`.

```
def g():
    .....
    .....
def func():
    .....
    g()
try:
    .....
    func()
except ValueError:
    .....
```

If any exception occurs while executing the code of function `g`, then that exception also can be caught and handled by the `except` blocks of the `try` statement if a match occurs. So, the `try` statement actually lets us catch an exception that is raised from within the `try` block or from any code called from any depth within the `try` block.

Now suppose inside the function `func` also, we have a `try...except` statement.

```
def g():
    .....
    .....
def func():
```

```

try:
    .....
    g()
except TypeError:
    .....

try:
    .....
    func()
except ValueError:
    .....

```

Now, effectively we have nested `try` statements. You can think of the `try` statement that is inside `func` as the inner `try` and the other one as the outer `try`. So, if an exception occurs inside the inner `try` block while executing the code inside it or while executing the function `g`, then Python will try to find a match in the `except` clauses listed there, if it does not find any match then it will go to the outer `try` to find a match.

Now, suppose inside the function `g()` also, we have a `try` statement.

```

def g():
    .....
    try:
        .....
    except ValueError:
        .....

def func():
    try:
        .....
        g()

```

```

    except TypeError:
        .....

try:
    .....
    func()
except ValueError:
    .....

```

Now if an exception occurs inside the `try` block of function `g()`, then first the `except` clause associated with this `try` block is tried. If a match is not found there then the control goes to the `try` statement inside `func` and tries to find a match in its `except` clause. If a match is not found here also, then the `except` clause of the `try` statement that is at the module level is tried. If a match is not found here also then the program terminates abnormally. This is what we call exception propagation. The control always enters the previously entered `try` block.

When an exception occurs, Python comes to the most recently entered `try` statement and tries to find a match. If it finds a match, then the corresponding `except` clause is executed, and execution resumes after the `try` statement; otherwise, it tries the previously entered `try` statements that have been entered but not yet exited. So, when there is a function call inside a `try` block, and the function's code has a `try` statement, then we effectively have nested `try` statements.

So, in this section we saw that exceptions can be raised from a deeply nested `try` block or from a nested function call and can be caught at higher levels.

20.13 Raising Exception

Till now, we have seen what happens when Python raises an exception and how to respond to exceptions raised by Python. So, basically, thus far the work of raising exceptions was done by Python, while we were just reacting to exceptions. Sometimes in your application, there will be situations when you want some exceptions to occur. For example, you would like an

exception to be raised when a certain condition is True. Or sometimes when you are inside a handler, you can only partially handle an exception, and you want to pass it to the higher level to be handled. In these sorts of cases, you can use the `raise` statement.

`raise` statement can be used to raise exceptions manually, which means that your code can explicitly raise exceptions. Before seeing the full syntax of the `raise` statement, let us see a simple example that uses `raise` statement.

```
age = int(input('Enter age : '))
if age < 0 or age > 120:
    raise ValueError
```

We have written the `raise` statement that consists of the `raise` keyword and the name of the exception. A `ValueError` exception will be raised when the value of `age` is less than 0 or greater than 120. Here are some sample runs of this code.

Sample run 1 -

```
Enter age : 22
```

Sample run 2 -

```
Enter age : 500
```

```
Traceback (most recent call last):
```

```
  File "C:\Users\deepali\test.py", line 3, in
<module>
```

```
    raise ValueError
```

```
ValueError
```

Sample run 3 -

```
Enter age : -5
```

```
Traceback (most recent call last):
```

```
File "C:\Users\deepali\test.py", line 3, in
<module>
```

```
        raise ValueError
ValueError
```

When we entered 22, there was no problem but when 500 or -5 was entered, a `ValueError` exception was raised. Whenever the `if` condition will be `True`, a `ValueError` exception will be raised. The program abnormally terminates in the last 2 runs, since an exception is being raised and is not handled.

The effect of raising an exception using `raise` statement, is the same as that of any exception raised by Python. The normal flow stops and search for a matching `except` clause starts in the enclosing `try` statements. The exceptions raised by the `raise` statement can be caught and handled in the same way as exceptions raised by Python are caught.

Now let us enclose the previous code inside a `try` block.

```
try:
    age = int(input('Enter age : '))
    if age < 0 or age > 120:
        raise ValueError
except ValueError:
    print('Invalid value for age')
else:
    print('Age is', age)
```

In the `except` block, we print an error message and in the `else` block we print the value of age.

Sample run 1 -

```
Enter age : 22
Age is 22
```

Sample run 2 -

```
Enter age : 500
```

Invalid value for age

Sample run 3 -

Enter age : five

Invalid value for age

When we entered 22, there was no exception so the `else` block was executed and value of `age` was printed. In the last 2 runs, a `ValueError` exception occurred so the `except` block was executed. In the second case the `ValueError` was raised by our code because the `if` condition was not `True`. In the last run, the `ValueError` was raised by the `int` function since the value 'five' is not a valid integer value.

Both kinds of `ValueErrors` are detected by the same `except` block, so there is no way to know why the value of `age` was invalid. It could be invalid because the input was not an integer or it could be invalid because the integer value was not within the valid range. Let us see what we can do to differentiate the two types of `ValueErrors`.

While raising the exception we can also send some arguments, so let us send a string argument when we are raising `ValueError` exception. In the `except` clause, we use the `as` keyword to get a reference to the exception object and inside the `except` block we print the exception object. All the arguments that are sent while creating the exception, go to the `args` tuple, and printing the exception prints the `args` tuple.

```
try:
```

```
    age = int(input('Enter age : '))
```

```
    if age < 0 or age > 120:
```

```
        raise ValueError('Age not in valid range')
```

```
except ValueError as e:
```

```
    print('Invalid value for age')
```

```
    print(e)
```

```
else:
```

```
    print('Age is', age)
```


Sample run 1 -

Enter age : 22

Age is 22

Sample run 2 -

Enter age : 500

Invalid value for age

Age not in valid range

Sample run 3 -

Enter age : five

Invalid value for age

invalid literal for int() with base 10: 'five'

Now we can see the cause of the ValueError in the message. We can send multiple arguments also, for example suppose we have two variables named `minimum` and `maximum` that denote the minimum and maximum valid values for age. To make our error message more informative, we can send these values also as arguments while raising the exception.

```
minimum = 18
```

```
maximum = 60
```

```
try:
```

```
    age = int(input('Enter age : '))
```

```
    if age < minimum or age > maximum:
```

```
        raise ValueError(f'Age not in valid range  
{minimum}-{maximum} ')
```

```
except ValueError as e:
```

```
    print('Invalid age value')
```

```
    print(e)
```

```
else:
```

```
print(age)
```

Sample Run-

```
Enter age : 500
```

```
Invalid age value
```

```
Age not in valid range 18-60
```

Now let us see the syntax of the `raise` statement.

```
raise exceptionClass(argument1, argument2, ..... )
```

We have the `raise` keyword followed by the name of an exception class and then inside parentheses we can have an optional list of arguments separated by commas. When this `raise` statement is written, an exception instance of the specified exception type is created and the arguments are sent to the initializer. You can send any number of arguments when you raise an exception. To access these arguments inside the handler, you can use the `args` attribute of the exception object. The arguments that we provide here give information about the error, they clarify the reason of the error. The handler can make use of these arguments while handling the error.

Here is the second form of the `raise` statement -

```
raise exceptionClass
```

Here we write just the exception name without any parentheses and arguments. This is actually equivalent to writing `raise exceptionClass()`. The initializer is called without any arguments and the instance is created in this form also. The exception that is raised can be a built-in exception or a user defined exception. We will see how to write user defined exception classes in [section 20.16](#).

The two forms of the `raise` statement will create a new instance of the exception class, and if there are any arguments then they are sent to the initializer of the class. You can also specify an existing instance of an exception class. The instance can be created before the `raise` statement and then can be raised by using the `raise` statement.

```
raise exceptionInstance
```

In the first two forms, the instance is implicitly created while here we are explicitly specifying the instance. The exception instance will be accessible

in the `except` block by using the `as` keyword which we have seen before. By using the `as` keyword, we get a reference to the instance, and so then we can access all the attributes and methods.

We can add attributes to the exception object before raising it. These attributes can be used in the `except` blocks to get information about the exception and handle it.

```
e = IndexError('some message ', 1, 2)
e.x = 10
e.message = 'xyz '
raise e
```

In the next two sections, we will see two more forms of the `raise` statement.

20.14 Re-raising Exception

When an exception is caught by an exception block, it dies, which means that it does not propagate further. However, there may be cases when you want to catch and handle an exception but do not want it to die; you want to pass it on to a higher level. So, basically, you want exception propagation to continue even after the exception has been caught and handled.

In the following `try` block, if a `LookupError` exception occurs, it will be caught by the matching `except` clause, statements of that `except` block will be executed and the exception will die.

```
try:
    .....
    .....
except LookupError:
    .....
    .....
except ZeroDivisionError:
```

```
.....  
.....
```

If we want the `LookupError` exception that is caught, to propagate further then we can write the `raise` statement inside the `except` block.

```
try:  
.....  
.....  
except LookupError:  
.....  
.....  
    raise  
except ZeroDivisionError:  
.....  
.....
```

Now, after execution of the statements inside the `LookupError` exception block, the exception will propagate to a higher level because of the `raise` statement.

In this form of `raise` statement, we write just the `raise` keyword without any exception name. This is called reraising the exception; it just reraises the currently active exception, so here no exception object is created. This form of `raise` statement can be used only inside an `except` block or any function that is called directly or indirectly by the `except` block. This statement will re-raise the currently active exception, which is the exception that was caught by the `except` block.

If you use this form at some place where there is no currently active exception, then it will result in a `RuntimeError` exception.

When this form of `raise` statement is executed, the `except` block in which this is placed terminates, and the exception propagation continues. It searches for appropriate `except` blocks in higher levels, and if not caught

anywhere, then it is caught by the default exception handler which prints the traceback and terminates the program.

We use this form of `raise` statement when an exception is caught inside an `except` block, but we do not want that exception to die even after the execution of that `except` block. We want the exception to propagate to the higher levels. We can reraise it further by using this `raise` statement.

You might want to reraise an exception if you do not have enough information to tackle the exception in the current `except` block or you can only partially handle the exception. You can do something in the `except` block like just log the exception message to a log file, or perform some cleanup and then raise it again. It is commonly used in catch all exception handlers.

```
try:
    .....
    .....
except Exception as e:
    .....
    .....
    raise
```

This `except` clause will catch any exception that is raised inside the `try` block, and then inside the `except` block you can perform some general cleanup, logging, or any other processing of the exception and then you can reraise the exception so that it propagates further and can be handled by another specific handler at higher levels. Let us see an example:

```
def func():
    try:
        f = open('xyz')
    except Exception as e:
        print('An exception occurred', e)
print('Start')
```

```
try:
    func()
except FileNotFoundError as e:
    print('Caught a FileNotFoundError', e)
    print('Handling the error')
print('End')
```

Output-

Start

An exception occurred [Errno 2] No such file or directory: 'xyz'

End

A `FileNotFoundError` exception occurs while opening the file. It is caught by the `except` clause of the `try` statement that is inside `func`, and it dies after that. Now, let us reraise the exception inside the `except` block by writing a `raise` statement.

```
def func():
    try:
        f = open('xyz')
    except Exception as e:
        print('An exception occurred', e)
        raise
print('Start')
try:
    func()
except FileNotFoundError as e:
    print('Caught a FileNotFoundError', e)
    print('Handling the error')
```

```
print('End')
```

Output-

Start

An exception occurred [Errno 2] No such file or directory: 'xyz'

Caught a FileNotFoundError [Errno 2] No such file or directory: 'xyz'

Handling the error

End

Now, we can see that the exception propagated and was caught by the `except` clause of the `try` statement at the module level.

20.15 Chaining Exceptions

We can add an optional `from` clause in the `raise` statement; this clause allows us to raise an exception from another exception. This way we can change or transform the current exception. We catch a type of exception and raise another type of exception. This is called exception chaining.

```
raise exceptionClass(optional arguments) from  
originalException
```

Here `originalException` is the exception object that is caught, and `exceptionClass` is the type of the new exception that will be raised. This original exception will be attached to the newly raised exception's `__cause__` attribute. The new exception contains the details of the original exception. Let us see an example.

```
def func():  
    try:  
        5 / 0  
    except ZeroDivisionError as excp:
```

```
        raise RuntimeError('An error occurred') from
excp
func()
```

We are raising `RuntimeError` in response to the `ZeroDivisionError` exception. If this new exception that is raised is not caught, then Python will print information of both the exceptions in the traceback. Here is the output of this program:

Output-

Traceback (most recent call last):

```
File "C:\Users\deepali\test.py", line 3, in func
    5 / 0
```

`ZeroDivisionError: division by zero`

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

```
File "C:\Users\deepali\test.py", line 6, in
<module>
    func()
File "C:\Users\deepali\test.py", line 5, in func
    raise RuntimeError('An error occurred') from
excp
```

`RuntimeError: An error occurred`

Now, let us enclose the call to function `func` inside a `try` block. The `except` clause for this `try` block catches the `RuntimeError` exception and inside the `except` block, we print the exception object and its `__cause__` attribute.

```
def func():
    try:
        5 / 0
```



```
except ZeroDivisionError as excp:
    raise RuntimeError('An error occurred') from
excp
try:
    func()
except RuntimeError as e:
    print(e)
    print(e.__cause__)
```

Output-

```
An error occurred
division by zero
```

From the output, we can see that `__cause__` attribute contains the message of `ZeroDivisionError` exception. So, we have a chain of two exceptions and this chain can be arbitrarily long. The causes of the exceptions are chained and this helps in debugging.

We can specify `None` in the `raise` statement, if we want to cancel any chained exceptions that have been gathered till now.

```
raise RuntimeError('An error occurred') from None
```

After making this change, if we run the program then this will be the output.

Output-

```
An error occurred
None
```

The exception chaining happens automatically when an exception is raised inside an `except` block or a `finally` block.

```
def func():
    try:
        5 / 0
    except ZeroDivisionError as excp:
```

```
        print(exc)

func()
```

Here a `NameError` exception occurs while handling the `ZeroDivisionError` exception. If we run this code, we can see the information of both the exceptions in the traceback.

Traceback (most recent call last):

```
File "C:\Users\deepali\test.py", line 4, in func
    5 / 0
```

`ZeroDivisionError: division by zero`

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

```
File "C:\Users\deepali\test.py", line 8, in
<module>
```

```
    func()
```

```
File "C:\Users\deepali\test.py", line 6, in func
    print(exc)
```

`NameError: name 'exc' is not defined. Did you mean: 'excp'?`

Here the exceptions are implicitly chained. In this case, the `__cause__` attribute is not set. Instead the `__context__` attribute is set to the original exception. We can catch the exception and see its `__context__` attribute.

```
def func():
    try:
        5 / 0
    except ZeroDivisionError as excp:
        print(exc)

try:
```

```
func()  
except Exception as e:  
    print(e)  
    print(e.__context__)
```

Output-

```
name 'exc' is not defined  
division by zero
```

So, we saw the following two forms of `raise` statement in this section, the first one is to raise another exception in response to an exception and the second one can be used to suppress chaining of exceptions.

```
raise exceptionClass(optional arguments) from  
originalException  
  
raise exceptionClass(optional arguments) from None
```

20.16 Creating your own exceptions in Python (Custom exceptions)

We have seen that there are many predefined exception classes in Python. These are raised by Python in appropriate situations, and you can also raise them in your code if required. While writing code, there can be some exceptional conditions that are specific to your application. In these cases, you can create your own exceptions that suit your requirements. These are called custom exceptions or user-defined exceptions. These exceptions can be used in the same way as the built-in exceptions.

We know that every exception in Python is a class, and so if you want to create your own exception, you need to define a class. To create a custom exception, you have to define a new exception class that is derived from the built-in `Exception` class directly or indirectly.

```
class MyError(Exception):  
    pass
```

Here we have created a new class that is inheriting from the `Exception` class; the class does nothing, but it needs a line of code to satisfy the syntax, so we have written the `pass` statement.

We have defined an empty class that does nothing, but since it is derived from the built-in class `Exception`, it has all the attributes and methods that are there in the `Exception` class. It can be used like any other built in exception class. Let us try to raise an exception of this type:

```
try:
    raise MyError
except MyError as e:
    print(e)
```

Inside the `try` block, we are raising the `MyError` exception, and in the `except` clause, we are catching it and then printing the exception object. We know that whenever a built-in exception instance is printed inside the handler, it prints those things that are passed when the exception instance was created. This is because of the inherited `__str__` method of the `BaseException` class. This behaviour is inherited by our custom classes also. So now let us send some arguments:

```
try:
    raise MyError('MyError exception raised', 1, 2,
3)
except MyError as e:
    print(e)
```

Output-

```
('MyError exception raised', 1, 2, 3)
```

From the output, we can see that the arguments are printed when we print the exception. The `args` attribute of the `Exception` class is also inherited.

```
try:
    raise MyError('MyError exception raised', 1, 2,
3)
```

```
except MyError as e:
    print(e)
    print(e.args)
```

Output-

```
('MyError exception raised', 1, 2, 3)
('MyError exception raised', 1, 2, 3)
```

So, although we have not defined anything in our class, all the functionality of the `Exception` class is available to our class. We can display the exception in string form and we have the `args` attribute that contains all the arguments that were passed at the time of creation of instance.

If we want, we can add attributes to our class. We can also define a custom display for our exception. To customize the display, we have to override the `__str__` method in our class. In the following class, we have added two attributes inside the `__init__` method, and we have also defined a `__str__` method.

```
class MyError(Exception):
    def __init__(self, x, y):
        self.data = x
        self.value = y
    def __str__(self):
        return f'Exception of type MyError raised, {self.data}, {self.value}'

try:
    raise MyError(23, 45)
except MyError as e:
    print(e)
    print(e.data, e.value)
```

Output-

```
Exception of type MyError raised, 23, 45
```

```
23 45
```

While raising the exception, we sent two arguments which are used to initialize the two attributes. When we print the exception instance, the string inside the `__str__` method is printed. This way, we can customize the display of our exception instance. If this exception is not caught, then also this string will be printed along with the name of the exception. For example, if the `LookupError` was specified in the `except` clause, then the exception raised in the `try` block will not be caught, and we will see a message that is actually the string specified in the `__str__` method.

This will be the output of the above program if we change the exception in the `except` clause to `LookupError`.

```
Traceback (most recent call last):
```

```
File "C:\Users\deepali\test.py", line 11, in  
<module>
```

```
    raise MyError(23,45)
```

```
MyError: Exception of type MyError raised, 23, 45
```

The attributes attached to the object can be accessed inside the `except` block. We have printed the attributes `data` and `value` inside the `except` block. This way, we can send state information with the exception instance. The `raise` statement and the place where the exception is caught might be in different files, so this passing of extra detail can be useful for the handler.

When you define your own `__init__` method and define attributes, you can use keyword arguments also.

```
raise MyError(data=23, value=45)
```

You cannot do this if you use just the default `args` attribute provided by the `BaseException` class.

