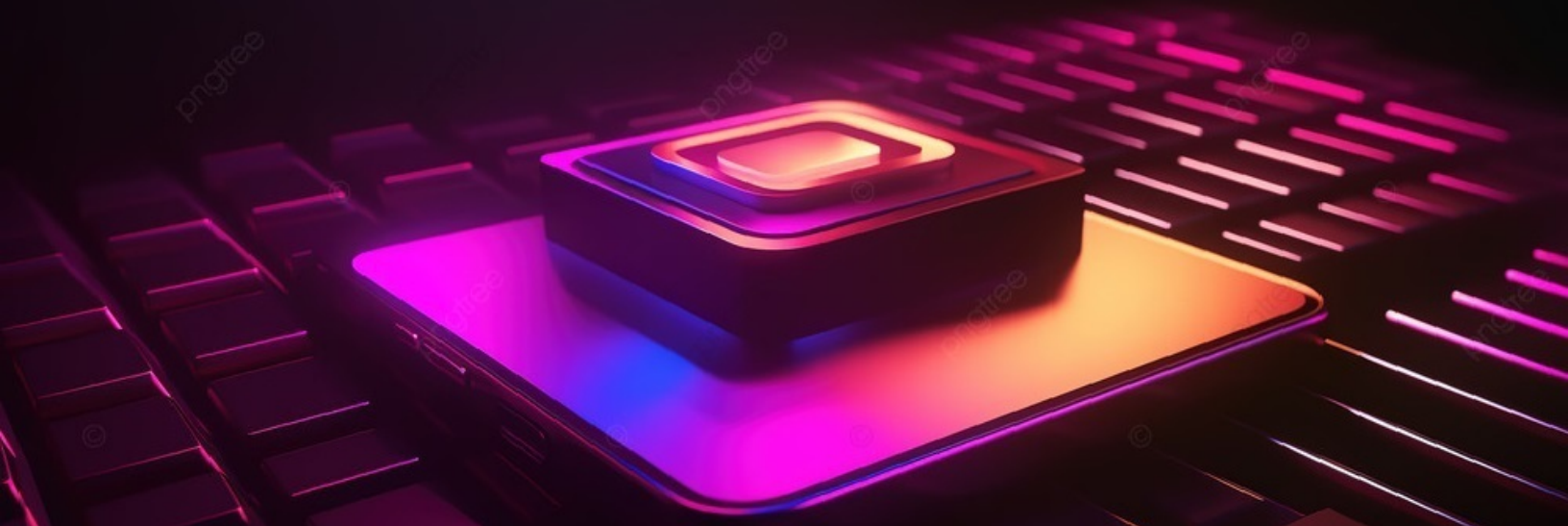
PYTHON

BECOME A MASTER

{ 'BEGINNER', 'INTERMEDIATE', 'ADVANCED' }

1 DAY



FOR FRESHER LEARN PYTHON ATTEND INTERVIEWS

200 PROJECTS TO CODE

WITH MORE THAN 400 CONCEPTS EXPLAINED

With Hands-On Exercises and Code Snippets

PYTHON PROGRAMMING BEGINNERS

In This Book You Will Learn Explanation with Diagram and Readymade Solution With Step By Step Explanation

AMIT K

Contents

Introduction

Learning Python Programming step by step.

Python's popularity stems from its simplicity, versatility, and robustness. Here are some of its main features:

- **Readable and Simple Syntax:** Python's syntax is designed to be easy to read and write, making it accessible to beginners and experienced programmers alike. It emphasizes readability, reducing the cost of program maintenance and development.
- **Extensive Standard Library:** Python comes with a large standard library that provides modules and packages for various tasks such as string manipulation, file I/O, networking, and more. This extensive library reduces the need for additional third-party modules and simplifies development.
- **Dynamic Typing and Dynamic Binding:** Python is dynamically typed, meaning you don't need to declare the type of variables. This allows for more flexibility and faster development cycles. Additionally, Python features dynamic binding, which means that variable names are bound to objects at runtime, providing more flexibility in code structure.

Chapter 1 :

Python Introduction

1 Introduction

Python's popularity stems from its simplicity, versatility, and robustness. Here are some of its main features:

- **Readable and Simple Syntax:** Python's syntax is designed to be easy to read and write, making it accessible to beginners and experienced programmers alike. It emphasizes readability, reducing the cost of program maintenance and development.