You can also define methods in your custom exception class and these methods can be called inside the handler with the help of the exception instance. Let us define a method in our `MyError` class:

```
class MyError(Exception):
```

```

def __init__(self, x,y):
    self.data = x
    self.value = y

def __str__(self):
    return f'Exception of type MyError
raised, {self.data}, {self.value}'

def func(self):
    print('func called')

try:
    raise MyError(23,45)
except MyError as e:
    print(e)
    e.func()

```

Output-

```

Exception of type MyError raised, 23, 45
func called

```

So, this way, by defining classes for exceptions, you can use all the object-oriented programming features in your exception handling mechanism. In our next example, we have a class that is used to create exceptions that are raised when a value is out of range.

```

class OutOfRangeError(Exception):
    '''Exception raised when a value is out of
acceptable range'''
    def __init__(self,name, minValue,maxValue):

```

```

        self.name = name
        self.minValue = minValue
        self.maxValue = maxValue
    def __str__(self):
        return f'{self.name} should be between
{self.minValue} and {self.maxValue}'
try:
    age = int(input('Enter age : '))
    if age < 18 or age > 60:
        raise OutOfRangeError(name = 'age',
minValue=18, maxValue=60)

    salary = int(input('Enter salary : '))
    if salary < 10000 or salary > 500000:
        raise OutOfRangeError('salary', 10000,
50000)
except OutOfRangeError as e:
    print(e)

```

Sample Run 1-

Enter age : 23

Enter salary : 0

salary should be between 10000 and 50000

Sample Run 2-

Enter age : 3

age should be between 18 and 60

In the `__init__` method, we have defined three attributes, the name will be the name of value that goes out of range, and the other two are the

boundary values of the range. We have customized the display also by defining the `__str__` method.

In the `try` block, we raise the `OutOfRangeError` exception when the value of `age` goes out of range, and when the value of `salary` goes out of range. In the first `raise` statement, we have used keyword arguments and in the second one we have used normal positional arguments.

You can derive your custom class from the `Exception` class or any other class that derives from `Exception` class. We know that most of the built-in classes are derived from the `Exception` class, so we can derive our custom class from any other built-in class also if it makes more sense. For example, if your custom exception is related to some arithmetic error, then you can derive it from the `ArithmeticError` class, if it is related to some attribute problem, you can derive it from the `AttributeError` class. So, you can decide which category your custom exception falls into, and you can derive from that particular exception class. This `OutOfRangeError` deals with a problem in the value, so we can derive it from the `ValueError` class.

```
class OutOfRangeError(ValueError):  
    '''Exception raised when a value is out of  
    acceptable range'''  
  
    .....  
  
    .....
```

Any code that catches the `ValueError` will catch this exception also. If you had not created this `OutOfRangeError` custom exception and raised `ValueError` in out-of-range cases, then the client calling your code would have no choice, and they would have to catch `ValueError` exception only, but now the clients can be selective in their handlers. They can now catch only the `OutOfRangeError` exception, if they do not want to catch all the `ValueError` type of exceptions. So, if you define your custom exceptions in your module then you give the client an option to handle your module's exceptions separately from other exceptions.

We have seen that in our custom exception class, we can do anything that we can do in a normal class, we can define new attributes and methods. Thus,

we can store some specific information in exception objects which can be accessed by the handlers.

Custom exception classes are generally not very complicated, they are very simple with only a few attributes. Most of the times you will need to define a custom exception class which is actually empty; you define it so that you get an exception with a name that is specific to your application. The name of the exception would tell the user about the purpose of the exception and so it makes your code more readable. Anybody reading or using your code can better understand the reason of the exception.

Empty classes can be defined by writing a `pass` statement or even a docstring would be sufficient.

```
class CustomError:
    pass

class CustomError:
    ''' Docstring for this class '''
```

You must have noticed that the names of all the standard exceptions end in 'Error', so the custom exceptions are also generally given names that end in 'Error'. This is not compulsory but it good to do so.

In the section on built in exceptions, we had seen the benefits of categorising exceptions using inheritance. It gives the handling code an option to handle a general category of exceptions. New exceptions can be raised inside your code in future, without breaking the existing client code. This is why all the standard exceptions are organised in an inheritance hierarchy. When we are working on a big and complicated application that includes many custom exceptions, it is better to group together the custom exceptions using inheritance.

We can create an exception tree for our application, by defining a root exception, and then make all other custom exceptions inside our application, inherit from this root exception directly or indirectly. The root exception is made to inherit from the built in `Exception` class. This way, all exceptions of our application will be inherited from the `Exception` class indirectly. Let us see an example to understand the benefits of arranging exceptions in a hierarchy.

Suppose you are creating an application, and you identify certain situations in which you need to raise exceptions. You decide to raise your application specific custom exceptions, and so you write separate classes for them.

```
NoDataFoundError           HostnameNotResolvedError
InvalidNumberError

StorageError       ProtocolFailedError
NegativeNumberError

CursorAlreadyOpenError  AddressNotReachableError
OutOfRangeError

InvalidCursorError      NetworkChangedError
```

The first group of exception classes are related to database problems, the second group is related to network problems and the third group is related to input problems. All of them are derived from the built in `Exception` class so that they qualify as exception classes. When clients use your application, they have to write `except` clauses that list all the exceptions even if a handler can handle a category of exceptions.

```
try:

    .....

    .....

except (HostnameNotResolvedError,
        ProtocolFailedError, AddressNotReachableError,
        NetworkChangedError):

    .....

    .....

except (NoDataFoundError, StorageError,
        CursorAlreadyOpenError, InvalidCursorError):

    .....

    .....
```

The first `except` clause is written to handle all network related errors, so all the errors have to be specified in it. It would be better if we could group

these exceptions using inheritance. It would give the client an option to catch a category of errors. If the client wants to catch all network related errors, then he can catch all of them using the category name rather than naming all of them in the `except` clause.

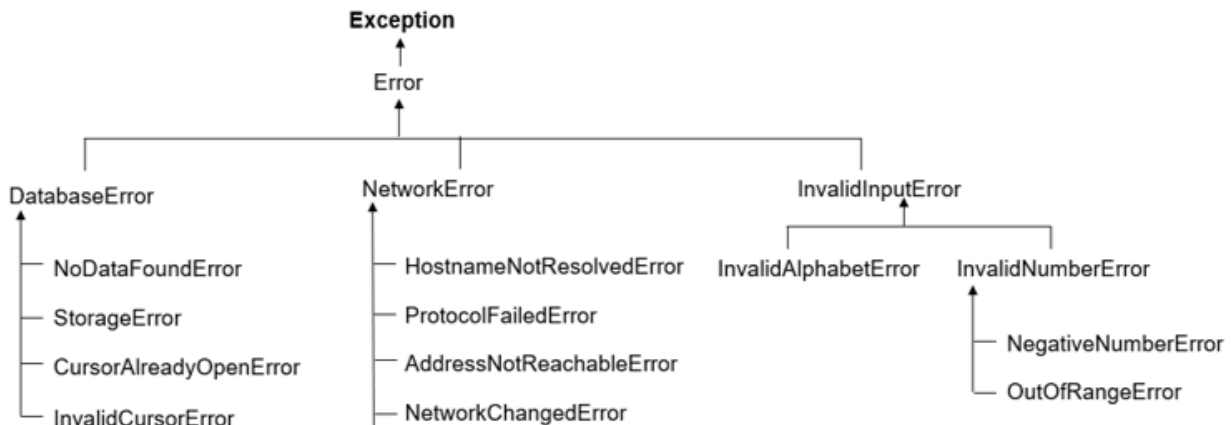


Figure 20.15: Exception hierarchy

The classes `DatabaseError`, `NetworkError`, and `InvalidInputError` act as base classes for the three groups of exceptions. We have created a root exception class named `Error` for our application; all these classes inherit from this root class, which inherits from the built-in `Exception` class. So now we have organized our application-specific custom exceptions using inheritance.

Exceptions grouped together in an inheritance hierarchy are better than standalone exceptions. You can create an exception hierarchy for your application and then the client code can catch all exceptions raised by our application by using the root class of your application. Clients will also be able to separately handle the exceptions of your application. So, the client of your code gets the choice of catching a specific custom exception from your application or a broader range of exceptions using the base class.

Now in the `except` clause, instead of listing all the network related exceptions we can just write the base class `NetworkError` and all the network exceptions will be caught here. Similarly, we can specify base class `DatabaseError` to catch all the database related errors.

`try:`

```

.....

.....

except NetworkError:

.....

.....

except DatabaseError:

.....

.....

```

In future, when you update your code and you need to add a new exception, it would not be a problem for the client code that uses a base class from your hierarchy. For example, if you add a new exception named `ConnectionClosed` as a derived class of `NetworkError` class, then your user's code will keep working if it catches the `NetworkError` exception; no changes will have to be made to the code. This insulates your client code from any changes in your exceptions set. So, your application becomes future proof.

The approach where you had standalone exception classes will create a problem for your clients since they will have to add the new exception in the `except` clauses where they are using your application. So, if you create a new exception class and your code raises the new exception, then the client will have to make changes at many places in the code.

Thus, in an application where there is hierarchical arrangement of exceptions, if you want to add a new exception then you can just inherit from a base class and your client code does not need to be changed.

Defining custom exceptions and arranging them in an inheritance hierarchy also helps in identifying bugs in your application. If your application raises any exception, other than the ones that it is supposed to raise then it is a bug in the application.

Let us see the program for the inheritance hierarchy that we have seen. In the following file, we have defined a root exception class that is derived from the `Exception` class. The classes `DatabaseError`, `NetworkError` and `InvalidInputError` are derived from the root class, and then we

have other classes derived from these two classes. After the class definition, we defined two functions that raise some of these errors:

-----myfile.py-----

```
class Error(Exception):
    '''Base class for all the exceptions raised in
    this application'''
class DatabaseError(Error):
    pass
class NetworkError(Error):
    pass
class InvalidInputError(Error):
    pass
class HostnameNotResolvedError(NetworkError):
    pass
class ProtocolFailedError(NetworkError):
    pass
class ConnectionClosedError(NetworkError):
    pass
class NetworkChangedError(NetworkError):
    pass
class AddressNotReachableError(NetworkError):
    pass
class InvalidNumberError(InvalidInputError):
    pass
class InvalidAlphabetError(InvalidInputError):
    pass
```

```

class NegativeNumberError(InvalidNumberError) :
    pass
class OutOfRangeError(InvalidNumberError):
    pass
class InvalidCursorError(DatabaseError):
    pass
class NoDataFoundError(DatabaseError):
    pass
class StorageError(DatabaseError):
    pass
def func1():
    print('func1 called')
    raise InvalidCursorError
def func2():
    print('func2 called')
    raise NetworkError

```

The following file contains the client code that imports `myfile` and calls `func1`.

-----clientcode.py-----

```

import myfile
print('Start')
try:
    myfile.func1()
    print('End')
except myfile.DatabaseError as e:
    print(type(e))

```

```
print('End')
```

When we execute this code, `func1` raises `InvalidCursorError`, and it is caught by the `except` block.

Now, suppose in the file `myfile.py` you add a new database exception `CursorAlreadyOpenError` and the function `func1` now raises this new exception.

```
class CursorAlreadyOpenError(DatabaseError):  
    pass  
  
def func1():  
    print('func1 called')  
    raise CursorAlreadyOpenError
```

The client code will still work and will be able to catch the new error.

20.17 Assertions

We have seen that an exception can explicitly be raised by using the `raise` statement. There is another statement that can also raise an exception which is the `assert` statement. It can raise only `AssertionError` exception and not any other type of exception. This statement is used as a debugging tool to detect programming bugs. Here is the syntax of this statement:

```
assert condition
```

After the `assert` keyword, a condition is written. When the `assert` statement is executed, this condition is tested. If it is `True`, then the normal program flow continues. If it is `False`, then an `AssertionError` exception is raised. So, this statement is used to test the truthiness of a condition. You can think of this `assert` statement as a conditional `raise` statement.

```
if not condition:  
    raise AssertionError
```


This is somewhat equivalent to the assert statement. If the condition is not True, then an `AssertionError` exception is raised. We know that raising an exception means that an exception instance is created, and the normal program flow stops. Like other exceptions, if this exception is not handled, then it terminates the program.

In the `assert` statement, after the condition, we can write an expression which is optional.

```
assert condition, expression
```

If this expression is there, then it serves as the argument for initializer of the `AssertionError`. This is generally an error message. You can think of this statement as equivalent to the following code:

```
if not condition:
    raise AssertionError(expression)
```

Let us see some examples to see how it works:

```
>>> a = 2
```

```
>>> b = 3
```

```
>>> assert a < b
```

```
>>> assert a > b
```

```
Traceback (most recent call last):
```

```
File "<pyshell#3>", line 1, in <module>
```

```
    assert a > b
```

```
AssertionError
```

The condition `a < b` is True, so nothing happened when we wrote the `assert` statement with this condition. When we changed the condition to `a > b`, an `AssertionError` exception was raised because the condition is False. Now, let us write an expression after the condition.

```
>>> assert a > b, 'a cannot be less than b'
```

```
Traceback (most recent call last):
```

```
File "<pyshell#5>", line 1, in <module>
```

```
assert a>b, 'a cannot be less than b'
```

```
AssertionError: a cannot be less than b
```

The expression that we have written after the condition is a string that represents the error message. This string was sent as argument when the exception instance was created. Instead of the string we can send any other expression also, but generally a string is sent that represents the error message.

Now, let us see when to use these `assert` statements in our code and what is their use. They are used for debugging during development time; they can be used to put sanity checks in your code. They are useful debugging tools that alert you when there is some bug in your code. Let us understand with the help of an example:

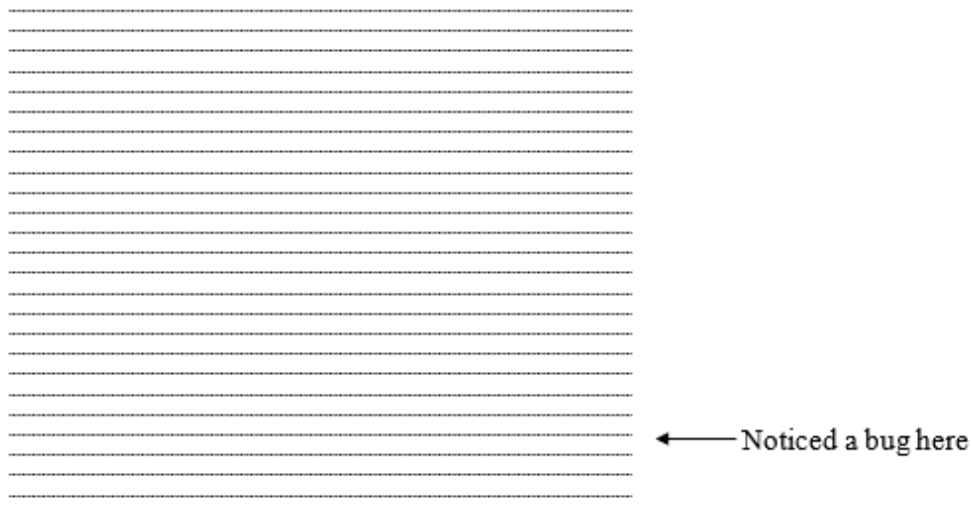


Figure 20.16: Noticed a bug in the code

Suppose this is the code that you have written and you notice a bug at some point in the program. The bug could be in the form of wrong output or a program crash or something else that goes wrong. The place marked in the figure with an arrow is the point at which your program fails, and you know that there is a bug in your program. You notice the bug at this place, but the root cause of the bug could be somewhere earlier in your code. Maybe it is because of a programming mistake that you did in the code or a computation that went wrong or maybe you called a function and it returned a wrong

value, and you used that return value here which became the reason of your bug.

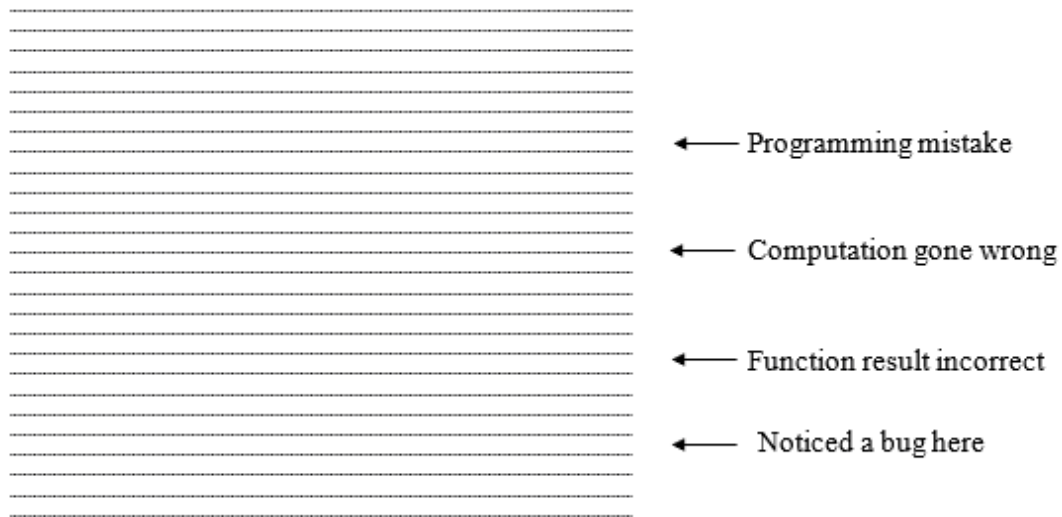


Figure 20.17: Probable reasons for the bug

a

So, there could be mistakes in your code that become apparent only after some time. To get to the root cause of the bug that you noticed, you need to examine the whole code before it and figure out what went wrong, where. If you had put sanity checks in between the code, this process of debugging would have become faster and easier. Let us do some sanity checks with the help of `assert` statements.

```
assert 0 < x < 100, 'x is incorrect'

assert myList, 'myList cannot be empty'

r = func()
assert r>0, 'r is not positive'
```

Figure 20.18: assert statements introduced in the code

You assert that these conditions should hold True, if everything is going as expected. If something goes wrong then the `assert` condition will fail and an `AssertionError` exception will be raised which will crash the program immediately.

With the statement `assert 0 < x < 100, 'x is incorrect'`, you are saying that we assert that if everything is going fine, the value of `x` will be between 1 and 100, and if it is not, then the program should terminate here. With the statement `assert myList, 'myList cannot be empty'`, you assert that `myList` will be non-empty, and if it is empty then the program should terminate here. In the last statement, you assert that the return value of the function `func` will be positive, if it is not then, the program should terminate here.

Including the custom messages makes the problem clear and can help in debugging. Since `AssertionError` is an exception, a stack trace will be displayed, which can also help in locating the cause of the bug. So, you have declared that these conditions should always hold True, and if any of these conditions is not met, then it means that there is a bug in the program. In that case, the program crashes and the traceback and this message will help you locate the bug and fix it.

In the code without the `assert` statements, the code might continue till the problem becomes noticeable while in the code with `assert` statements, your code will halt as soon as an assertion fails.

By putting these sanity checks in between, the programmer can make sure that the code is doing its job correctly and is working as expected. These sanity checks are runtime self-checks in the program. They perform automatic debugging while the program is running. These `assert` statements are useful debugging tools that alert you when there is some bug in your code. They help the developer find the reason of a bug quickly by failing fast. With the help of this example, we have seen that failing fast is better than failing later and then spending lot of time and effort in finding the reason of the failure.

An `assert` condition should fail only when there is a bug in the program, so an `AssertionError` exception is raised only when a bug is detected, and in that case the program needs to terminate. Thus, the `AssertionError` exception is normally not handled with `try...except` blocks like other exceptions. If an assertion is `False`, then the program should crash.

The `assert` statements should be used only during the development to identify any bugs in the code, their purpose is to alert the programmer about any bugs. They should not be used to detect and handle any normal runtime errors. The reason for this is that `assert` statements can be disabled. To understand, let us see how they are implemented.

`__debug__` is a built-in name that is set to `True` under normal circumstances, and when `-O` flag is used on the command line it is set to `False`. `O`, here, stands for optimization. So, when `-O` option is used, all statements that are written with the condition `if __debug__` are skipped and they will not be executed. Earlier, we had seen that we can think of `assert` statement as equivalent to:

```
if not condition:
    raise AssertionError(expression)
```

Actually, it is equivalent to this:

```
if __debug__:
    if not condition:
```

```
raise AssertionError(expression)
```

`assert` statements will be executed only when `__debug__` is `True`. If the user runs the program in optimized mode by using the `-O` flag on the command line, then `__debug__` will be `False`, and hence all the `assert` statements will be skipped. So, `assert` statements can be disabled using the `-O` flag on the command line. That is why they should not be used for regular error detection. For example, they should not be used to validate input data or handling errors such as resource not found, because if the user chooses to run the program in the optimized mode, then all the assertions will be disabled and your regular error checking will not happen, which might result in incorrect program behaviour.

For regular run time errors, you can use `if` statements to check conditions, and `raise` statement to raise exceptions. `assert` statements are used only during development time to ensure that certain conditions hold true. You should not rely on them for any checking any run time errors.

Exercise

1. In which of these cases does Python raise an exception.
(A) When a run time error occurs
(B) When the program gives incorrect output
2. If your program runs but does not do what it is supposed to do then it is said to have _____.
(A) Syntax Error (B) Logical error
3. In which of these approaches, do we put checks before attempting the operation.
(A) LBYL (B) EAFP
4. Syntax errors occur when the program is executing.
(A) True (B) False
5. In EAFP approach, we execute the code and if something goes wrong, we deal with it.

(A) True (B) False

6. What will happen when we try to execute this code?

```
numbers = [2, 4, 6, 8, 10]  
print(numbers[10] + 10)
```

(A) Syntax Error occurs

(B) An exception is raised

7. _____ is the base class for all the standard exceptions.

(A) Error (C) BaseException

(B) Exception

8. Most of the built in exception classes are derived directly or indirectly from _____ class.

(A) Exception (C) SystemExit

(B) Error

9. In Python 3, we can write both string-based exceptions and class-based exceptions.

(A) True (B) False

10. If an exception is mentioned in a handler, then it will handle that exception and also any subclass of that exception.

(A) True (B) False

11. To catch a whole category of errors, we can specify the _____ in the handler.

(A) subclass (B) superclass

12. Which of these exceptions is raised when you use an identifier that has not been defined?

(A) AttributeError

(B) NameError

(C) ValueError

13. Inside the _____ block we write those statements that can cause exceptions
(A) `try` (B) `except`
14. If an exception occurs inside the `try` block, the remaining statements of the `try` block are _____
(A) not executed
(B) executed after executing the appropriate `except` block
15. The code to handle errors is written inside the
(A) `except` block
(B) `else` block
(C) `finally` block
16. A `try` statement can have a _____
(A) single `except` clause
(B) multiple `except` clauses
17. The `finally` block is placed _____ all the `except` blocks.
(A) before (B) after
18. _____ block is always executed before leaving the `try` block.
(A) `else` (B) `finally`
19. When no exception occurs in `try` block, the `finally` block will be executed after the execution of _____
(A) `try` block
(B) corresponding `except` block
20. The code for releasing any resources is placed in the _____ block.
(A) `else`
(B) `except`
(C) `finally`

21. If you are writing a `finally` block, there should be at least one `except` block.
(A) True (B) False
22. If you are writing an `else` block, there should be at least one `except` block.
(A) True (B) False
23. The `else` block should be placed _____ the `finally` block.
(A) before (B) after
24. The code in the _____ block is executed when no exception occurs during the execution of the `try` block.
(A) `except`
(B) `else`
(C) `finally`
25. The `finally` block is executed only when the raised exception is handled.
(A) True (B) False
26. To access the exception object, you can use the ____ keyword in the `except` clause.
(A) `as` (C) `and`
(B) `or` (D) `from`
27. All exception classes have an attribute named _____.
(A) `arguments`
(B) `args`
(C) `values`
28. The `args` attribute is a _____.
(A) list
(B) tuple
(C) dictionary

29. The `raise` statement can raise _____.
(A) only built-in exceptions
(B) only user defined exceptions
(C) both built-in and user defined exceptions
30. In the `raise` statement, writing the exception name is _____.
(A) optional
(B) compulsory
31. When an exception is reraised, a new exception object is created.
(A) True (B) False
32. If you re-raise an exception at some place where there is no currently active exception then it will result in a _____ exception.
(A) `SyntaxError`
(B) `RuntimeError`
33. The _____ clause in the `raise` statement allows us to raise an exception from another exception.
(A) `for`
(B) `from`
(C) `with`
34. In case of implicitly chained exceptions the _____ attribute is set to the original exception.
(A) `__cause__`
(B) `__context__`
35. User defined classes are generally derived from the _____ class.
(A) `BaseException` (B) `Exception`
36. The `assert` statement can raise only `AssertionError` exception.
(A) True (B) False

37. If the condition in the assert statement is _____ then an AssertionError exception is raised.

(A) True (B) False

38. assert statements will be skipped if __debug__ is _____

(A) True (B) False

39. assert statements can be disabled by using the _____ flag on the command line.

(A) -D

(B) -O

(C) -P

40. What will be the output of the following code?

```
try:
    print(10 / 0)
except:
    print('Default except')
except ZeroDivisionError:
    print('ZeroDivisionError')
```

(A) Default except

(B) ZeroDivisionError

(C) Shows Syntax error

41. What will be the output if the user enters?

(i) 2 (ii) 0 (iii) two

```
def f1():
    print('AA')
    f2()
    print('BB')

def f2():
```

```

        print('CC')
        x = int(input('Enter a number '))
        print(10%x)
        print('DD')
print('Begin')
f1()
print('End')

```

42. What will be the output of the following code?

```

try:
    print(x + 5)
except ValueError:
    print('ValueError')
except TypeError:
    print('TypeError')

```

43. What will be the output if the user enters:

(i) Raj (ii) Ron (iii) Tom

```

students = {'Raj': [80, 60, 70], 'Deep': [80, 90], 'Ron': [], 'Sam': [70, 50]}
try:
    name = input('Enter student name : ')
    average_marks = sum(students[name]) / len(students[name])
    print(name, average_marks)
except KeyError:
    print('Invalid name')
print('End')

```

44. Modify the program in the previous question so that it keeps asking for a student name till the user enters a valid name. (Use while loop)
45. Modify the program that you wrote in the question 44, so that it catches `ZeroDivisonError` also.
46. What will be the difference in the output of the following two pieces of code?

(i)

```
students = {'Raj': [80, 60, 70], 'Deep': [80, 90], 'Ron': [], 'Sam': [70, 50]}

try:
    for name in students.keys():
        average_marks =
sum(students[name]) / len(students[name])
        print(name, average_marks)
except ZeroDivisionError:
    pass

print('End')
```

(ii)

```
students = {'Raj': [80, 60, 70], 'Deep': [80, 90], 'Ron': [], 'Sam': [70, 50]}

for name in students.keys():
    try:
        average_marks = sum(students[name]) /
len(students[name])
        print(name, average_marks)
    except ZeroDivisionError:
        pass

print('End')
```

47. What will be the output of the following piece of code?

```
try:
    print(3 / 0)
except Exception:
    print('xx')
except ZeroDivisionError:
    print('yy')
```

48. Rewrite this code using a single except handler:

```
try:
    func()
except IndexError:
    log_it()
except TypeError:
    log_it()
except ValueError:
    log_it()
```

49. Is there a better way to write the given except clause?

```
try:
    func()
except ArithmeticError, FloatingPointError,
OverflowError, ZeroDivisionError:
    print('Arithmetic problem')
```

50. What will be the output of the following code, if the user enters:

(i) 2 (ii) 200 (iii) two

```
try:
    age = int(input('Enter age : '))
```

```

except ValueError:
    print('Not a valid integer value')
if age < 0 or age > 120:
    print('Age cannot be more than 120 or
less than 0')
else:
    print('Age is', age)

```

What changes should be made to avoid the problem that occurs in (iii)?

51. Given below are two almost similar pieces of code. The first one uses `else` block and in the second one, the remaining code is written after the `try...except`. What is the difference between these two pieces of code, is the `else` block really required or will the two work in the same way?

(i) `while True:`

```

    try:
        age = int(input('Enter age : '))
    except ValueError:
        print('Please enter a valid integer
value')
        continue
    else:
        if age < 0 or age > 120:
            print('Age cannot be more than
120 or less than 0. Please enter again')
        else:
            print('Age is ', age)
            break

```

```
(ii) while True:
    try:
        age = int(input('Enter age  : '))
    except ValueError:
        print('Please enter a valid integer
value')
        continue
    if age < 0 or age > 120:
        print('Age cannot be more than 120 or
less than 0. Please enter again')
    else:
        print('Age is ', age)
        break
```

52. In the code given below, flag is taken so that the code after the try...except executes only when there is no IndexError. Rewrite the code without using the flag.

```
L = [1,2,3,4]
flag = False
i = int(input('Enter an integer  :'))
try:
    x = L[i] + 1000
    flag = True
except IndexError as e:
    print(e)
if flag:
    print(x)
```


53. Both the `int` function and `math.factorial` raise `ValueError` when some invalid value is given to them. In the given try block, if any of these functions raises a `ValueError`, then it is caught in the except block, and the message 'The text should be a positive integer' is displayed. What can you do to display different error messages in case of `ValueError` raised by `int` function and `ValueError` raised by the factorial function?

```
import math

while True:

    try :

        n = int(input('Enter a number : '))

        f = math.factorial(n)

    except ValueError:

        print('The text should be a positive integer')

    else:

        print('Factorial of the number is', f)

        break
```

Output given by the above code is this:

```
Enter a number : two
The text should be a positive integer
Enter a number : -2
The text should be a positive integer
Enter a number : 4
Factorial of the number is 24
```

Output that we require is this:

```
Enter a number : two
invalid literal for int() with base 10: 'two'
```

```
Enter a number : -2
```

```
factorial() not defined for negative values
```

```
Enter a number : 4
```

```
Factorial of the number is 24
```

54. Rewrite this code so that in case of a `ZerodivisionError`, the value from the first list is displayed.

```
import math

L1 = [10, 20, 30, 40, 70]
L2 = [2, 0, 2, 0, 7]
for m, n in zip(L1, L2):
    print(m / n)
```

55. In the following code, we have used a `try...except` block inside a `while` loop to ensure that the user enters a valid integer.

```
while True:
    try:
        n = int(input('Enter an integer : '))
        print(n + 100)
        break
    except ValueError:
        print('The text you entered is not valid')
```

Based on the above code, write a function named `input_int`, that you can use instead of `input` function to ensure that a valid integer is entered. The function will be called like this:

```
n = input_int()
print(n + 100)
age = input_int('Enter age : ')
print(age)
```

56. Similar to the `int_input` function that you wrote in the previous question, write a function `input_value` that can be used to input a value of any type. The function will be called like this:

```
n = input_value(int)
print(n + 100)
age = input_value(int, 'Enter age : ')
print(age)
x = input_value(float)
print(x)
length = input_value(float, 'Enter length : ')
print(length)
```

57. Write a function named `read_files` that takes a list of filenames as arguments and prints the contents of all the files in the list on the console. If an error occurs while opening or reading any file, it just ignores that error and continues reading the next file.
58. The following code does not work if the first file is not found. What changes can you make in it, to make it work?

```
for filename in ['data.txt', 'data11.txt',
                 'data2.txt', 'text.txt']:
    try:
        f = open(filename, 'r')
    except OSError:
        print(f'{filename} could not be
opened')
    else:
        print(f'{filename} has {len(f.read())}
characters')
    finally:
```

```
f.close()
```

59. Write a program that inputs the name of a file and reads it. If the file is not found, it asks the user to enter the name of the file again or 'x' to exit (use `sys.exit()`). The program keeps asking the user for a filename, till the user enters a file that is found or till he enters 'x'.
60. Write the output of the following code if user enters:

(i) 20 (ii) 100 (iii) thirty

```
minimum = 18
```

```
maximum = 60
```

```
try:
```

```
    age = int(input('Enter age : '))
```

```
    if age < minimum or age > maximum:
```

```
        raise ValueError(minimum, maximum)
```

```
except ValueError as e:
```

```
    print('Invalid age value')
```

```
    print(f'Value of age should be in between  
{e.args[0]} and {e.args[1]}')
```

```
else:
```

```
    print(age)
```

61. In the program of the previous question, what changes do you need to make so that it works for 'thirty' also?
62. Write a function named `factorial` that accepts a single argument and returns the factorial of its argument. It accepts only a positive integer as an argument, so raise a `TypeError` if the argument is not an integer (use `isinstance()`) and raise a `ValueError` if the argument is negative.
63. Is there anything wrong with this code?

```
marks = {'Sam': 20, 'John': 30, 'Tim': 25,  
        'Jim': 22}
```

```

try:
    name = input('Enter name ')
    if name not in marks.keys():
        raise KeyError(name)
    print(marks[name])
except KeyError as e:
    print(e, 'not present in the dictionary')

```

64. How would you write except blocks if you want to ignore `TypeError` and `ValueError`, and want to propagate up all other exceptions?

65. What is the difference between these two pieces of code?

(A)

```

def func():
    try:
        print(3 / 0)
    except ZeroDivisionError as e:
        print('Caught a ZeroDivisionError in
func : ', e)
    try:
        func()
    except ZeroDivisionError as e:
        print('Caught a ZeroDivisionError : ',
e)

```

(B)

```

def func():
    try:
        print(3 / 0)
    except ZeroDivisionError as e:

```

```

        print('Caught a ZeroDivisionError in
func : ', e)
        raise

    try:
        func()
    except ZeroDivisionError as e:
        print('Caught a ZeroDivisionError : ',
e)

```

66. Given below are two pieces of code, which one will abnormally terminate?

<pre> def func(): try: print(4 + 'x') print(4+'x') except TypeError as e: print('Caught a TypeError in func:',e) print('Caught a TypeError in func:',e) func() func() </pre>	<pre> def func(): try: print(4 + 'x') except TypeError as e: raise func() </pre>

67. How would you write the except blocks to catch and handle OSError, ValueError, and ZeroDivsionError and reraise the rest of the errors? Before reraising, print the name of the error.

68. What is the output of this code?

```

class CustomError(Exception):
    def __init__(self, x, y):
        self.data = x
        self.value = y

```

```

        def __str__(self):
            return f'CustomError raised,
{self.data}, {self.value}'
try:
    raise CustomError(4, 8)
except CustomError as e:
    print(e)
    x, y = e.args
    print(x, y)
    print(e.data, e.value)

```

69. In the following code, the exceptions of type `ZeroDivisionError`, `FloatingPointError` and `OverflowError` are transformed to the custom exception type `MyError`.

```

class MyError(Exception):
    pass
def func(m,n):
    try:
        print(m/n)
        import math
        print(math.exp(m))
    except (ZeroDivisionError,
FloatingPointError, OverflowError) as excp:
        raise MyError('An error occurred')
from excp

```

Write code to call the function `func` in a try block. In the except block, catch `MyError`, print the exception, and in a logfile write the

`__cause__` attribute of the exception and also write the values given by the `sys.exc_info()` function.

What do you think can possibly be the use of converting the exceptions here?

70. Custom exceptions can be used to exit out of nested loops. Rewrite the following program so that a custom exception is raised when target is found.

```
courses_data = {'Python': {'Average': ['Tom',
'Jim'],
                           'Bright': ['John',
'Tim', 'Ria']}},
                 'SQL': {'Average': ['Ken', 'Ben',
'Ron'],
                           'Bright': ['Max', 'Nia']}},
                 'Web Design': {'Average': ['Geo',
'Ray', 'Leo'],
                                'Bright':
['Sam'],
                                'Excellent':
['Roe', 'Pam']}},
                 }

target = input('Enter name : ')
found = False

for course, data in courses_data.items():
    for category, names_list in data.items():
        for rank, name in
enumerate(names_list, 1):
            if target == name:
                found = True
```



```
                break
            if found:
                break
        if found:
            break
    if found:
        print('Name : ', name)
        print('Course :', course)
        print('Category : ', category)
        print('Rank : ', rank)
    else:
        print(target, 'not found')
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



In this chapter, we will learn about the `with` statement and context managers. They are generally used to automate common resource management patterns like opening and closing files, connecting and disconnecting from a database, and locking and unlocking threads. Resources like files, database connections, or network connections are limited, and not managing them properly can lead to resource leaks, slow down, or sometimes data corruption. Whenever we work with such limited resources, we have to ensure that they are properly released after use. The context managers and the `with` statement help in the safe acquisition and guaranteed release of system resources. They also help avoid repetition of acquisition and release code. The most common use of context managers is to manage resources like files, locks, databases, or network connections; however, you can use them anywhere where you need to surround some portions of your code with some pre and post-code.

21.1 `with` statement

First, we will see the syntax of `with` statement and understand how it works, and after that we will see where it can be used. Here is the syntax of the `with` statement:

```
with expression as var:
    statements
```

The `expression` should be a context manager object, or it should produce a context manager object. When this `with` statement is executed, the first

thing that happens is that the `expression` is evaluated and it gives an object which is a context manager. Now let us see what is a context manager.

A context manager is an object that follows the Context Management Protocol. This protocol states that an object should support `__enter__` and `__exit__` methods to be qualified as a context manager.

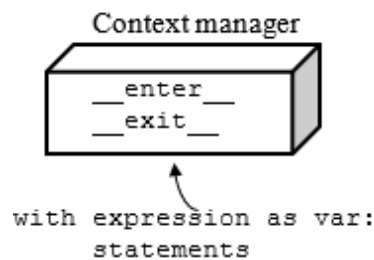


Figure 21.1: Context manager

The object that is returned by the expression should support the two magic methods `__enter__` and `__exit__`, then only it can be used in the `with` statement.

Any class that implements a context manager should have these two magic methods defined. When we instantiate such a class, the objects that we get are context managers.

```
class CM():
    .....
    def __enter__(self):
        pass

    def __exit__(self, exc_type, exc_value, traceback):
        pass
    .....
```

The `__enter__` method takes only a single argument, which is `self`, and the `__exit__` method takes three more arguments in addition to `self`. We will talk more about these arguments later. The class implementing a context manager can have `__init__` method and other methods also if required.

So, instances of any class that defines `__enter__` and `__exit__` methods conform to the Context Management Protocol and thus can be used in the `with` statement.

Now, let us see the flow of control when the `with` statement executes. As we have seen, first of all the expression written after the `with` keyword is evaluated and we get a context manager object. After this the `__enter__` method of this context manager is called. The value returned by the `__enter__` method is assigned to the variable that is specified after the `as` keyword.

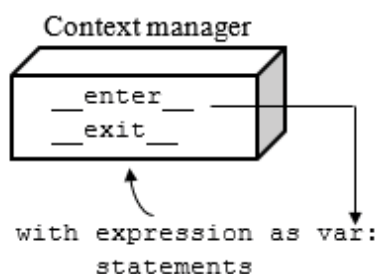


Figure 21.2: Context manager

The value that is returned by `__enter__` is something that we would like to use inside the `with` code block, this is generally the context manager itself, but it can be anything else also. So, mostly the `__enter__` method returns `self` but it can return something else, too. The `as` keyword and the variable written after it are optional. If they are not present, the value returned by `__enter__` is just discarded. This is why it is not necessary for `__enter__` to return any value.

After the `__enter__` method has finished executing and its return value has been assigned to `var`, the body of the `with` statement executes, so all the statements inside the `with` code block are executed.

After all the statements have been executed, the `__exit__` method of the context manager is called and executed. So, this is how the `with` statement works. Here is a review of the control flow:

- Expression is evaluated to get a context manager object
- The `__enter__` method of the context manager is executed
- Value returned by the `__enter__` method is assigned to `var`

- Statements inside the `with` block are executed
- The `__exit__` method of the context manager is executed

If an exception occurs during the execution of statements inside the `with` code block, then the rest of the statements in the `with` code block are skipped, and the control is transferred to `__exit__`. So, the `__exit__` method is always called when the `with` code block is exited, no matter how the block is exited whether it is due to the end of the block, a `return` statement or an exception.

Like the `finally` block of the `try` statement, the context managers's `__exit__` method is guaranteed to be always called, and so you can place any cleanup code in this method.

21.2 Implementing our own context manager

Before seeing other details, let us write a simple context manager class and use its instance in the `with` statement.