- **Extensive Standard Library:** Python comes with a large standard library that provides modules and packages for various tasks such as string manipulation, file I/O, networking, and more. This extensive library reduces the need for additional third-party modules and simplifies development.
- **Dynamic Typing and Dynamic Binding:** Python is dynamically typed, meaning you don't need to declare the type of variables. This allows for more flexibility and faster development cycles. Additionally, Python features dynamic binding, which means that variable names are bound to objects at runtime, providing more flexibility in code structure.

- **Cross-platform Compatibility:** Python is available for all major operating systems (Windows, macOS, Linux) and is portable across platforms. This allows developers to write code once and run it anywhere, making it highly versatile.
- **High-level Language:** Python abstracts many low-level details like memory management and provides high-level data types and constructs, making it easier to focus on solving problems rather than dealing with implementation details.
- **Support for Multiple Programming Paradigms:** Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the most appropriate approach for their specific needs.
- **Large and Active Community:** Python has a vast and active community of developers who contribute to its growth and development. This community support means there are abundant resources, libraries, frameworks, and tools available for Python development.
- **Scalability and Performance:** While Python may not be as fast as compiled languages like C or C++, it offers good performance for most applications. Additionally, Python can be easily integrated with high-performance languages for performance-critical tasks, and there are various optimization techniques available to improve performance.

- **Open Source:** Python is open source, meaning its source code is freely available and can be modified and redistributed by anyone. This fosters innovation and collaboration within the Python community.

These features collectively contribute to Python's popularity and widespread adoption in various domains such as web development, data science, machine learning, artificial intelligence, scientific computing, automation, and more.

Compiler Install

Python is an interpreted language, so it doesn't require compilation in the traditional sense. However, you do need to install the Python interpreter on your system to run Python code. Here's how you can download and install Python:

- **Visit the Python website:** Go to the official Python website at https://www.python.org/.
- **Download Python:** On the homepage, you'll see a prominent button for downloading Python. Click on it to navigate to the download page.

- **Choose the installer:** You'll be presented with different versions of Python. Typically, you'll want to download the latest stable version for your operating system (Windows, macOS, or Linux). Python distributions are available for both 32-bit and 64-bit systems.
- **Download the installer:** Click on the download link for the installer that matches your operating system and architecture.
- **Run the installer:** Once the installer is downloaded, run it by double-clicking on the downloaded file.
- **Follow the installation wizard:** The installation wizard will guide you through the installation process. You can choose the installation directory, customize the installation options if needed, and install additional features like pip (Python's package manager) and adding Python to the system PATH (recommended).

- **Complete the installation:** Once you've configured the installation options, proceed with the installation. The installer will copy the necessary files to your system and set up Python.
- **Verify the installation:** After the installation is complete, you can verify that Python is installed correctly by opening a command prompt (Windows) or terminal (macOS/Linux) and typing python --version or python3 --version. This should display the installed Python version.

Hello world

Writing "Hello, World!" to the console in Python is straightforward. Here's the code:

```
print("Hello, World!")
```

You can write this code in a text editor or an integrated development environment (IDE) such as Visual Studio Code, PyCharm, or IDLE. Save the file with a .py extension, such as hello_world.py.

To run the Python code and see the output in the console:

1 Open a command prompt (Windows) or terminal (macOS/Linux).

2 Navigate to the directory where your Python file (hello_world.py) is saved using the cd command.

3 Once you're in the correct directory, type python hello_world.py and press Enter (or python3 hello_world.py on some systems). This will execute the Python script, and you should see "Hello, World!" printed to the console.

Comments

In Python, you can write comments to document your code or provide explanations. Comments are ignored by the Python interpreter and are meant for human readers. Here's how you can write comments in Python:

Single-line comments:

```
# This is a single-line comment
print("Hello, World!")  # This is also a single-line comment
```

Multi-line comments (technically, multi-line strings that are not assigned to a variable):

```
"""
This is a multi-line comment.
You can write multiple lines of text within triple quotes.
This is often used as a docstring for documenting functions or modules.
"""
print("Hello, World!")
```

Alternatively, you can use the # character to comment out multiple lines:

```
# This is a comment
# This is another comment
print("Hello, World!")
```

It's a common convention in Python to write clear and concise comments to explain your code, making it easier for others (and your future self) to understand the purpose and functionality of the code.