```
class CM():  
    def __enter__(self):  
        print('__enter__called')  
  
    def __exit__(self, exc_type, exc_value,  
exc_traceback):  
        print('__exit__called')  
  
with CM():  
    print('Hello')  
    print(15 / 3)  
    print('Bye')
```

We have defined a simple class which has only two methods `__enter__` and `__exit__`. It conforms to the Context Management Protocol and so instances of this class will be context managers. Inside the methods we are not doing much, we are just printing some messages so that we will know when these methods are called.

After defining the class, we have written a `with` statement. The expression `CM()` will give us an instance of class `CM`, so it gives us a context manager. The `as` keyword that we saw in the syntax is optional; we have not used it here. Inside the `with` code block, we have written three statements. When we execute the above code, we will get the following output:

```
__enter__ called
```

```
Hello
```

```
5.0
```

```
Bye
```

```
__exit__ called
```

We can see that first, the `__enter__` of the context manager was called, then the `with` code block was executed and, then the `__exit__` method was called.

Now, in our `CM` class, let us define `__init__` and another method named `do_something`.

```
class CM():
    def __init__(self, name):
        print('__init__ called')
        self.name = name
    def __enter__(self):
        print('__enter__ called')
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__ called')
```

```

        def do_something(self):
            print('Something is being done')
with CM('xyz'):
    print('Hello')
    print(15 / 3)
    print('Bye')

```

Output-

```

__init__ called
__enter__ called
Hello
5.0
Bye
__exit__ called

```

Inside the `__init__` method we have attached an attribute named `name`. The `__init__` method has a parameter, so now we have to send an argument at the time of instantiation. This is why the expression in `with` statement is `CM('xyz')` instead of `CM()`. When we execute the above code, first `__init__` is executed, then `__enter__` then the `with` code block is executed followed by `__exit__`.

The `with` statement is internally using the context manager object returned by the expression `CM('xyz')` for calling its `__enter__` and `__exit__` methods. The context manager object cannot be used inside the `with` code block since we do not have any reference to it there. For example, suppose we want to call the object's `name` attribute or call the method `do_something` inside the `with` code block; we cannot since we do not have any reference to the context manager object.

To make this possible, we will make the `__enter__` return `self`, and in the `with` statement, we will write the `as` keyword and a variable.

```

class CM():
    def __init__(self, name):

```

```

        print('__init__called')
        self.name = name
    def __enter__(self):
        print('__enter__called')
        return self
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__called')
    def do_something(self):
        print('Something is being done')
with CM('xyz') as c:
    print('Hello')
    print(15/3)
    print('Bye')
    print(c)
    print(c.name)
    c.do_something()

```

We know that whatever is returned by the `__enter__` method is assigned to the variable after the `as` keyword. The `__enter__` method returns `self`, and so the context manager itself is assigned to variable `c`. Now, inside the `with` code block, we can access the context manager object because the variable `c` will be a reference to the context manager object returned by the expression `CM('xyz')`. So now we can print `c`, `c.name` or call `c.do_something`.

When we execute the above code, we get the following output:

```

__init__called
__enter__called
Hello

```


5.0

Bye

<__main__.CM object at 0x00000225BF58B650>

xyz

Something is being done

__exit__ called

The `__enter__` method is generally made to return `self`, but it can return anything and whatever it returns will be assigned to the variable after the `as` keyword. Suppose now it returns the `name` attribute.

```
class CM():
    def __init__(self, name):
        print('__init__ called')
        self.name = name
    def __enter__(self):
        print('__enter__ called')
        return self.name
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__ called')
    def do_something(self):
        print('Something is being done')
with CM('xyz') as c:
    print('Hello')
    print(15 / 3)
    print('Bye')
    print(c)
```

Output-

```
__init__called
__enter__called
Hello
5.0
Bye
xyz
__exit__called
```

Now, inside the `with` code block, the variable `c` refers to the `name` attribute of the context manager object. `__enter__` is made to return something that we intend to use inside the `with` code block, and most of the time, it is the context manager itself.

We know that `as` keyword is optional, and if it is not present, the value returned by `__enter__` is discarded. This is mostly done when you need the side effects of the context manager. In these cases, the context manager is not assigned to anything, its entry and exit methods are called internally. When you need a handle to the context manager for performing operations inside the `with` block, you can include the `as` clause.

21.3 Exception raised inside with block

Let us see what happens when an exception is raised by any statement inside the `with` code block.

```
class CM():
    def __init__(self,name):
        print('__init__called')
        self.name = name

    def __enter__(self):
        print('__enter__called')
        return self
```

```

    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__called')
    def do_something(self):
        print('Something is being done')
with CM('xyz') as c:
    print('Hello')
    print(15/0)
    print('Bye')
    print(c)
    print(c.name)
    c.do_something()

```

Output-

```

__init__called
__enter__called
Hello
__exit__called
Traceback (most recent call last):
  File
"E:\Deepali\Programs\21_ContextManager\P21_5.py",
line 20, in <module>
    print(15/0)
ZeroDivisionError: division by zero

```

When the expression 15/0 is evaluated, a ZeroDivisionError exception is raised. From the output, we can see that after printing Hello, ZeroDivisionError exception was raised. So the rest of the with block is not executed, but the `__exit__` method is still called and

executed. So, we can see that the `__exit__` method is called even if the exit from `with` statement is due to an unhandled exception.

Now, let us talk about the three parameters of the `__exit__` method. If an exception occurs inside the `with` block, then the context manager's `__exit__` method is informed about that exception with the help of three arguments. The three arguments provide the details of the exception to the `__exit__` method. So, when an exception is raised while executing the `with` code block, the interpreter sends three arguments to the `__exit__` method, and that is why we need to place three parameters in its definition.

The first argument that is sent is the type of the exception, the second is the exception value, or the exception object and the third is the traceback object. These arguments are the same as those returned by `sys.exc_info()` function.

Let us print the three parameters of the `__exit__` method inside its definition:

```
def __exit__(self, exc_type, exc_value,
exc_traceback):
    print('__exit__ called')
    print(exc_type)
    print(exc_value)
    print(exc_traceback)
```

There is nothing special about these parameter names, you can use any other names also, but these are the names that are conventionally used. Now, when we run the previous code with this `__exit__`, we will get the following output for the execution of `__exit__`:

```
__exit__ called
<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x000001F2FC2FCDC0>
```

The `__exit__` method can either use this exception information to handle the exception or it can just ignore it. If no exception is raised inside the `with` code block, then the `__exit__` method is called with the three

arguments as `None`. In the `with` code block, if we change `print(15 / 0)` to `print(15 / 3)` there would not be any exception, and so on printing the three parameters, we will get `None`.

So, if any exception occurs while executing the `with` block statements, then the three parameters are filled with the exception details. Otherwise, all three are `None`.

When an exception occurs in the `with` block, the `__exit__` method has three options:

- (i) it can propagate the exception
- (ii) it can suppress the exception or
- (iii) it can raise another exception

The return value of the method `__exit__` indicates whether the exception is propagated or terminated. If the `__exit__` method returns `False` or any value whose Boolean value is `False`, then the exception is propagated up after `__exit__` finishes executing. If you return nothing, then by default `None` is returned, whose Boolean value is `False`, so any exception that has occurred propagates up to the next level. In our example, we have not returned anything from `__exit__` so `None` will be returned which is a Falsy value and so any exception will be propagated from this method. All the context managers in standard library, propagate the exception. So, this is the preferred behavior for your classes also, whenever you create any.

If the `__exit__` method wants to suppress the exception, it should return `True`. If `True` is returned, it means that the exception is not propagated to the next level. The exception just vanishes as if nothing has happened and the execution continues after the `with` statement. But this can be dangerous and should rarely be done. Suppressing exceptions leads to code that is very difficult to debug.

In our example, let us make `__exit__` return `True`, and after the `with` code block, we print a message so that we know whether the execution continues after the `with` block.

```
class CM():  
    def __init__(self, name):
```

```

        print('__init__called')
        self.name = name
    def __enter__(self):
        print('__enter__called')
        return self.name
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__called')
        return True
    def do_something(self):
        print('Something is being done')
with CM('xyz') as c:
    print('Hello')
    print(15 / 0)
    print('Bye')
    print(c)
print('End')

```

Output-

```

__init__called
__enter__called
Hello
__exit__called
End

```

From the output, we can see that the exception was swallowed by the `__exit__` method, and the execution continues after the `with` statement so the message 'End' is printed. If we delete `return True` from `__exit__` then `None` will be returned, which is a Falsy value, and so the

exception will be propagated up. It is not handled anywhere resulting in abnormal termination of the program.

The `__exit__` method has the full information of the exception along with the traceback, so it can also do something to handle the exception if required. For example it can at least log the exception. It can also raise another exception. If you want, you can selectively handle an exception and raise others.

```
def __exit__(self, exc_type, exc_value,
exc_traceback):
    print('__exit__called')
    if exc_type is ZeroDivisionError:
        print('Handling the ZeroDivisionError')
    return True
```

In this definition of `__exit__`, we are handling the `ZeroDivisionError` but if any other error occurs then it will be propagated up.

21.4 Why we need with statement and context managers

In this section, we will see why we need `with` statements and context managers. Suppose in your program, you have different pieces of code where you have to interact with a database. Before executing the code that communicates with the database, you need to execute some setup code that connects to the database and, after you have finished working with the database, you need to execute some teardown code that disconnects the database and performs any cleanup actions.

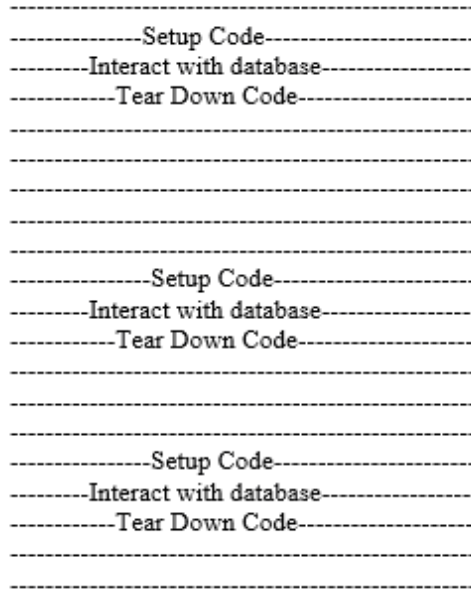


Figure 21.3: *Interaction with database*

You want to ensure that the cleanup code is executed no matter what happens, even if an exception occurs while working on the database. We have seen earlier that the `finally` blocks guarantee the execution of our cleanup code. So, we can put the teardown code in the `finally` block of the `try` statement.


```

-----Setup Code-----
try:
    -----Interact with database-----
finally:
    -----Tear Down Code-----
-----
-----
-----
-----
-----Setup Code-----
try:
    -----Interact with database-----
finally:
    -----Tear Down Code-----
-----
-----
-----
-----Setup Code-----
try:
    -----Interact with database-----
finally:
    -----Tear Down Code-----
-----
-----
-----

```

Figure 21.4: *Interaction with database using try...finally*

The `with` statement provides a better alternative to the `try...finally` approach. The setup and teardown code will be the same every time you interact with the database. So, when you make your context manager class, you can place the repetitive setup and teardown code in it, and then there will be no need to repeat the code every time. The setup code goes in the `__enter__` method, and the teardown code goes in the `__exit__` method.

```

-----
with CManager():
    -----Interact with database-----
-----
-----

-----
class CManager():
    def __enter__(self):
        ---Setup Code---

    def __exit__(self, exc_type, exc_value, traceback):
        ---Tear down Code---

-----
with CManager():
    -----Interact with database-----
-----
-----

-----
with CManager():
    -----Interact with database-----
-----
-----

```

Figure 21.5: *Interaction with database using the with statement*

When the `with` statement is executed, we will get a context manager object which will be an instance of class `CManager`. First, the `__enter__` method of this object will be executed, and so the setup code will execute. Then the statements inside the `with` code block will execute and after that the `__exit__` will execute, so the teardown code will execute. Even if any exception occurs while interacting with the database in the `with` code block, the `__exit__` method will be executed, and the database will be properly disconnected.

We can see that the code has become cleaner, less verbose, and more readable, and we also get the guarantee of execution of our cleanup code. The setup and teardown code can be lengthy and complex, and writing it every time you use the resource is not desirable; context managers help you avoid repeating the same code at many places, and at the same time, they give you the guarantee that the teardown code will definitely be executed.

We have moved the boilerplate entry and exit code in the context manager class, so we do not have to repeat it every time, and we can focus on the main task that we have to perform. The details of the setup and teardown code are hidden inside the context manager, and in your main program, only the database processing code is seen. So, we can abstract away most of the resource management logic by using a context manager.

The `try...finally` approach is more explicit; you can see the full code there, but that is why it is also more verbose. If you have to repeat it at many places in your program, then it increases the code size.

It is a very common thing to acquire a resource and then release it when we are done with it. We saw the example of connecting and disconnecting a database; the resource could be a network connection, file, lock, web transaction, or logged-in session, or we could temporarily change a setting in the program and then restore it back to the original, or we could start something like a timer and stop it automatically. In these types of scenarios, when there is some setup code and some teardown code that needs to be executed multiple times, you can create a context manager class and write the `with` statement. They provide us with a mechanism for automatic setup and teardown of resources.

So, they make our code more readable by simplifying the common resource management patterns. Of course, they help us avoid any resource leaks as they ensure that the resources are deallocated and default settings are restored in any case.

Here are some examples of cases where you need to execute some setup code and teardown code:

Setup code	Teardown code
Acquire a resource	Release the resource
Connect to database	Disconnect from database
Connect to network	Disconnect from network
Open a file	Close the file
Open a socket	Close the socket
Acquire a lock	Release the lock
Change a setting	Restore the default setting
Start timer	Stop timer

Table 21.1: Examples of setup and teardown code

So, when you want to ensure execution of some special code before or after a piece of code, and you want to do this multiple times in your program, you can use a `with` statement. Context managers are in a way like decorators, they are used to surround code with some special code, but the difference is that decorators are used to wrap defined blocks of code like functions or

classes, while with context managers, you can surround any arbitrary piece of code.

`with` statements are generally used when you want to temporarily acquire a resource, work with it, and release it, or when you have to restore some previous state that was temporarily changed for some time. However, `with` statements can be used only for objects that follow the context management protocol, while the `try...finally` approach can be used to perform cleanup actions for any sort of object.

21.5 Runtime context

Let us see what we mean by ‘context’ in the name context manager. A `with` statement creates a runtime context which is a kind of temporary environment that exists only while the `with` code block is executing.

Whatever statements you write inside the `with` block are executed in this special environment called the runtime context. For example, if you use the `with` statement for opening and closing a file, then the environment or context inside the `with` code block is that the file is open. If you take the database example, the context is that the database is connected, or in the lock example the context is that the lock is held.

The unique context provided by the `with` statement is not present in any code executed before or after it. When you come out of the `with` statement, the file is closed or the database has been disconnected, so the special context is gone. It was only a temporary environment that was created by the `with` statement. In this environment, generally a resource is temporarily acquired, the program interacts with the resource and then it is released. So, whenever you see a `with` statement in a program, you should understand that the code inside it is running in a special context.

To set up this special environment or context and to tear it down, the `with` statement needs help. It needs someone to manage this context. We know that any object that satisfies the context management protocol can do this job and it is called a context manager. The context manager defines this runtime context. It controls what happens when the program execution enters and exits the context of a `with` statement.

The `with` statement sets up a runtime context and tears it down, with the help of a context manager object. Setup is performed when the context is entered. The `__enter__` method of context manager is responsible for the setup. It is executed when the execution enters the context of the `with` statement. Teardown is performed when the context is exited. The `__exit__` method of context manager is responsible for tearing down the context. It is executed when the execution leaves the context of the `with` statement. So, the `with` statement creates the runtime context, and the context manager defines the context.

21.6 Example: Sending output of a portion of code to a file

We have seen that we can implement a context manager by defining a class that follows the context management protocol. In this section and the next section, we will see two examples of creating a context manager class and using it in our code.

Here is a program in which we have written some random code:

```
print('Welcome')
numbers = [1, 2, 3, 4]
print(numbers)
print(numbers * 2)
for n in numbers:
    print(sum(range(1, n + 1)), end=' ')
print()
print([x * 100 for x in numbers])
d = {61: 'a', 32: 'b', 31: 'c'}
print(d)
print(sorted(d.keys()))
s1 = {'x', 'a', 'b'}
s2 = {'x', 'a', 'b', 'c', 'd'}
print(s1 | s2)
```

```
print(s2 - s1)
print(2 + 3 * 5)
print('Bye')
```

The output of this program prints on the screen by default; all the `print` function calls will show the output on the screen. Now, suppose for a particular portion of the program, we want the output to be sent to a file instead of the screen. To do this, we will have to change some setting before that portion of the code so that the output is redirected to the file, and after that portion of the program, we have to restore the original setting so that the output is again displayed on the screen. In the following code, we have made some changes so that in a portion of the program, the `print` calls send the output to the file.

```
import sys
print('Welcome')
numbers = [1, 2, 3, 4]
print(numbers)
print(numbers * 2)
print('Redirecting Output to test.txt.....')
f = open('test.txt', 'a')
original = sys.stdout
sys.stdout = f
for n in numbers:
    print(sum(range(1, n + 1)), end=' ')
print()
print([x * 100 for x in numbers])
sys.stdout = original
f.close()
print('Output prints to screen now.....')
d = {61: 'a', 32: 'b', 31: 'c'}
print(d)
print(sorted(d.keys()))
```

```

s1 = {'x', 'a', 'b'}
s2 = {'x', 'a', 'b', 'c', 'd'}
print(s1 | s2)
print(s2 - s1)
print(2 + 3 * 5)
print('Bye')

```

First, we open the file `test.txt` in append mode; the output will be redirected to this file. Then we save the original `stdout` file in variable `original` and then change `sys.stdout` to file object `f`. This change redirects the output of `print` to the file `test.txt`. To return to the normal setting, we change the `sys.stdout` to `original`, then close the file object.

When we run this program, we will see that the output of the portion of the program where we changed the setting did not print on screen. That output is sent to the file `test.txt`, which we can open and see.

Suppose we want to redirect the output of some other portions of code to the file. So, we will have to wrap those portions of code also with these pieces of pre and post code which change and restore the setting. We have to surround the portions of code with the same pair of blocks of code, so we can create a context manager class here. The pre and post-codes will go in the `__enter__` and `__exit__` methods respectively. Then, we can write the portions of code inside the `with` statement.

```

import sys
class OutputManager():
    def __enter__(self):
        print('Redirecting Output to
test.txt.....')
        self.f = open('test.txt', 'a')
        self.original = sys.stdout
        sys.stdout = self.f
    def __exit__(self, exc_type, exc_value,
exc_traceback):

```

```

        sys.stdout = self.original
        self.f.close()
        print('Output prints to screen now.....')
print('Welcome')
numbers = [1, 2, 3, 4]
print(numbers)
print(numbers * 2)
with OutputManager():
    for n in numbers:
        print(sum(range(1, n + 1)), end=' ')
    print()
    print([x * 100 for x in numbers])
d = {61: 'a', 32: 'b', 31: 'c'}
print(d)
print(sorted(d.keys()))
with OutputManager():
    s1 = {'x', 'a', 'b'}
    s2 = {'x', 'a', 'b', 'c', 'd'}
    print(s1 | s2)
    print(s2 - s1)
print(2 + 3 * 5)
print('Bye')

```

The output of the code that is inside the `with` statement will be sent to the `test.txt` file. Now, we also have the guarantee that the cleanup code will execute and close the file even if an exception occurs while executing the statements.

We can make a little update to this class so that we can send the output to any file instead of sending the output to the same file every time.

```

import sys
class OutputManager():

```



```

    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        print('Redirecting Output to ',
self.filename, '.....')
        self.f = open(self.filename, 'a')
        self.original = sys.stdout
        sys.stdout = self.f
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        sys.stdout = self.original
        self.f.close()
        print('Output prints to screen now.....')
print('Welcome')
numbers = [1, 2, 3, 4]
print(numbers)
print(numbers * 2)
with OutputManager('text.txt'):
    for n in numbers:
        print(sum(range(1, n + 1)), end=' ')
    print()
    print([x * 100 for x in numbers])
d = {61: 'a', 32: 'b', 31: 'c'}
print(d)
print(sorted(d.keys()))
with OutputManager('log.txt'):
    s1 = {'x', 'a', 'b'}
    s2 = {'x', 'a', 'b', 'c', 'd'}
    print(s1 | s2)
    print(s2 - s1)

```

```
print(2 + 3 * 5)
print('Bye')
```

Inside the `OutputManager` class, we have added the `__init__` method that takes a filename as argument and, we have added a new attribute named `filename`. Now, while opening the file and in the message, we will write `self.filename` instead of `test.txt`. In the `with` statement, we will send the names of the file as the argument. So, output of first portion will go to `test.txt` and output of the second portion will go to `log.txt`.

If we want, we can provide a default argument in the `__init__` method.

```
def __init__(self, filename='test.txt'):
    self.filename = filename
```

So, whenever we do not provide a filename while instantiating `OutputManager`, by default the output will go to `test.txt`.

```
with OutputManager():
    .....
    #Output goes to test.txt
    .....
```

21.7 Example : Finding time taken by a piece of code

In the following program, we want to find out the time taken by some portions of code. For this, we have to note the start time before the specified code and the end time after the code.

```
print('Welcome')
x = 999999
y = x**566999
numbers = [1,2,3,4]
print(numbers)
```

```

n = 141111
fact = 1
while n > 0:
    fact *= n
    n -= 1
x = 999999
y = 674444
z = x + 10000 ** y
print('Bye')

```

We can make a `Timer` class that follows the Context Management Protocol. In the `__enter__` method, we will note the start time, and in the `__exit__` method, we will note the end time, calculate the total time taken and display it.

```

from time import time

class Timer():
    def __enter__(self):
        print('Starting timer')
        self.start = time()

    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.end = time()
        print('Timer stopped')
        time_taken = self.end - self.start
        print('Time taken is', time_taken)

```

Now, we can place the portions of code that we want to time inside with statement.

```

print('Welcome')

```

```
with Timer():
    x = 999999
    y = x ** 566999
numbers = [1, 2, 3, 4]
print(numbers)
with Timer():
    n = 141111
    fact = 1
    while n > 0:
        fact *= n
        n -= 1
with Timer():
    x = 999999
    y = 674444
    z = x + 10000 ** y
print('Bye')
```

Output-

```
Welcome
Starting timer
Timer stopped
Time taken is 1.296593189239502
[1, 2, 3, 4]
Starting timer
Timer stopped
Time taken is 4.7801361083984375
Starting timer
Timer stopped
Time taken is 0.7810640335083008
Bye
```

21.8 Using context managers in the standard library

In our examples, we saw how to create classes that can be instantiated to get context manager objects, and these context manager objects can be used in the `with` statement. You do not always have to write classes to get context managers. There are context managers available in the standard library and in other libraries, and these context managers can also be used in the `with` statement. For example, the `open` function returns a context manager object, and so it can be used in the `with` statement. It is the most common use of the `with` statement.

```
with open('test.txt', 'w') as file:
    file.write('Hello')
```

This code opens the file named `test.txt`, writes data to it and then also closes it automatically. Even if an exception occurs while executing the code block, the file will be closed. We do not see any explicit call to `close` method here but the file is closed automatically when the `with` block finishes executing. This is because the call to file object's `close` method is there in the `__exit__` method of the context manager that is returned by the `open` function.

We have seen earlier that we can use the `try...finally` statement to get guaranteed execution of the file object's `close` method. If we do not write the `with` statement, we have to write the following code:

```
file = open('test.txt', 'w')
try:
    file.write('Hello!')
finally:
    file.close()
```

Operations on files are done frequently, so this type of code can be there at multiple places in our program. Every time we need to use a file, we repeat this pattern: open the file, perform the action and then close the file. Instead of writing `try...finally` every time, we can write the equivalent code by using a `with` statement. It provides a cleaner and less verbose alternative

to the `try...finally`. This is the most common use of `with` statement, and if you have done any file handling, you must have used this `with` statement in your programs, without knowing how this actually works. But now you can understand how it works behind the scenes.

We know that the `open` function returns a file object. If we call `dir` function for this file object, we can see the methods `__enter__` and `__exit__` which shows that this object is a context manager.

```
>>> f = open('test.txt', 'w')
>>> type(f)
<class '_io.TextIOWrapper'>
>>> dir(f)
[..... ' __enter__ ', ..... , ' __exit__ ', .....
```

Since the object returned by `open` function is a context manager, we can use it in the `with` statement.

```
with open('log.txt', 'w') as file:
    file.write('Hello')
```

The `__enter__` method returns the file object itself, and it is assigned to the variable that we place after the `as` keyword, so we get a handle to the file object. We can use this handle inside the `with` statement to call any of these methods or access any attribute. For example, here we have called the `write` method on the file object

There are some other built-in types that follow the context management protocol and can be used in the `with` statement. For example, in the `threading` module, we have some lock objects that can be used as context managers. The lock is acquired on entry and is released on exit even if an exception occurs. Another example of a context manager is in the `sqlite` module. The connection object in this module can be used as a context manager that automatically commits or rolls back open transactions.

In the `decimal` module, the `localcontext` function returns a context manager that can be used to change and restore the current decimal context. This can be used when we want to do some calculations with particular

settings. We can change the precision or rounding inside the `with` statement but it is restored back to normal, after the execution of `with` statement.

```
import decimal
print(decimal.Decimal('22') / decimal.Decimal('7'))
with decimal.localcontext() as ctx:
    ctx.prec = 40
    print(decimal.Decimal('22') /
decimal.Decimal('7'))
print(decimal.Decimal('22') / decimal.Decimal('7'))
with decimal.localcontext() as ctx:
    ctx.prec = 4
    print(decimal.Decimal('22') /
decimal.Decimal('7'))
```

Output-

```
3.142857142857142857142857143
3.142857142857142857142857142857142857143
3.142857142857142857142857143
3.143
```

So, you can use the readymade context managers available in Python and other third-party libraries, or you can create your own context managers by writing classes that follow the Context Management Protocol.

21.9 Nested with statements and multiple context Managers

There can be situations when we need to use more than one context manager together. In these situations, we can write nested `with` statements.

```
with expression1 as var1:
    with expression2 as var2:
        statements
```

Here, we have a `with` statement nested inside another `with` statement. For example, suppose we have to read input from a file and write the processed output to another file. We need two context managers, so we can write nested `with` statements. The `close` method for both file objects will be executed.

```
with open('data.txt', 'r') as f1:
    with open('new.txt', 'w') as f2:
        for line in f1:
            f2.write(line + '\n')
```

From Python 3.1 onwards, we have a more concise and readable syntax to write nested `with` statements. We can have multiple context managers in a single `with` statement.

```
with expression1 as var1, expression2 as var2:
    statements
```

This is the simpler equivalent form of the nested structure. Now, we can write our code that involved two files in a concise form.

```
with open('data.txt', 'r') as f1, open('new.txt',
    'w') as f2:
    for line in f1:
        f2.write(line + '\n')
```

If we had to write this code using the `try...finally` form, it would be this long:

```
f1 = open('data.txt', 'r')
f2 = open('new.txt', 'w')
try:
    for line in f1:
        f2.write(line + '\n')
finally:
```



```
f1.close()
f2.close()
```

We can write multiple context managers in multiple lines if they are surrounded by parentheses. Here is an example:

```
with (
    open('data.txt', 'r') as f1,
    open('new.txt', 'w') as f2,
    open('test1.txt', 'w') as f3,
):
    for line in f1:
        f2.write(line + '\n')
        f3.write(line + '\n')
```

Let us see another example that makes use of nested `with` statement. The following class `OpenReadOnly` gives a context manager similar to the one given by the `open` function but the difference is that the file is always opened in read mode.

```
class OpenReadOnly:
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        self.f = open(self.filename, 'r')
        return self.f
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.f.close()
```

Here is the `Timer` class that we have seen earlier:

```
from time import time
class Timer():
    def __enter__(self):
        print('Starting timer')
```

```

        self.start = time()
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.end = time()
        print('Timer stopped')
        time_taken = self.end - self.start
        print('Time taken is', time_taken)

```

Here is a with statement that uses context managers given by both these classes. We have written the short form of nested with statement.

```

with OpenReadOnly('test.txt') as file, Timer():
    print(file.read(), end='')

```

Output-

Starting timer

This is first line of the file.

This is second line of the file.

Timer stopped

Time taken is 0.003991603851318359

From the output, we can see that the file is opened in read mode, and the timer code is also executed.

21.10 Implementing a context manager by using a decorator on a generator

We can use either predefined context managers (like the file object) or create our own context managers. There are two ways to create our own context managers. They can be implemented either by writing classes or by writing decorated generator functions. We have already seen how to write classes that can be instantiated to give context managers; in this section, we will see how to get context managers by using the second approach. This approach is

simpler, but you need to have a basic knowledge of decorators and generators to implement it.

The `contextlib` module of the standard Python library contains a decorator called `contextmanager`. This decorator can be used on a generator function to create a factory of context managers. Let us see how this works.

We have defined a function; it contains the `yield` keyword, which makes it a generator function.

```
>>> def manager():
...     print('Entering')
...     yield
...     print('Exiting')
```

When we call this generator function, we get a generator object.

```
>>> x = manager()
>>> type(x)
<class 'generator'>
>>> dir(x)
[....., '__iter__', ....., '__next__', ..... ]
```

`x` is a generator object, as we can see the `__iter__` and `__next__` methods when `dir` function is used on it.

If we apply the `contextmanager` decorator to the generator function, then it will give us a context manager that has the `__enter__` and `__exit__` methods.

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def manager():
...     print('Entering')
...     yield
```

```
>>> x = manager()
>>> type(x)
<class 'contextlib._GeneratorContextManager'>
>>> dir(x)
[....., '__enter__', ..... , '__exit__', .....]
```

Now the object that we get does not support the `__iter__` and `__next__` methods, but we can see the `__enter__` and `__exit__` methods, so it can be used in a `with` statement. In the following code, we have written two simple `with` statements that make use of the context manager returned by this function:

```
from contextlib import contextmanager
@contextmanager
def manager():
    print('Entering')
    yield
    print('Exiting')
with manager():
    print('xxx')
    print('yyy')
print()
with manager():
    print('Python')
    print(4 + 15 / 3)
    print('Runtime')
    print('Context')
```

Output-

```
Entering
```

xxx

yyy

Exiting

Entering

Python

9.0

Runtime

Context

Exiting

Let us see how this works. When the `with` statement is executed, first the code before the `yield` statement is executed, then when the `yield` statement is encountered. The execution of the function is stopped temporarily, and the control is transferred to the `with` code block. The `with` code block is executed, and when it finishes executing, the control returns to the function and whatever is there after the `yield`, is executed.

Whatever we write before the `yield` statement executes before the execution of `with` code block, so we can write our setup code there.

Whatever we write after the `yield` statement executes after the execution of `with` code block, so the teardown code can be written after `yield`. So before `yield`, we will write the code that would have gone in the `__enter__` method had we written a class, and after the `yield` we would write the code that would have gone in the `__exit__` method.

We have seen that we can have an optional `as` keyword in the `with` statement. So, suppose we have the `as` keyword and a variable after it in our `with` statement.

```
with manager() as var:
```

```
    print('xxx')
```

```
    print('yyy')
```

```
    print(var)
```

We know that we can use the variable `var` in the `with` code block to interact with the context. We have printed it inside the `with` code block. When we run our code with this `with` statement, we will see that `None` is printed for the variable `var`.

Now, let us change our `manager` function. If we write the `yield` keyword with a value after it, then that value will be bound to the variable placed after the `as` keyword in the `with` statement. This is equivalent to returning a value from the `__enter__` method, if you were writing a class.

```
@contextmanager
def manager():
    print('Entering')
    x = 5
    yield x
    print('Exiting')
```

Here, we have specified variable `x` in the `yield` statement, so now the variable `var` in our `with` statement will be bound to this variable `x`. Now when we run our previous `with` statement, we get 5 printed for variable `var`.

So, if you want your function to give out a value that can be assigned to the target variable in the `as` clause, then instead of writing a plain `yield`, make your `yield` produce a value.

Now, let us see what happens when an exception is raised inside the `with` code block.

```
from contextlib import contextmanager
@contextmanager
def manager():
    print('Entering')
    yield
    print('Exiting')
```

```
with manager():
    print('Python')
    print(4 + 15 / 0)
    print('Runtime')
    print('Context')
```

Output-

Entering

Python

Traceback (most recent call last):

```
File
"E:\Deepali\Programs\21_ContextManager\P21_20.py",
line 11, in <module>
    print(4 + 15 / 0)
```

ZeroDivisionError: division by zero

A `ZeroDivisionError` will be raised and since it is not handled, it is passed to the `yield` expression. The `yield` statement reraises the exception, but there is no error handling, so we will have abnormal termination and the teardown code will not be executed. The program is abnormally terminated.

Thus, the proper way to write the generator function is to enclose the `yield` statement in a `try` block and write the teardown code in the `finally` block. This ensures the execution of teardown code even when an exception occurs.

```
from contextlib import contextmanager

@contextmanager
def manager():
    print('Entering')
    x = 5
```

```
    try:
        yield x
    finally:
        print('Exiting')
with manager():
    print('Python')
    print(4 + 15 / 0)
    print('Runtime')
    print('Context')
```

Output-

Entering

Python

Exiting

Traceback (most recent call last):

```
File
"E:\Deepali\Programs\21_ContextManager\P21_21.py",
line 14, in <module>
    print(4 + 15 / 0)
```

ZeroDivisionError: division by zero

The teardown code is now inside finally block so it is executed even when an exception occurred in the with block. The program was terminated because the exception could not find an appropriate handler.

Whenever an unhandled exception occurs in the with code block, it is reraised by yield inside the generator. If we want to handle the exception, we can write except blocks inside the generators which can handle the exception and suppress it, or we can just partially handle the exception and then reraise it again. If the exception is totally handled and suppressed inside the generator, then the execution will resume with the statement that immediately follows the with statement in which the exception occurred.

Here is the proper format for writing a decorated generator function that gives us a context manager:

```
from contextlib import contextmanager
@contextmanager
def manager():
    SET UP CODE
    try:
        yield x
    finally:
        TEARDOWN CODE
```

In the beginning of the function, we write the setup code. Inside try block we write the `yield` expression, the value produced by `yield` will be assigned to the target variable in the `as` clause. In the `finally` block, we write the teardown code. There should be only one `yield` statement inside the function and at this point the `with` code block execution starts.

Now, let us see a few examples. We had created a class `OutputManager` for redirecting output of some parts of code to a file instead of the screen.

```
import sys
class OutputManager():
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        print('Redirecting Output to ',
self.filename, '.....')
        self.f = open(self.filename, 'a')
        self.original = sys.stdout
        sys.stdout = self.f
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        sys.stdout = self.original
        self.f.close()
```

```
print('Output prints to screen now.....')
```

Now, we will use the contextmanager decorator for getting the desired results.

```
import sys
from contextlib import contextmanager
@contextmanager
def output_manager(filename):
    print('Redirecting Output to', filename,
          '.....')
    f = open(filename, 'a')
    original = sys.stdout
    sys.stdout = f
    try:
        yield
    finally:
        sys.stdout = original
        f.close()
        print('Output prints to screen now.....')
print('Welcome')
numbers = [1, 2, 3, 4]
print(numbers)
print(numbers * 2)
with output_manager('test.txt'):
    for n in numbers:
        print(sum(range(1, n + 1)), end=' ')
    print()
    print([x * 100 for x in numbers])
d = {61: 'a', 32: 'b', 31: 'c'}
print(d)
print(sorted(d.keys()))
```

```

with output_manager('log.txt'):
    s1 = {'x', 'a', 'b'}
    s2 = {'x', 'a', 'b', 'c', 'd'}
    print(s1 | s2)
    print(s2 - s1)
print(2 + 3 * 5)
print('Bye')

```

Output-

```

Welcome
[1, 2, 3, 4]
[1, 2, 3, 4, 1, 2, 3, 4]
Redirecting Output to test.txt .....
Output prints to screen now.....
{61: 'a', 32: 'b', 31: 'c'}
[31, 32, 61]
Redirecting Output to log.txt .....
Output prints to screen now.....
17
Bye

```

The `__enter__` method in our class was not returning anything so we have written just plain `yield` in our function.

Here is another example. We had written the class `OpenReadOnly` that gives a context manager similar to the one given by the `open` function but the difference is that the file is always opened in read mode.

```

class OpenReadOnly:
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        self.f = open(self.filename, 'r')
        return self.f

```

```
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.f.close()
```

Now, we will write a generator function named `open_readonly`, which will take a filename as an argument and that file will be opened in read mode.

```
# open a file only for reading
from contextlib import contextmanager
@contextmanager
def open_readonly(filename):
    file = open(filename, 'r')
    try:
        yield file
    finally:
        file.close()
with open_readonly('test.txt') as f:
    print(f.read())
```

The file will be opened in read mode, so any write attempt to write to this file will result in an exception. The file will be automatically closed when the `with` block exits, even if it is due to an exception.

So, we have seen both the methods of creating context manager; by writing classes and by writing decorated generator functions. Both these methods are equivalent, you can implement whichever you find convenient. For implementing short and simple context managers, the generator approach is preferred as writing a class would be overkill.

Exercise

```
1.with expression as var:
    statements
```

The context manager returned by the expression is assigned to variable `var` that is present after the `as` keyword.

(A) True (B) False

2. Which are the two methods that an object needs to support to satisfy the context management protocol?

(A) `__init__` , `__exit__`

(B) `__enter__` , `__exit__`

(C) `enter`, `exit`

3. The `__enter__` method should always return `self`.

(A) True (B) False

4. How many parameters are there in the definitions of `__enter__` and `__exit__` methods. (including self)?

(A) 1,3

(B) 3,1

(C) 1,4

5. The `__exit__` method of the context manager will not be executed if an exception occurs inside the `with` code block.

(A) True (B) False

6. If nothing is returned from `__exit__`, then the exception is suppressed and is not propagated up.

(A) True (B) False

7. The `with` statement allows us to add special code before or after a piece of code. To implement this, it needs a _____.

(A) decorator

(B) generator

(C) context manager

8. The statements in the `with` block are running in a specific runtime context which is set up by _____ and torn down by _____

(A) `__init__` , `__exit__`

(B) `__enter__` , `exit__`

9. Which error will be raised if you use an object in the `with` statement that has no `__enter__` method?

(A) `TypeError`

(B) `AttributeError`

(C) `ValueError`

10. The _____ code is written before the `yield` statement and the _____ code is written after the `yield` statement in the generator function decorated by `contextmanager` decorator.

(A) `setup`, `teardown` (B) `teardown`, `setup`

11. Will the following function give a generator object?

```
@contextmanager
def manager():
    print('Entering')
    yield
    print('Exiting')
```

12. Which one is written with correct syntax?

(A) `with open('data.txt', 'r') as f1, with open('new.txt', 'w') as f2:`

```
    for line in f1:
        f2.write(line + '\n')
```

(B) `with open('data.txt', 'r') as f1, open('new.txt', 'w') as f2:`

```
    for line in f1:
        f2.write(line + '\n')
```

What is the output of the code given in questions 13 and 14?