2 Python Variables

Introduction

In Python, variables are used to store data values. A variable is a name that refers to a value stored in memory. Unlike some other programming languages, Python is dynamically typed, meaning you don't need to declare the type of a variable before assigning a value to it. Here's how you can use variables in Python:

1 Variable Assignment: You can assign a value to a variable using the = operator.

```
x = 10
name = "Alice"
```

2 Variable Naming Rules:

- Variable names must start with a letter (a-z, A-Z) or an underscore _.
- Variable names can contain letters, digits (0-9), and underscores _.
- Variable names are case-sensitive (name and Name are different variables).
- Python keywords (e.g., if, for, while, def, etc.) cannot be used as variable names.

3 Data Types: Python variables can hold values of different data types, including:

- Integer (int): Whole numbers, e.g., 10, -5, 1000.
- Float (float): Floating-point numbers, e.g., 3.14, 2.718.
- String (str): Sequence of characters, e.g., "Hello", 'Python'.
- Boolean (bool): Represents True or False.
- List (list), Tuple (tuple), Dictionary (dict), Set (set), etc.

```
age = 25  # integer
pi = 3.14  # float
name = "Alice"  # string
is_student = True  # boolean
```

4 Variable Reassignment: You can change the value of a variable by assigning a new value to it.

```
x = 5
x = x + 1  # x now holds the value 6
```

5 Variable Scope: Variables have a scope, which defines where they can be accessed from. Variables declared inside a function have local scope,

meaning they are only accessible within that function. Variables declared outside of any function have global scope and can be accessed from anywhere in the code.

```
x = 10  # global variable

def my_function():
    y = 20  # local variable
    print(x)  # x can be accessed here
    print(y)  # y can be accessed here

my_function()
print(x)  # x can be accessed here
print(y)  # Error: y is not defined
```

Variables are fundamental to programming in Python, as they allow you to store and manipulate data within your programs.

Constants

In Python, constants are typically created by defining variables with uppercase names and treating them as if they were constant, although Python does not have built-in support for constant variables like some other programming languages. Here's how you can create constants in Python:

```
MY_CONSTANT = 10
ANOTHER_CONSTANT = "Hello"
```

By convention, constants are usually written in uppercase to distinguish them from regular variables. However, it's important to note that Python doesn't enforce constants; you can still modify their values, but it's a convention to treat them as immutable.

If you want to prevent accidental modification of a constant value, you can use the readonly or dataclass module to create immutable objects. Here's an example using readonly:

```
from readonly import readonly

@readonly
class Constants:
    MY_CONSTANT = 10
    ANOTHER_CONSTANT = "Hello"
```

```
# Now you cannot modify the values of these constants:
# Constants.MY_CONSTANT = 20  # This will raise an AttributeError
```

Chapter 2 :

Python Data Types

1 Python boolean

Introduction

Boolean values are a data type in Python that represent truth values. They can only have one of two values: True or False. Boolean values are commonly used in conditional statements, loops, and logical operations to control the flow of the program or to represent the result of a comparison.

Here's how you can use Boolean values in Python:

1 **Boolean Variables:** You can assign Boolean values to variables like any other data type.

```
is_student = True
has_passed_exam = False
```

2 **Conditional Statements:** Boolean values are often used in conditional statements (if, elif, else) to execute different blocks of code based on whether a condition is True or False.

```
if is_student:
    print("The person is a student.")
else:
    print("The person is not a student.")
```

3 **Logical Operators:** Boolean values can be combined using logical operators (and, or, not) to create compound conditions.

```
age = 20
is_adult = age >= 18
is_teenager = age >= 13 and age <= 19
```

4 **Loops:** Boolean values are often used as loop conditions to control the execution of the loop.

```
while is_student:
    print("Still studying...")
    # Some condition to update is_student, otherwise, it may result in an infinite loop
    is_student = False
```

5 **Function Returns:** Functions can return Boolean values to indicate the success or failure of an operation or to indicate a condition.

```
def is_even(number):
    return number % 2 == 0

print(is_even(4))  # True
print(is_even(5))  # False
```

6 **Built-in Functions:** Python provides built-in functions like bool() to convert other data types to Boolean values.

```
print(bool(0))     # False
print(bool(1))     # True
print(bool([]))    # False (empty list)
print(bool([1]))   # True (non-empty list)
```

boolean to int

In Python, you can convert a Boolean value to an integer using the int() function. When you convert a Boolean value to an integer, True is represented as 1 and False is represented as 0. Here's how you can do it:

```
```

```
# Convert True to integer
boolean_value = True
integer_value = int(boolean_value)
print(integer_value)  # Output: 1

# Convert False to integer
boolean_value = False
integer_value = int(boolean_value)
print(integer_value)  # Output: 0
```

boolean to string

In Python, you can convert a boolean value to a string using either the str() function or by using string formatting methods. Here are a example:

Using str() function:

```
# Convert True to string
boolean_value = True
string_value = str(boolean_value)
print(string_value)  # Output: "True"

# Convert False to string
boolean_value = False
string_value = str(boolean_value)
print(string_value)  # Output: "False"
```

True False

In Python, True and False are the two boolean values that represent the truth values. They are used to control the flow of the program, make decisions, and perform logical operations. Here are some important notes on True and False and how to use them in Python:

1 **True and False Values:**

- True and False are reserved keywords in Python.
- True represents the truth value true, while False represents the truth value false.
- These are the only two boolean values in Python.

2 **Boolean Operations:**

- Boolean values are commonly used in logical operations such as AND (and), OR (or), and NOT (not).
- The and operator returns True if both operands are true, otherwise it returns False.
- The or operator returns True if at least one of the operands is true, otherwise it returns False.
- The not operator returns the opposite boolean value of its operand.

3 **Comparison Operators:**

- Comparison operators (==, !=, <, <=, >, >=) return boolean values (True or False) based on the comparison result.
- For example, x == y returns True if x is equal to y, otherwise it returns False.

4 **Control Structures:**

- Boolean values are extensively used in control structures such as if, elif, else statements and loops (while, for).
- These structures allow you to execute different blocks of code based on conditions evaluated to True or False.

5 **Truthiness and Falsiness:**

- In addition to True and False, other values in Python can be evaluated as either true or false in a boolean context.
- Values such as empty sequences (lists, tuples, strings, etc.), 0, and None are evaluated as False. Any non-zero number or non-empty sequence is evaluated as True.

6 **Return Values:**

- Functions can return boolean values to indicate the success or failure of an operation or to represent a condition.
- For example, a function that checks if a number is even might return True or False based on the result of the check.

Understanding how to use True and False effectively is essential for writing clear, concise, and efficient Python code. They enable you to express conditions and logic in a way that is easy to understand and maintain.

2 Python String

Introduction

Creating a variable and assigning a string value to it in Python is straightforward. Here's how you can do it:

```
# Create a variable named 'my_string' and assign a string value to it
my_string = "Hello, World!"

# Print the value of the variable
print(my_string)
```

In this example:

- We create a variable named my_string.
- We use the assignment operator = to assign the string "Hello, World!" to the variable my_string.
- We then print the value of my_string using the print() function.

You can assign any string value to a variable in Python, and the variable will hold that value until it is changed or deleted.

Variables in String

In Python, you can use variables inside strings using string formatting techniques. There are several ways to achieve this, including using the % operator, the str.format() method, and f-strings (formatted string literals). Here's how you can use variables inside strings using each method:

1 **Using % Operator:** You can use the % operator to insert variables into strings. This method is older and less recommended compared to newer methods like f-strings and str.format().

```
name = "Alice"
age = 30
greeting = "Hello, %s! You are %d years old." % (name, age)
print(greeting)
```

2 **Using str.format() Method:** The str.format() method allows you to format strings with placeholders {} that are replaced with variable values.

```
name = "Alice"
age = 30
greeting = "Hello, {}! You are {} years old.".format(name, age)
print(greeting)
```

3 **Using f-strings (Formatted String Literals):** F-strings provide

a more concise and readable way to insert variables into strings. You can directly include variables and expressions within curly braces {} inside the string.

```
name = "Alice"
age = 30
greeting = f"Hello, {name}! You are {age} years old."
print(greeting)
```

All three methods achieve the same result, but f-strings are generally preferred due to their simplicity and readability. They were introduced in Python 3.6 and offer a more intuitive way to format strings with variables.