13. `from contextlib import contextmanager`

`@contextmanager`

`def manager():`

`print('Entering', end = ' ')`

`yield`

`print('Exiting')`

`with manager() as var:`

`print(var, end = ' ')`

14. `from contextlib import contextmanager`

`@contextmanager`

`def manager():`

`print('Entering', end = ' ')`

`v = 100`

`yield v`

`print('Exiting')`

`with manager() as var:`

`print(var, end = ' ')`

15. Where should the yield statement be written?

`from contextlib import contextmanager`

`@contextmanager`

`def manager():`

`____ (A) ____`

`try:`

`____ (B) ____`

`finally:`

____(C)____

16. What is the output of the following?

```
class CM():
    def __init__(self,name):
        print('__init__called')
        self.name = name
    def __enter__(self):
        print('__enter__called')
        return self
    def __exit__(self, exc_type, exc_value,
exc_traceback):
        print('__exit__called')
    def f(self):
        print('f called')
with CM('ABC') as c:
    print(c.name)
    print(10 % 2)
    c.f()
with CM('LMN') as c:
    print(c.name)
    print(10 % 0)
    c.f()
```

17. Change the `Timer` class made in the chapter so that a warning message is displayed if the code in the `with` code block takes more than a specified number of seconds to execute. The number of seconds that should be exceeded for the warning to be displayed, should be given in the argument.

18. The `open` function takes a filename and a mode as arguments and gives us a context manager. Write a class that will give a context manager similar to the one given by the `open` function but the difference should be that the file should be opened always in read mode.
19. Write a class named `Indenter` that implements a context manager which is used to indent the output.

```
print('Welcome')

with Indenter(4): # output indented by 4
    spaces
        print('Hello')
        L = [1,2]
        print(L)
print('Python')

with Indenter(8): # output indented by 8
    spaces
        print('Hi')
        x = 5
        y = 7
        print(x + y)

with Indenter(): # By default, output
    indented by 2 spaces
        print('Programming')
print('Bye')
```

Output-

```
Welcome
    Hello
    [1, 2]
```

Python

Hi

12

Programming

Bye

[Hint : Change the print function method temporarily and then restore it]

20. Write a class named Repeater that implements a context manager which is used to print anything twice.

```
print("Yes ", "No", 1, 2)
print(2+3)
L = [1,2,3]
print(L)
with Repeater():
    print("Yes ", "No", 1, 2)
    print(2+3)
    L = [1,2,3]
    print(L)
print("Yes ", "No", 1, 2)
```

Output-

```
Yes  No 1 2
5
[1, 2, 3]
Yes Yes    NoNo  11  22
55
[1, 2, 3][1, 2, 3]
```

Yes No 1 2

Note that with Repeater everything is printed twice, even the newline.

[Hint : Change the `sys.stdout.write` method temporarily and then restore it]

21. We saw the following `Timer` class in the chapter. Implement the same context manager using `contextlib` instead of writing a class.

```
from time import time

class Timer():
    def __enter__(self):
        print('Starting timer')
        self.start = time()

    def __exit__(self, exc_type, exc_value,
exc_traceback):
        self.end = time()
        print('Timer stopped')
        time_taken = self.end - self.start
        print('Time taken is', time_taken)
```

22. Rewrite this code using a single with statement.

```
with open('data.txt', 'r') as f1:
    with open('new.txt', 'w') as f2:
        for line in f1:
            f2.write(line + '\n')
```

23. Rewrite the code inside the following function using a single with statement.

```
def file_compare(file1, file2):
    try:
```

```

f1 = open('test.txt', 'r')
f2 = open('test1.txt', 'r')
line_number = 1
while True:
    line1 = f1.readline().strip()
    line2 = f2.readline().strip()
    if line1 != line2:
        print('Line : ',
line_number)
        print(file1, line1)
        print(file2, line2)
    if line1 == '' and line2 == '':
        break
    line_number += 1
finally:
    f1.close()
    f2.close()

```

```

file_compare('test.txt', 'test1.txt')

```

24. Write a decorated generator function named `timed_open` using the `contextmanager` decorator from the `contextlib` module. It should work like the `open` function, and in addition, it should calculate the time elapsed also.
25. Write a few sample with statements for the following context manager.

```

class File():

```

```

    def __init__(self, filename='test.txt',
mode='a'):

        self.filename = filename

        self.mode = mode

    def __enter__(self):

        self.file = open(self.filename,
self.mode)

        return self.file

    def __exit__(self, exc_type, exc_value,
exc_traceback):

        self.file.close()

```

26. What is wrong with the following code?

```

class FileWriteOnly:

    def __init__(self, filename='test.txt'):

        self.filename = filename

    def __enter__(self):

        self.f = open(self.filename, 'w')

        return self.f

    def __exit__(self, exc_type, exc_value,
exc_traceback):

        self.f.close()

with FileWriteOnly as f:

    f.write('Hello')

```

27. In Exercise 20, you wrote a Repeater class to create a context manager used to print anything twice. Implement the same context manager using the context manager from the contextlib module.

28. What is the purpose of the following class ListProtector that implements a context manager?

```

class ListProtector:
    def __init__(self, original_list):
        self.original = original_list
    def __enter__(self):
        self.copy_of_list =
self.original.copy()
        return self.copy_of_list
def __exit__(self, exc_type, exc_val, exc_tb):
    if exc_type is None:
        self.original[:] =
self.copy_of_list
    else:
        print('Error while working on
the list')
        print('Any changes to the list
are discarded.')
    return True
mylist = [10, 20, 30]
with ListProtector(mylist) as working_mylist:
    working_mylist.append(40)
    working_mylist.append(20 + 100)
print(mylist)
with ListProtector(mylist) as working_mylist:
    working_mylist.append(60)
    working_mylist.append(34)
    working_mylist.append(20 / 0)
print(mylist)

```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Solutions

Solutions to programming problems are available in the source code provided with the book.

Chapter 2: Getting Started

1. (B) None is a keyword in Python
2. (C)
3. (C)
4. (A)
5. (C)
6. (B) Keywords True, False and None are in title case
7. (C)
8. (B) An integer literal starting with 0o is in octal base, but 9 is not an octal digit.
9. (D)
10. (B) Anything enclosed between quotes is a string literal
11. (B)
12. (A)
13. (C)
14. (B)
15. (B)

- 16. (A)
- 17. (B)
- 18. (B)
- 19. (B)
- 20. (D)
- 21. (B)
- 22. (C)
- 23. (A)
- 24. (A)
- 25. (A)
- 26. (A)
- 27. (A)
- 28. (C)
- 29. (A)
- 30. (A)
- 31. (B)
- 32. (B)
- 33. (C)
- 34. (B)
- 35. (C)
- 36. (C)
- 37. (D)
- 38. (B)
- 39. (D) Commas are not allowed in integers, so print takes it as three integers.
- 40. (A)
- 41. (C) 3 is converted to 3.0

42. 17.5 15.0 17 In the expression `int(3.5)/0.2`, first 3.5 is converted to int value, and then divided by 0.2. In the expression `int(3.5/0.2)`, first 3.5 is divided by 0.2 and then converted to int.

43. (B) The result exceeds the range of `float`, this condition is arithmetic overflow.

44. (B) The result is very small, it is not in the range of `float`, this condition is arithmetic underflow.

45. (B)

46. (A)

47. (A)

48. (B)

49. `print('My name is', name, 'and age is', age)`

50. 3 3 -4 -3

The `int` function removes the decimal part from the `float` and hence always rounds towards zero. The floor division operator(`//`) rounds towards minus infinity. So, if you want the truncated value for both positive and negative integers, use `int()` function instead of `//` operator.

51. True

52. Syntax Error

53. 2 2

54. 20

55. 2 4 Expressions `x + 4` and `y + 5` are evaluated and `int` objects for values 6 and 9 are created but they are not assigned to anything, so the two statements have no effect.

56. 14.0

- 57. Syntax error Cannot assign to an expression, left side of assignment statement has to be a name.
- 58. SyntaxError: leading zeros in decimal integer literals are not permitted
- 59. Syntax error : there is no operator `=<` , use `<=`
- 60. Syntax error : `raise` is a keyword
- 61. Indentation error
- 62. `Hello,Hi,Hey,`
- 63. `True`
- 64. `92 -92` Inserting a +ve sign before a number, has no effect. Inserting a – sign makes the number negative.
- 65. This code will print Hello world and then show TypeError.

In the second statement, we have assigned to the name `print`. There was no syntax error as `print` is not a keyword, it is a built-in function name. When the statement `print(2 + 5)` was executed, `print` was referring to an `int` object, it was no longer a function and that is why we get TypeError: 'int' object is not callable.
- 66. `2555 n1` and `n2` refer to `str` objects, for integer arithmetic we need to convert the input to `int` type.

Chapter 3: Strings

- 1. (B) To get the last character we need to write `s[len(s)-1]`
- 2. (B)
- 3. (B)
- 4. (A)
- 5. (C)
- 6. (B) There is no character type in Python
- 7. (B)

- 8. (C)
- 9. (C) Strings are immutable, so can't do this
- 10. (C) `len()` is a built-in function not a `str` method, so should not be called with dot syntax.
- 11. (B)
- 12. (C)
- 13. (B) 3 Searches for 'n' in the last 10 characters
- 14. (B)
- 15. (B)
- 16. (A)
- 17. (B)
- 18. (B)
- 19. (D)
- 20. (A)
- 21. (C)
- 22. (A)
- 23. (D)
- 24. (B)
- 25. (A)
- 26. (C)
- 27. (B)
- 28. (B)
- 29. (C)
- 30. (B)
- 31. `s[:5]`
- 32. `s[-5:]`
- 33. `s[4]`

```
34. s[-1]
35. s[::-1]
36. s[:-1]
37. s[:-5]
38. s[5:]
39. IndexError
40. IndexError
41. 'are easy, execution is hard.'
42. 'Ideas'
43. s1 = s[:]
44. s2 = s[:-3]
45. Empty String
46. s[4:15:2]
47. s = s[-1] + s[1:-1] + s[0]
48. s3 = s1[-4:] + s2[:3]
49. s[:2] * 5 + s[2:-1] + s[-1] * 3
50. email_id = input('Enter an email id - ')
    username = email_id[email_id.index('@')]
    domain_name = email_id[email_id.index('@') +
1:]
    print(username)
    print(domain_name)
```

```

51.s1 = s[ s.index('*')+1 : s.rindex('*') ]
52.s = s.strip().title()
53.s = s.replace('he', 'she').replace('that',
    'this', 3)

54.s = s[:len(s)//2].upper() +
    s[len(s)//2:].lower()
55.s.startswith('Line') and s.endswith('Done')

56.code = name[:8:2] + dob[:2] + dob[-2:] +
    city[:3]

57.print('-' * 80)

58.print('\n' * 5, end = '')

59.r = int(str(n)[::-1])
60.No

    >>> s = ' Python '
    >>> s.rjust(20, '-').strip()
    '----- Python'
    >>> s.strip().rjust(20, '-')
    '-----Python'

61.h h

62.1 14 4

63.print('Hello', end = ':\n')

64.caattt's curiosity killed the
    's curiosity killed the

```

Chapter 4: Lists and Tuples

1. `TypeError` - Only `int` type is allowed for index
2. `[1, 2, 3, 100]`
3. `[6, 8, 10, 12]`
4. `IndexError: list assignment index out of range`
5. `[1, 2, 10, 20, 30, 40, 50, 5, 6, 7, 8, 9]`
6. `[1, 2, 3, 6, 7, 8, 9]`
7. `[1, 2, 3, [], 5, 6, 7, 8, 9]`
8. `[]`
9. `[1, 2, 3, 'a', 'b', 'c', 'd', 4, 5]`
10. `True`
11. `[1, 2, 3, 1, 2, 3, 1, 2, 3]`
12. `TypeError`
13. `['', '', '']`
14. `['W', 'e', 'l', 'c', 'o', 'm', 'e']`
15. `[0, 1, 2, 3, 4]`
16. `[]`
17. `[3, 6, 9, 12]`
18. `['Thor', 'Iron man', 'Hulk', 'Ant-Man']`
19. `['ab', 'cd', 'de', 'fg-hi-jk']` The second argument limits the number of splits.
20. `[10, [1, 2, 3], 20]`

21. [1, 2, 3]
22. [['x'], ['x'], ['x']]
23. a
24. 5
25. [0, 45, 0, 0, 0]
26. [[0, 4, 5, 9], [0, 4, 5, 9], [0, 4, 5, 9], [0, 4, 5, 9]]
27. [[2, 6, 11], [1, 5, 9], [2, 6, 11], [5, 9, 1], [2, 6, 11], [5, 9, 1]]
28. (6, 7, 8, 6, 7, 8)
29. ValueError: too many values to unpack (expected 3)
30. 1 3 [4, 5, 6, 7, 8]
31. [4, 5, 9, 10] [4, 5, [], 10]
32. None None Methods append and sort return None. The objects are changed in place so there is no reason to reassign.
33. 6 7
34. [[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
35. 09-08-1973
36. <class 'tuple'>
37. <class 'str'> <class 'tuple'>
38. ValueError: not enough values to unpack (expected 3, got 2)
39. 3

40. 3, 4, 5]

41. [1, 2, 3, 4] [4, 3, 2, 1]

42. True

43. 40

44. (1, 2, 3) 1 2 3

45. [1, 2, 3, 4, []] [1, 2, 3, 4]

46. [2, 3, 5, 1, 6, 8, 9] [1, 2, 3, 5, 6, 8, 9]

47. L1 += 100 will give TypeError, we need to write L1 += [100]

48. [9, 11]

49. (D)

50. (C)

51. (C)

52. (B)

53. (B)

54. (C)

55. (C)

56. (A)

57. (B)

58. (C)

59. (B)

60. numbers [-2] = 200

61. numbers [2:6] = [30, 40, 50, 60, 70, 80]

62. numbers[3:] = 'pqr'

```
63. numbers [5:5] = [10, 20, 30, 40, 50]
64. numbers [2:6] = []
65. cpy = numbers[:]
66. rev = numbers [::-1]
67. numbers.append(100)
68. numbers.insert(0, 200)
69. numbers.insert(3, 150)
70. numbers.extend([12, 13, 14, 15])
71. numbers.remove(5)
72. numbers.pop()
73. x = numbers.pop(5)
74. numbers.pop(0)
75. numbers.clear()
76. del numbers[5]
77. del numbers[-3:]
78. numbers.count(55)
79. numbers.index(55)
80. len(numbers) - numbers[::-1].index(55) - 1
81. numbers.index(55, 4, 10)
82. numbers.index(min(numbers))
```

```
83. numbers[numbers.index(max(numbers))] = 1000
84. Second largest -- sorted(numbers)[-2]
    Third smallest -- sorted(numbers)[2]

85. list1 = sorted(numbers)[-3:]

86. sum(sorted(numbers)[:5])

87. x = min(numbers[:len(numbers)//2])

88. avg = sum(numbers) / len(numbers)

89. newlist = sorted(numbers)[-5:]

90. newlist = sorted(numbers)[:5]

91. numbers.sort(reverse = True)

92. newlist = sorted(numbers)

93. fruits.sort(key=len)

94. fruits.sort(key=str.lower)

95. listA = [None] * 20

96. list(range(1000, 0, -100))

97. list(range(56, 150, 7))

98. disney_characters = ','.join(listD)

99. fruits[2][::-1]

100. ('John', 25, [88, 90, 92, 89, 98])

101. [100, 2, 3]
```

To avoid the side effect, instead of `X = ['a', L]`, write any of these.

```
X = ['a', L[:]]      X = ['a', L.copy()]
X = ['a', list(L)]
```

102. `x = 3`

`y = 2`

`x, y = y * x, x`

`print(x, y)`

103. `L1 = L1 + L2` creates a new list object while `L1.extend(L2)` changes the list in-place. In-place changes are generally more efficient than making a new object.

104. `listA.clear()` and `del listA[:]` make the list empty, they change the list object in-place. `del listA` will undefine the name `listA`. After `del listA`, if you reference the name `listA`, `NameError` will be raised.

105. `L1 = L.sort()` will sort the values in the list object referenced by `L`, and will assign `None` to `L1`

`L1 = sorted(L)` will assign sorted list object to `L1`, the list object referenced by `L` won't be changed.

106. `L[:3] = []` will delete all elements from the start index till index 2, `L[3:] = []` will delete all elements from index 3 till last index. `L[3] = []` will replace the element at index 3 with `[]`.

107. `name, city, age, salary = employee`

108. `L = [1, 2, 3, 4, 5]`

`L[0], L[-1] = L[-1], L[0]`

```
109.colors = input('Enter 5 colours separated by  
dashes: ').split('-')
```

Chapter 5: Dictionaries and Sets

1. (C)
2. (B)
3. (A)
4. (B)
5. (B)
6. (A)
7. (B)
8. (B)
9. (B)
10. (B)
11. (B)
12. (B)
13. (A)
14. (C)
15. (B)
16. (B)
17. (B)
18. (B)
19. (B) Duplicate keys not allowed, so only one L is added as the key
20. (A)
21. (C)
22. (B) Set can contain only immutable types

- 23. (B)
- 24. (A) Because frozenset is immutable
- 25. (B)
- 26. (A)
- 27. (C) You should provide a single iterable object like string, list or tuple; cannot provide a list of argument values.
- 28. (D)
- 29. (A)
- 30. (C)
- 31. (B)
- 32. (D)
- 33. (B)
- 34. (C)
- 35. (D)
- 36. (B)
- 37. (B)
- 38. (A)
- 39. (A) Dictionaries also perform fast lookup like sets.
- 40. (B) Strings are immutable
- 41. (C) Sequences perform sequential searches while sets are optimized for very fast lookup. A set has highly optimized method for checking whether an element is present in the set.
- 42. (D)
- 43. 20
- 44. {'g': 0, 'o': 2, 'd': 3} Value for 'o' is overwritten
- 45. 33
- 46. TypeError: 'set' object is not subscriptable
- 47. False

```
48. True
49. {'a', 'c', 'b'}
50. {'hello'} {'l', 'e', 'o', 'h'}
51. {'a': [1, 55, 3], 'b': 99, 'c': 12}
52. {'a': [1, 55, 3], 'b': 10, 'c': 12}
53. 5

54. frozenset({'e', 'a', 'b', 'c', 'd'}) As in strings,
    the augmented assignment will not modify in-place, it will reassign.
    It is equivalent to x = x | y

55. 2 0 0 {'a': 2} {'a': 2} {'a': 2}

56. >>> currency = {}
    >>> currency['India'] = 'Rupee'

    >>> currency['UK'] = 'Pound'

    >>> currency['Japan'] = 'Yen'

    >>> currency['Austria'] = 'Euro'

    >>> currency['Bangladesh'] = 'Taka'

57. del currency['UK']

58. c = currency.pop('Japan')

59. currency['Switzerland'] = 'Swiss Franc'

60. currency['India'] = 'Indian Rupee'

61. currency.popitem()
```

```

62. list(currency.keys())
    list(currency.values())

    list(currency.items())

63. x = fruits_prices.setdefault('apple', 0)
    y = fruits_prices.setdefault('grapes', 0)

64. login = dict.fromkeys(names, None)

65. D = dict(zip(designation, salary))

66. books = {'python': python_books, 'c++':
    cplusplus_books, 'java': java_books}

67. book_prices.update(new_stock)

68. dict.fromkeys(range(1000, 10000, 1000), None)

69. student['name']['last']

70. L = sorted(d.keys())

71. sum(marks[2135])

72. matrix = {(0, 5): 4, (1, 3): 8, (3, 4): 6, (5, 2): 3}

```

To access value at row 0 and column 5, you can write `matrix[0,5]`. This syntax is different from the nested list representation of matrices, here we are using a tuple of 2 integers as index.

73. Use get method to access the elements-

```

matrix.get((1,3), 0)           returns 8
matrix.get((1,2), 0)           returns 0

```


In the `get()` method, first we send the key and then next we send the element that is returned when the key is not present in the dictionary.

```
74. s1 = input('Enter first string : ')
    s2 = input('Enter second string : ')

    L = list(set(s1) & set(s2))

75. set(string1.split(' ')) & set(string2.split(' '))

76. len(set(list1))

77. list(set(L)), No order will not be preserved

78. text = input('Enter some text : ')
    vowels = set('aeiou')

    consonants = set('bcdfghjklmnpqrstvwxyz')

    v = vowels & (set(text))

    c = consonants & (set(text))

79. set(L1) == set(L2)
    set(str1) == set(str2)

80. set(L1) - set(L2)

81. set(s1) & set(s2) & set(s3)
```

```
82. toppers.remove('id11')
```

```
83. toppers.add('id46')
```

```
    toppers.add('id20')
```

```
84. toppers - champions
```

```
85. champions - toppers
```

```
86. toppers & champions
```

```
87. toppers | champions
```

Chapter 6: Conditional Execution

1. Syntax error

2. 95 `NameError` will occur only when the control will reach the statement `bill = uniits * 1.5`

and it will reach there when value of `units` is greater than or equal to 100.

3. that For comparing with `None`, remove the quotes.

4. Hello

5. Good Evening

6. 20

7. 2

8. (B)

9. (A)

10. (D)

11. (C)

24.

```
(i) if grade != 'A':  
    print('Work Hard')  
(ii) if age >= 18:  
    print('You can vote')  
(iii) if n % 2 != 0:  
    print('n is odd')  
(iv) if marks <= 0 or marks > 100:  
    print('Out of range')  
(v) if age >= 18 and weight <= 60:  
    print('Allowed to play the game')
```

25. Boolean values True and False can act as integers in arithmetic operations. The equivalent integer value for True is 1 and for False is 0. So, the first statement prints 5 and the second one -3.

Chapter 7: Loops

1. Infinite

2. 0

3. Infinite

4. (i) 0 iterations (ii) 5-i iterations (iii) infinite

5. 4 3 2 1 0

6. 4, 8, 12, 3 After the `for` loop terminates, the control variable remains bound to the last value to which it was bound.

7. 2

8. 20

9. `abcabcabc`

10. `1 4 7`

11. `0 2 0 4`

12. `10 16 22 28`

13. `Be happy`

14. `10 11 12 13 14 15 16 17 18 19`

`break` should be written after `isprime=False` to get prime numbers

15. `x 6 x 7 y 6 y 7`

16. `1 2 4 5 7 8`

17. 14 Finds the sum of all the digits in number `n`

18. `[] [12, 9, 6, 3]`

19. `No senior citizens`

20. `Yes`

21. `Yes`

22. `(A)`

35. Problem is aliasing, names `evens` and `odds` are aliases to same list.

You need to initialize `evens` and `odds` separately.

```
evens = []
```

```
odds = []
```

53. `else` should be aligned with `for`, not with `if`.

55.

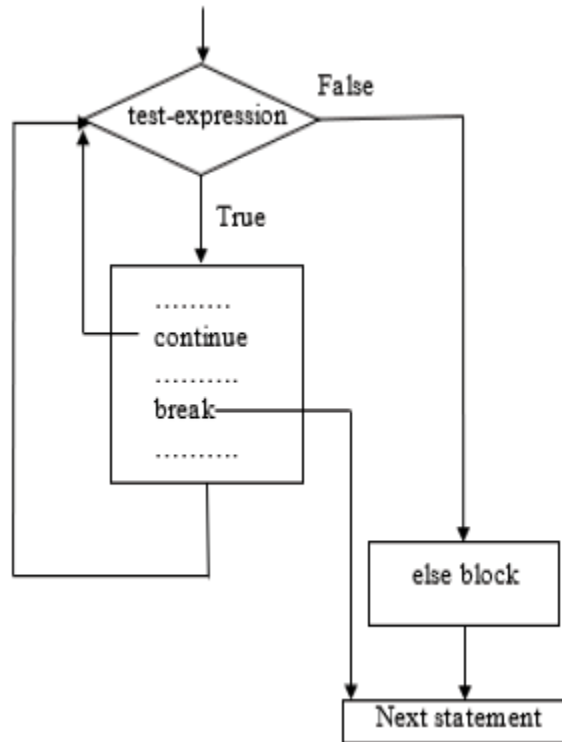


Figure 22.1

Chapter 8: Looping Techniques

1. 5 10 6 20 7 30

2. [2, 1, 4, 3, 6, 5]

3. IndexError

4. 10 20 30 40 50

5. 14 11 8 5

6. (2, 2) (3, 3) (4, 4)

7. ['yes', 'no', 'this']

8. [10, 11, 12, 13]

9. ['London', 'Paris', 'Noida', 'Perth', 'Rome',
 'New York', 'New Delhi']

10. Infinite Loop

11. [1, 2, 3, 1, 2, 3]

12. 6 5 3 2 1

13. [10, 23, 34, 90]

14. [12, 27, 35, 94]

15. Tom London Rob Paris

16. Infinite loop

17. Pam John Neil

20. This loop iterates over the list in reverse order. We can use the
reversed function instead.

31. You need to iterate over a copy of the list. A better approach could be
to use a while loop.

32. 0 1 | 1 2 | 2 -3 | 3 -5 | 4 8 | [1, 2, 4, -6,
8]

34. This approach does not change the original list in-place. It creates a
new object and makes L refer to it. If there are multiple references to
the original list, they will not be updated.

36. None of them will work correctly. In the first code snippet we are
modifying the list while iterating over it, and so it will not work. In
the second code snippet, the statement temp = names1 does not
create a separate copy, it just creates an alias, so we are iterating over
the original list in the for loop, and that is why it does not work. To
make it work we need to iterate over a copy of the list.

Chapter 9: Comprehensions

1. `[[1, 2, 3], [2, 4, 6], [3, 6, 9]]`
2. 30
3. `['W', 'i', 'y', 'n']`
4. `SyntaxError` There is no else clause in list comprehension
5. `[1, 10, 13, 56]`
6. `[2, 10, 6, 14, 18, 1, 4, 2]` Odd numbers of the list are multiplied by 2 and even numbers are divided by 2.
7. `[4, 5, 6, 8, 10, 12, 12, 15, 18]` `[4, 10, 18]`
8. `True`
9. `[-2, -4, -3, -7, 1, 21, 20, 9, 0]`
10. (C)
11. (A)
12. No, we will get the original dictionary back only when it contains immutable and unique values
13. (A)
14. `roots = [n ** 0.5 for n in L if n > 0]`
15. `import random`
`randoms = {random.randint(1, 1000) for _ in range(10)}`
16. `[str(num) + 'x' for num in range(5, 18, 2)]`
17. `L = [x * y for x, y in zip(X, Y)]`

18. In (A), the list comprehension creates and returns a new list object which is assigned to variable `names`. The new list contains all the names in title case. In (B), no new list object is created. The list is not changed because in each iteration, the changes are made to the loop variable `name`. In (C) also, no new list object is created but the list is changed, all the strings are titlecased. (A) and (C) give the same result, but if there are multiple references to the original list, then they will not be updated if we use (A).

47. The first one creates the list `[12, 15, 18, 24, 30, 36, 28, 35, 42]` and it is equivalent to the following for loop code.

```
L1 = []  
for x in [3, 6, 7]:  
    for y in [4, 5, 6]:  
        L1.append(x * y)
```

The second comprehension is a nested comprehension, and it creates the following list.

```
[[12, 24, 28], [15, 30, 35], [18, 36, 42]]
```

It is equivalent to the following code.

```
L2 = []  
for y in [4, 5, 6]:  
    temp = []  
    for x in [3, 6, 7]:  
        temp.append(x * y)  
    L2.append(temp)
```

48. This is the correct way-

```
board = [[' ' for i in range(3)] for j in  
range(3)]
```


Chapter 10: Functions

1. (B)
2. (B)
3. (B)
4. (A)
5. (A)
6. (C)
7. (B)
8. (B)
9. (A)
10. (A)
11. (A)
12. (A)
13. (B)
14. (B)
15. (B)
16. (B) For gathering keyword arguments, you need to use 2 asterisks
17. (D)
18. (C)
19. (D)
20. (A)
21. (A)
22. (D)
23. (C)
24. (B)
25. `continue` is a keyword, can't give the function a name that is a keyword

26. No, Default arguments should appear at the last
27. Yes
28. Yes
29. Yes
30. TypeError: func() got multiple values for argument 'a'
31. Yes, problem will occur when you call the function
32. It will give NameError. The function name does not exist until the control reaches def and runs the def statement. Before you call a function, you have to create it.
33. It will give NameError: x is a local variable and therefore can't be accessed outside the function.
34. (9, 24)
35. Hello None
36. 18
37. 14
38. {1: 'a', 2: 'b'} {}
39. [5, 6, 7, 8] (5, 6)
40. {1: 'a', 2: 'b', 3: 'c'}
41. [1, 2, 3, 4, 10]
42. [2, 4, 6, 8] [1, 3, 5, 7, 1, 3, 5, 7]
43. {1: 'a', 2: 'xxxx', 3: 'c'} {1: 11, 2: 22, 3: 33}
44. 10 1.5 [1, 2, 3] hello
45. 35 [1, 2, 3, 4] [10, 20, 30, 40, 100]

46. 10

47. (3, 4, 3, 4)

48. {10: 0} {10: 0, 20: 0}

49. ('a', 'b', 'c') A dictionary can also be unpacked using a single asterisk to get the keys

50. 2 3 {}

51. (1, 2, 3) {'x': 5, 'y': 10}

52. Hello

53. hello hello hello hello

54. TypeError: result() got multiple values for argument 'standard'

55. 4 6

(4, 6) 8

56. [1, 2, 1] [1, 2]

57. 9 64 5 This function computes an.

58. Hello Hello A call to function func will not give any error as long it is called with a number that is greater than or equal to 5. When func is called with a number less than 5 then only the interpreter will realise that there is no function named priiint and it will raise an error.

59. 3

60. <class 'int'> <class 'function'>

61. Hello Hi Hey Jack

62. 5 4 3 2 1

1 2 3 4 5

In `display1`, the `print` calls are executed in winding phase while in `display2` the `print` calls are executed in the unwinding phase. In unwinding phase, the calls return in reverse order.

63. `[[1, 7, 8], [6, 5, 9], [2, 6, 3], [3, 9, 2]]`

T is the transpose of the matrix M. Transpose of a matrix is a matrix in which rows and columns are interchanged; rows become columns and columns become rows. When we send `*M` to the `zip` function, the list M is unpacked and the inner lists are sent as arguments. So, the `zip` function gets the 3 inner lists as arguments.

64. No

77. The `grade` parameter is placed after `*args`, so it will accept keyword only argument. In the second call, the value `True` is sent by position so it is collected in the tuple `args`. The default value of `grade` is used which is `False`, so `grade` is not printed. Value of `True` is 1 so total is $90 + 90 + 90 + 1 = 271$

82. `5.5 5.5 [2, 3, 4, 5, 6, 6, 8, 9] [2, 4, 5, 8, 6, 6, 3, 9]`

The function `median1` changes the list in-place. To make sure that the list remains safe, send a copy of the list.

```
print(median1(nums1[:]))
```

91. Yes, they will work in the same way. The absence of `else` does not affect the correctness of the code. Using an `else` is not necessary because the `return` statement inside the `if` block effectively terminates the function when the condition is satisfied. In these coding examples, `else` is included for explicitness and clarity.

Chapter 11: Modules and Packages

1. C
2. B
3. B
4. B
5. B
6. A
7. B
8. B
9. A
10. B
11. B
12. C
13. B
14. B
15. B
16. A

Chapter 12: Scope

1. (B)
2. (B)
3. (B)
4. (A)
5. (A)
6. (B) Names are deleted, not objects. An object will be garbage collected only when its reference count drops to zero

- 7. (C)
- 8. (B)
- 9. 2
- 10. 10
- 11. `NameError` `f` can be called inside `func` only, it has scope local to `func`.
- 12. `NameError` Outer functions can't access local variables defined in inner functions.
- 13. 20 10
- 14. 20 20
- 15. The first call to `min` will work, but the second one will show error
- 16. `UnboundLocalError`
- 17. `NameError: name 'n' is not defined`
- 18. 5
- 19. `SyntaxError: no binding for nonlocal 'n' found`
- 20. `SyntaxError: no binding for nonlocal 'n' found` If a name is declared nonlocal, it is searched only in the enclosing function scopes.
- 21. 10 15 5
- 22. 10 10 10 5
- 23. 5
- 24. (B)
- 25. (D) Mutable objects like lists and dictionaries can be changed in place without global or nonlocal statement. Changing an object in-place is not the same as assignment to a name; assignment rebinds the name.

Chapter 13 Files

1. (A) In a+ mode, cursor is at the end of the file, so before reading you need to take it to the beginning.
2. (B)
3. (A)
4. (A)
5. (B)
6. (B)
7. (B)
8. (B)
9. (B)
10. (B)
11. (B)
12. (A)
21. The last line may not have a newline character at its end, and so the last character of the last line may be lost.
25. When the file is read second time, nothing is printed since after the first read the file cursor comes to the end of the file, We need to take it to the beginning by calling the `seek` method.

Chapter 14: Object Oriented Programming

1. False True
2. (A)
3. (B)
4. (B) You need to provide the parameter `self`
5. (B)
6. (B) Interpreter always provides the argument for `self`, so you never have to provide it.

7. The call to `method1` should be prefixed with `self` and a dot.
8. (C)
9. `AttributeError: 'Test' object has no attribute 'x'.`
`method1` was not called so instance variable `x` was not created for instance object `t`.
10. `UnboundLocalError`, `x` inside `method2` was not prefixed with `self`, so it is considered a local variable.
11. (A)
12. (B)
13. (B) class method always receives the class as the first argument, whether it is invoked through the class or through the instance.
14. (B)
15. (B)
16. No, in a class method, you can use any word instead of `cls`, it could be `self` also. But never do this as it is unconventional and confusing.
17. No
18. Yes
36.

```
class BankAccount:
    bank_name = 'ABC bank, XYZ Street, New Delhi'

    def __init__(self, name, balance=0,
bank=bank_name):
        self.name = name
        self.balance = balance
    self.bank = bank

    def display(self):
```



```

print(self.name, self.balance, self.bank)
def withdraw(self, amount):
    self.balance -= amount

    def deposit(self, amount):
        self.balance += amount

a1 = BankAccount('Mike', 200, 'PQR Bank
Delhi')
a2 = BankAccount('Tom')
a1.display()
a2.display()

```

Note that here we used the class variable `bank_name` without preceding it with the class name.

Any statements that are written inside the class methods, have to use the fully qualified class variable name, for example `MyClass.x`.

Any statement that is at the class level, i.e. outside the class methods should use simply the variable name, not the fully qualified name.

For example -

```

class MyClass():
    x = 6
    y = x + 10

```

Here you need to use it as `x`, not as `MyClass.x`

Chapter 15: Magic Methods

```

4. def __lt__(self, other):
    return self.age < other.age
5. 10 < 15

```

Chapter 16: Inheritance and Polymorphism

2. Can cook noodles

Can cook pasta

Can cook butter chicken

3. I am a Person

I am a Student

I am a Teaching Assistant

4. 2675T

4567S

3421T

5749

Using the base class names can cause bugs in multiple inheritance. If we use `super`, the problem will not occur.

5. Base : method1

Base : method2

Base : method3

Base : method1

Derived : method2

Base : method3

Derived : method3

method1 is implicitly available in Derived class. method2 is overridden in Derived. method3 of Derived class uses method3 of Base and has its own code also.

Chapter 17: Iterators and Generators

1. (B)
2. (A)
3. (A)
4. (B)
5. (A)
6. (A)
7. (B)
8. (A)
9. (C) Since each iterator is an iterable
10. (A)
11. (B)
12. (C)
13. 1 2 3 0
14. 3 7
15. 1 2 3 0 0 Iterator does not have its own copy of the elements, if the iterable is changed, then the iterator will get the updated element.
16. [(1, 'a'), (2, 'b')] () {}
17. 1 2 3
18. 1 2 When return statement is executed inside a generator function, it raises StopIteration exception ending the iteration.
19. 1 2 3 4 5
20. 1 3 5 7 9 11 13 15 17 19
100

1 3 5 7 9 11 13 15 17 19

It is because of the statement `self.num = 1` in `__iter__`. In any iteration context the `iter` function calls `__iter__` method, and in this `__iter__` method, `self.num` is assigned 1, so in each iteration context `self.num` starts with 1.

21. 2 4 8 16 32
62

This class implements an iterator that gives powers of two.

22. (B)

36. By removing the square brackets we can change the list comprehension to a generator expression.

```
dot_product = sum(a * b for a, b in zip(L1, L2))
```

40. 284

4 16 36 64
284

The function `func` is iterating over data two times. When the argument sent is an iterator, it exhausts in the first scan only and so the next scan does not work and there is no error also.

We can change the function so that it raises an error if an iterator is sent as the argument.

```
def func(data):  
    if iter(data) is data:  
        raise TypeError('This function does  
not work with iterators')  
    print(sum(data))
```

```

for i in data:
    if i % 2 == 0:
        print(i, end=' ')
print()

```

or we can turn the argument into a list

```

def func(data):
    data = list(data)
    print(sum(data))
    for i in data:
        if i % 2 == 0:
            print(i, end=' ')
    print()

```

or we can make an iterable class instead of the generator function.

```

class GetSquares:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
    def __iter__(self):
        i = self.start
        while i <= self.stop:
            yield i * i
            i += 1

g = GetSquares(2, 9)

```

45. Both will give same output, first one uses a list comprehension and the second one uses a generator expression. List comprehension will compute entire list and send it to the `sum` function. The list of values is used just for the intermediate step, we don't need a list as the result, so in this case it better to use a generator expression. In the second statement, the `sum` function works on the generator object returned by the generator expression.
46. The first loop will load the whole file in a list which can be problematic if the file is too large. The second one reads the file line by line, since it works on the iterator returned by the `open` function.

Chapter 18: Decorators

1. `NameError: name 'g' is not defined`
2. `Hello Welcome to Python`
3. `Learning Decorators @`
4. `None`
5. `('Welcome', 'to', 'Python')`
6. `Hello Hello Welcome to Python`
7. `1 Hello. 2 Hello. 3 Hello.`
8. `False`
9. `(B)`
16. As in previous question, this decorator will also be a simple decorator that does not require any wrapper and will just return the original function after adding the new attribute.

Chapter 19: Lambda Expressions and Functional Programming

1. (A)
2. (A)
3. (B) They can access variables in enclosing, global and built in scope as well.
4. (A)
5. (B)
6. (B)
7. (C)
8. (B)
9. Yes
10. Yes
11. 15
12. [0, 4, 8, 12, 16]
13. Gives Error . If you want to return a tuple from a lambda, you need to place the parentheses

```
(lambda x, y: (x + y, x - y))(7, 3)
```
14. [2, 3, 4, -22, 32, -44]
15. [(1, 'one'), (2, 'two'), (4, 'four'), (5, 'five'), (3, 'three')]
16. [('s', 'p', 'a', 'm'), ('t', 'e', 'n'), ('r', 'u', 'n')]
17. 22
18. Everything is affordable

19.48

20. Dear Sir, Please contact me. Thankyou

Sir/Madam, Please respond. Thanks

Hi, How are you? Bye

Hi, Where are you? Bye

21. (B)

22. `max([4, 3, 2, 7, 6]), min([4, 3, 2, 7, 6])`

29. `reduce(lambda x, y: x + ', ' + y, L)`

The other way of doing this is by using the `join` method. `','.join(s for s in L)`

If the list does not contain string values, then we can write `','.join(str(s) for s in L)`

35. `reduce(func, range(1, 5))`

39. `reduce(lambda x, y: x * y, range(1, n + 1))`

To make sure that the call works for 0 also, send 1 as the initial value.

`reduce(lambda x, y: x * y, range(1, n + 1), 1)`

We can also use the `prod` function from the `math` module.

`math.prod(range(1, n + 1))`

Chapter 20: Exception Handling

1. (A)

2. (B)

3. (A)
4. (B)
5. (A)
6. (B)
7. (C)
8. (A)
9. (B) String based exception are supported only in Python 2, not in Python 3.
10. (A)
11. (B)
12. (B)
13. (A)
14. (A)
15. (A)
16. (B)
17. (B)
18. (B)
19. (A)
20. (C)
21. (B) You can have a try statement that consist of only the try block and finally block.
22. (A)
23. (A)
24. (B)
25. (B) It is executed even when the exception is not handled.
26. (A)
27. (B)
28. (B)

- 29. (C)
- 30. (A)
- 31. (B)
- 32. (B)
- 33. (B)
- 34. (B)
- 35. (B)
- 36. (A)
- 37. (B)
- 38. (B)
- 39. (B)
- 40. (C) The bare except clause should be used at last
- 41. (i) Begin

AA

CC

Enter a number 2

0

DD

BB

End

- (ii) Begin

AA

CC

Enter a number 0

Traceback

.....

ZeroDivisionError

(iii) Begin

AA

CC

Enter a number two

Traceback

.....

ValueError

42. NameError: name 'x' is not defined

43.

(i) Enter student name : Raj

Raj 70.0

End

(ii) Enter student name : Ron

Traceback

.....

ZeroDivisionError:

(iii) Enter student name : Tom

Invalid name

End

46. (i)

Raj 70.0

Deep 85.0

End

(ii)

Raj 70.0

Deep 85.0

Sam 60.0

End

47. xx

48. try:

func()

except (IndexError, TypeError, ValueError):

log_it()

49. try:

func()

except ArithmeticError:

print('Arithmetic problem')

50. (i) Age is 2

(ii) Age cannot be more than 120 or less than 0

(iii) NameError: name 'age' is not defined

51. They work in the same way; else block is not required. It is because the continue statement ensures that the age variable is assigned.

60. (i) Enter age : 20

20

(ii) Enter age : 100

Invalid age value

Value of age should be in between 18 and 60

(iii) Enter age : thirty

Invalid age value

Traceback

ValueError: invalid literal for int() with
base 10: 'thirty'

During handling of the above exception,
another exception occurred:

Traceback

IndexError: tuple index out of range

61. If `ValueError` is raised by the `int` function, the `args` tuple will have only one value and if the `ValueError` is raised by the `raise` statement of our program then it will have 2 values. So we will conditionally execute the statement that uses `args[1]`.

63. There is no need of raising `KeyError`, Python will raise it anyways. Do not use the `raise` statement to duplicate what Python is already doing for you. If you want to change or add some additional exception information, then you can raise the exception.

65. In (A), the exception dies once it is caught in the `except` block. It is not propagated to the caller code.

In (B), we reraise it, so the active exception is propagated further.

Output of (A)

Caught a ZeroDivsionError in func : division
by zero

Output of (B)

Caught a ZeroDivsionError in func : division
by zero

Caught a ZeroDivsionError : division by zero

66. The second one will abnormally terminate as the exception is reraised, but it is not handled at higher level.

68. CustomError raised, 4, 8

4 8

4 8

69. By converting the exceptions, we could hide implementation details, and the user would see a simple and consistent error message. The details of the exception are written to the logfile which the developer can use for debugging.

Chapter 21: Context Managers

1. (B) The return value of `__enter__` is assigned to the variable.
2. (B)
3. (B)
4. (C)
5. (B)
6. (B)
7. (C)
8. (B)
9. (A)
10. (A)
11. No
12. (B)
13. Entering None Exiting
14. Entering 100 Exiting
15. (B)
16. `__init__` called
`__enter__` called

ABC

0

f called

__exit__called

__init__called

__enter__called

LMN

__exit__called

Traceback

ZeroDivisionError: integer division or modulo
by zero

26. You need to instantiate the `FileWriteOnly` class to get a context manager object.
28. Changes made to a list are saved only when a whole block of code successfully runs without any exceptions. If any exception occurs then any changes made to the list are discarded. This is achieved by working on a copy of the list.

Index

Symbols

- (minus)
 - difference of sets [137](#)
 - negation operator [23](#)
 - subtraction operator [23](#)
- != (inequality operator) [24](#)
- # (hash for comments) [37](#)
- % (percent sign)
 - modulo operator [23](#)
 - string formatting [71](#)
- * (asterisk)
 - multiplication operator [23](#)
 - repetition operator [55](#), [96](#)
 - tuple unpacking [113](#)
 - unpacking arguments [263](#)
- ** (double asterisk)
 - exponentiation operator [23](#)
 - unpacking arguments [264](#)
- , (comma) [20](#), [33](#)
- / (division) [23](#)
- // (floor division) [23](#)
- : (colon) [31](#), [145](#)
- ; (semicolon) [31](#)
- @ symbol for decorators [398](#), [521](#)
- [] (square brackets)
 - indexing [50](#), [86](#)
 - list comprehension [214](#)
- \(backslash)
 - escape sequence [68](#)
 - \n(newline) [68](#)
 - \t (tab) [68](#)
- _ (underscore) [12](#), [15](#), [43](#), [113](#), [296](#), [380](#)
- () (parentheses) [30](#), [108](#)
- { } (curly braces) [118](#), [133](#)
- + (plus)
 - addition [23](#)
 - concatenation [55](#), [96](#)
- += (augmented assignment) [27](#)
- < (less than) [24](#), [137](#)

<= (less than or equal to) [24](#), [136](#)
== (equality operator) [24](#)
> (greater than) [24](#), [137](#)
>= (greater than or equal to) [24](#), [136](#)
' (single quotes) [15](#), [49](#)
"(double quotes) [15](#), [49](#)
"""(triple quotes) [15](#), [58](#)
^ symmetric difference of sets [137](#)
| union of sets [137](#)
& intersection of sets [137](#)

A

absolute import [307](#)
abstract base classes [475](#)
aliasing [17](#), [130](#), [220](#)
and operator [157](#), [159](#)
anonymous functions [548](#)
argument passing
 about [245](#), [246](#)
 advantages [253](#)
 argument changes [252](#), [253](#)
 mutables, immutables [249-251](#)
 other languages [253](#)
 parameter rebound [248](#), [249](#)
arguments [237-238](#)
ASCII character set [77](#)
assert statement [626](#)
assertions [626-629](#)
assignment statement [16-20](#)
augmented assignment statements [27](#)

B

base class [461](#)
binary files [330-332](#)
bool type [15](#)
break statement [180-183](#), [210](#)
buffering [329](#), [330](#)
built-in exceptions [585](#)
built-in functions [41](#)
built-in functions [96](#)
built-in reducing functions [568](#)
byte code [3](#), [44](#)

C

chaining exceptions [618](#), [619](#)
chaining method calls [66](#)

- character encodings [76](#)
- chr function [79](#), [170](#)
- class decorators [543](#), [544](#)
- class decorators with parameters [544](#), [545](#)
- class definition [369](#)
- class methods [390](#), [391](#)
- class namespaces [387](#)
- class variables [384](#)
- classmethod decorator [390](#)
- closures [554](#), [555](#)
- command line arguments [348-350](#)
- comments [37](#)
- composition [477-480](#)
- comprehensions [214](#)
- concatenating lists [96](#)
- concatenating strings [55](#)
- conditional execution [145](#)
- constants [21](#)
- container types [39](#)
- context managers [638](#)
 - exception in with block [642-644](#)
 - implementation using class [639-642](#)
 - implementation using decorator [654-659](#)
 - standard library [651](#), [652](#)
- contextlib module [654](#)
- continue statement [183-187](#)
- copy module [105](#)
- custom exceptions [620](#)

D

- data hiding [379-384](#)
- data type [13-15](#)
- date class project [447-459](#)
- decimal module [652](#)
- decorator factory [535](#)
- decorators
 - about [519](#)
 - applications [525](#)
 - automatic decoration syntax [521](#), [522](#)
 - examples [522-524](#), [527-528](#)
 - general template [532](#), [533](#)
 - with parameters [533-535](#)
 - prerequisites [517-519](#)
 - preserving metadata [531](#), [532](#)
 - returning values [526](#), [527](#)
- decorators with parameters [533](#)
- deep copy [103-105](#), [130-132](#)
- def statement [233](#), [549](#)

- default arguments [253-256](#)
- del statement [20](#), [91](#)
- derived class [461](#)
- dictionaries
 - about [118-120](#)
 - adding key-value [120](#), [121](#)
 - checking key existence [123](#), [124](#)
 - combining [128](#)
 - comparing [124](#)
 - creating [126](#) -[127](#)
 - deleting pairs [124](#)
 - get() method [121](#), [122](#)
 - items() method [123](#)
 - iterating [173](#)
 - keys() method [123](#)
 - modifying values [121](#)
 - nesting [128](#)
 - setdefault() method [122](#)
 - values() method [123](#)
- dictionary comprehensions [222](#)
- dir() function [43](#), [218](#)
- division operations [23](#)
- docstrings [277](#)
- duck typing [474](#)
- dynamically typed [19](#)

E

- elif clause [151](#)
- else clause in if statement [148](#)
- else clause in loops [187](#)
- enumerate function [201](#)
- errors [44](#), [576-578](#)
- escape sequences [68-70](#)
- exception handling [576](#)
 - class hierarchy [585-587](#)
 - customized handling [587-590](#)
 - default handling [582-584](#)
 - else block [600-603](#)
 - finally block [595](#)
 - getting exception details [606-610](#)
 - nested try statement [610-613](#)
 - strategies [580-582](#)
- expressions [28](#)

F

- False value [15](#), [24](#), [25](#)
- files [326](#)

- accessing [327, 328](#)
- binary, text files [330-332](#)
- close method [332-334](#)
- modules [347](#)
- opening [327-329](#)
- print() function [342](#)
- random accessing [334, 335](#)
- read() function [338, 339](#)
- reading [338, 339](#)
- reading and writing [336-338](#)
- seek [335-336](#)
- with statement [333](#)
- writing [341](#)
- filter() function [564](#)
- finally block [595](#)
- float type [15](#)
- floor division [23](#)
- for loop
 - about [168-170](#)
 - index-based [198](#)
 - dictionaries, sets [173, 174](#)
 - range function [174, 175](#)
 - sequences [171-173](#)
 - strings [170, 171](#)
 - working [487](#)
- format() method [74](#)
- from statement [295](#)
- frozenset [139](#)
- function annotations [278, 279](#)
- function objects [272](#)
- functional programming [548](#)
- functions [41, 233-234](#)
 - arguments checking [238, 239](#)
 - attributes [275](#)
 - call [234-236](#)
 - definition [233](#)
 - local variables [239](#)
 - parameter, arguments [237, 238](#)
 - returning multiple values [244, 245](#)

G

- garbage collection [247, 312, 332](#)
- generator expressions [511, 512](#)
- generators [504-506](#)
- getattr() function [276](#)
- global statement [319-322](#)
- global variables [240, 313](#)

H

- hangman game project [355-35](#)
- help() function [11](#)
- hex() function [41](#)
- hexadecimal [14](#), [31](#), [41](#), [74](#), [77](#)

I

- id() function [16](#)
- IDE [7](#)
- identifiers [12](#), [13](#)
- IDLE [7](#), [9](#)
- if clause in list comprehension [217](#)
- if else operator [160](#)
- if statement [145-148](#)
- immutable types [39](#)
- importing [42](#), [293-297](#)
- in operator [56](#), [95](#), [109](#)
- indentation in Python [38](#), [165](#)
- indexing [50](#), [86](#)
- inheritance
 - about [461](#), [462](#)
 - base method, invoking [464](#), [465](#)
 - base method, overriding [463](#), [464](#)
 - derived class, adding [463](#)
 - multilevel inheritance [465](#)
 - multiple inheritance [466-468](#)
- initializer [376-378](#), [392](#)
- in-place changes [40](#), [97](#), [104](#), [198](#), [249](#)
- input function [34](#)
- installing Python [4](#)
- instance objects [369](#)
- instance variables [371](#)
- int type [14](#)
- interactive prompt [7](#)
- interpreter [2](#)
- is operator [26](#)
- isinstance [239](#), [439](#), [462](#)
- iter() function [483](#)
- iterables [483](#)
- iteration tools [489](#)
- iterators [483-487](#)
- itertools module [502](#)

J

- join() method [100](#)
- just-in-time compiler [4](#)
- Jython [4](#)

K

- key-value pairs [118](#)
- keyword arguments [259](#), [266](#)
- keyword-only arguments [268](#)

L

- lambda expressions
 - about [548](#), [549](#)
 - creating jump tables [555](#)
 - operator module [570-572](#)
 - returning function objects [553](#), [554](#)
 - uses [552](#), [553](#)
- lazy evaluation [501](#)
- LEGB rule [317](#)
- len() function [41](#), [50](#), [80](#), [120](#), [198](#)
- line-oriented methods [339-341](#)
- list comprehensions [214-217](#)
 - getting dictionary keys [219](#), [220](#)
 - if keyword [217](#), [218](#)
 - modifying list [219](#)
 - nested list comprehension [221](#), [222](#)
 - ternary operator [218](#)
- lists
 - about [85](#), [86](#)
 - append() method [89](#)
 - changing a portion [88](#), [89](#)
 - changing item [87](#), [88](#)
 - clear() method [92](#)
 - comparing [95](#), [96](#)
 - concatenation [96](#)
 - copying a list [101-103](#)
 - creation [98](#), [99](#)
 - extend() method [90](#)
 - indexing [86](#)
 - insert() method [89](#), [90](#)
 - pop() method [91](#)
 - remove() method [91](#), [92](#)
 - repetition [96](#), [99](#), [105](#)
 - reversing [94](#)
 - searching [94](#), [95](#)
 - slicing [87](#)
 - sorting [92-94](#)
- literals [14](#)
- local scope [315](#)
- local variables [239](#)
- log in system project [424](#)
- logical errors [44](#)

- logical operators [25](#)
- looping techniques [196](#)
- loops [164](#)
 - for loop vs. while loop [189](#)
 - index based for loop [198](#)
 - infinite loop with break [206-209](#)
 - in-place changes [198-200](#)
 - reverse order [196, 197](#)
 - skipping items [200](#)
 - unique values [197, 198](#)
 - zip sequences [202, 203](#)

M

- magic methods [430](#)
- map function [561-563](#)
- math module [42, 293](#),
- max() function [41, 74, 96, 135, 489](#)
- membership operator [123-124](#)
- memory management [1, 21](#)
- Method Resolution Order (MRO) [468-470](#)
- methods [41, 369](#)
- min() function [41, 274, 489](#)
- module object [300, 301](#)
- modules [291, 302](#)
 - byte-compiled version [301](#)
 - creating [293](#)
 - documenting [298, 299](#)
 - exploring [292, 293](#)
 - importing [293-297](#)
 - reloading [302](#)
 - search path [299, 300](#)
- multiline statements [43](#)
- multiline strings [57](#)
- mutable types [39](#)

N

- name resolution [317-319](#)
- namespaces [310-315](#)
- naming conventions and rules [12, 21, 170, 234, 380](#)
- nested data structures [178-180](#)
- nested if statements [149](#)
- nested list comprehensions [221](#)
- nested lists [101](#)
- nested loop [175-177](#)
- nested try statements [610](#)
- nested with statement [652-654](#)
- next function [484](#)

None object [15](#), [26](#)
nonlocal statement [322-324](#)

O

object class [465](#), [466](#)
object namespaces [387](#)
object-oriented programming
 about [366-368](#)
 adding methods to class [369-371](#)
 class designing [402](#)
objects [16](#)
open() function [326](#)
operator module [570](#)
operator overloading [430](#)
operators [22](#)
 arithmetic [22-24](#)
 bitwise [27](#)
 identity [26](#), [27](#)
 logical [25](#)
 membership [27](#)
 precedence, associativity [28](#), [29](#)
 relational [24](#)
 type conversion, types [30](#), [31](#)
or operator [157](#), [159](#)
ord() function [79](#), [170](#)

P

packages [305](#)
 importing contents [305](#), [306](#)
 relative imports [307](#), [308](#)
 subpackages [307](#)
parameters [237](#)
pass by assignment [246-248](#)
pass statement [189](#)
pickle module [350](#)
polymorphism [472-474](#)
positional arguments [259](#), [264](#), [270](#)
positional-only arguments [270](#)
pprint module [130](#)
print function [32-34](#)
programming paradigms, [366](#)
properties [396-401](#)
Python [1](#)
 container types [39](#)
 executing scripts [8-9](#)
 help [11](#)
 implementation [2-4](#)

- importance [1-2](#)
- installing [4-7](#)
- interactive mode [7-8, 43](#)
- objects [16](#)
- types [13-15](#)

Python Enhancement Proposals(PEPs) [45](#)

Q

quiz creation project [408-417](#)

R

raise statement [613](#)
raising exception [613-616](#)
random module [98](#)
range [98, 200](#)
raw string literals [70](#)
Read-Eval-Print Loop (REPL) [10](#)
recursion [279-284](#)
reduce() function [567](#)
relative import [307](#)
reload() function [302](#)
repr() function [440](#)
re-raising Exception [616-618](#)
return statements [240-243](#)
reversed() function [94, 123, 196](#)
runtime context [646-647](#)

S

scope [315](#)
scripts [302](#)
self parameter [370](#)
sequential control [145](#)
set comprehensions [225](#)
sets

- about [132, 133](#)
- adding, removing elements [135](#)
- comparing sets [135](#)
- creation [134](#)
- difference [137](#)
- intersection [137](#)
- iterating [173](#)
- union [137](#)

shallow copy [103, 130](#)
slicing [52, 87](#)
snakes and ladders project [417-423](#)
sorted() function [196, 556-560](#)

- statements [31](#), [32](#)
- static methods [394](#), [395](#)
- strings [49](#), [50](#)
 - aligning text [62](#)
 - case-changing methods [60](#), [61](#)
 - chaining method calls [66](#), [67](#)
 - character classifying methods [61](#)
 - comparisons [67](#)
 - concatenation [55](#), [56](#)
 - conversion [68](#)
 - encodings [76-81](#)
 - formatting [71-74](#)
 - indexing [50](#), [51](#)
 - iterating over [170](#)
 - membership checking [56](#)
 - methods [60](#)
 - multiline [57-60](#)
 - mutability [51](#), [52](#)
 - repetition [55-56](#)
 - replacing substring [63-66](#)
 - searching [63](#)
 - slicing [52-55](#)
 - stripping methods [62](#), [63](#)
 - whitespace adding [57](#)
- sum() function [96](#), [216](#), [245](#)
- super() function [464](#), [470](#)

T

- ternary operator [160](#), [218](#)
- truthiness [154](#)
- try..except [587-590](#)
- tuples
 - about [107-111](#)
 - packing [111](#)
 - unpacking [111-113](#), [171](#)
- type conversion [30](#)
- type() function [15](#)

U

- Unicode character set [77](#)
- unpacking a tuple [111-113](#), [171](#)
- unpacking arguments [262](#), [271](#)
- user defined exceptions [620](#)
- UTF-16 encoding [78](#)
- UTF-8 encoding [78](#)

V

variables [16-20](#)

W

while loop

- about [164](#), [165](#)

- else block [187](#), [188](#)

- input error checking [167](#)

- storing user input [168](#)

- vs. for loop [189](#), [190](#)

with statement [333](#), [638](#)

Y

yield statement [654](#)

Z

zip() function [127](#), [202](#)