Escape

Escape sequences in Python strings are special characters that are preceded by a backslash \. These sequences allow you to include characters in strings that are difficult or impossible to type directly in source code. Here are some commonly used escape sequences in Python:

1 **\n:** Newline character. It inserts a newline into the string.

```
print("Line 1\nLine 2")
```

Output:

```
Line 1
Line 2
```

2 **\t:** Tab character. It inserts a horizontal tab into the string.

```
print("Column 1\tColumn 2")
```

Output:

```
Column 1    Column 2
```

3 **\r:** Carriage return character. It moves the cursor to the beginning of the line.

```
print("Hello\rWorld")
```

Output:

```
World
```

4 **\\:** Backslash character. It inserts a literal backslash into the string.

```
print("This is a backslash: \\")
```

Output:

```
This is a backslash:
\
```

Quote

To include single quotes (') or double quotes (") inside a Python string, you can use the opposite type of quote to delimit the string, or you can escape the quote character using a backslash (\). Here's how you can do it:

1 **Using Opposite Type of Quote:** If you want to include single quotes inside a string delimited by double quotes or vice versa, you can simply use the opposite type of quote inside the string.

```
# Using double quotes to delimit the string with single quotes inside
string_with_single_quotes = "He said, 'Hello'"
print(string_with_single_quotes)

# Using single quotes to delimit the string with double quotes inside
string_with_double_quotes = 'She said, "Hi"'
print(string_with_double_quotes)
```

Output:

```
He said,
'Hello'
```

```
She said, "Hi"
```

2 **Escaping Quote Characters:** If you want to include the same type of quote character inside a string, you can escape it using a backslash (\).

```
# Using single quotes with escaped single quote inside
string_with_escaped_single_quote = 'I\'m fine'
print(string_with_escaped_single_quote)

# Using double quotes with escaped double quote inside
string_with_escaped_double_quote = "She said, \"It's raining\""
print(string_with_escaped_double_quote)
```

Output:

```
I'm fine
She said, "It's raining"
```

Both of these methods allow you to include single quotes or double quotes inside a Python string without causing syntax errors. Choose the method that best fits your code style and readability preferences.

3 Python Number

Introduction

Python supports several numerical data types, each with its own characteristics and use cases. Here are the main numerical data types that Python can handle:

1 **Integer (int):**

- Integers represent whole numbers without any decimal point.
- Example: 5, -10, 1000.

2 **Floating-Point (float):**

- Floating-point numbers represent real numbers with a decimal point.
- Example: 3.14, 2.718, -0.5.

3 **Complex (complex):**

- Complex numbers represent numbers with both a real part and an imaginary part.
- They are written with a j or J suffix to denote the imaginary part.
- Example: 3 + 2j, -4.5 - 1.2j.

4 **Decimal (decimal.Decimal):**

- Decimal numbers represent fixed-point decimal numbers with arbitrary precision.
- They are useful for financial and other applications requiring exact decimal representations.
- Example: Decimal('3.14'), Decimal('10.5').

5 **Fraction (fractions.Fraction):**

- Fraction numbers represent rational numbers as fractions of integers.
- They are useful for exact representation of fractions.
- Example: Fraction(3, 4), Fraction(5, 2).

6 **Boolean (bool):**

- Boolean values represent truth values, which can be either True or False.

- They are commonly used for logical operations and control flow.
- Example: True, False.

7 **Rational (sympy.Rational):**

- Rational numbers represent fractions of integers with arbitrary precision.
- They are similar to fractions but offer additional capabilities and are part of the sympy library.
- Example: Rational(3, 4), Rational(5, 2).

These are the main numerical data types that Python can handle. Depending on your requirements, you can choose the appropriate data type for your calculations and applications.

Integer

Creating integer numbers and performing simple calculations in Python is straightforward. You can define integer variables and use mathematical operators to perform calculations. Here's an example:

```
# Create integer variables
x = 5
y = 3

# Perform arithmetic operations
sum_result = x + y
difference_result = x - y
product_result = x * y
quotient_result = x / y  # Division returns a float in Python 3.x
floor_division_result = x // y  # Floor division returns an integer
remainder_result = x % y  # Modulus operator returns the remainder

# Print the results
print("Sum:", sum_result)
print("Difference:", difference_result)
print("Product:", product_result)
print("Quotient:", quotient_result)
print("Floor Division:", floor_division_result)
print("Remainder:", remainder_result)
```

Output:

```
Sum: 8
Difference: 2
Product: 15
Quotient: 1.6666666666666667
Floor Division: 1
Remainder: 2
```

In this example:

- We create two integer variables x and y.
- We perform simple arithmetic operations using the +, -, *, /, //, and % operators.
- We print the results of the calculations.

You can perform various arithmetic operations on integer numbers in Python, including addition, subtraction, multiplication, division, floor division, and modulus. Depending on the operands and operators used, Python will return the appropriate result, which can be an integer or a floating-point number.

Floats

Creating floating-point numbers and performing simple calculations in Python is similar to working with integer numbers. Here's an example of creating floating-point variables and performing arithmetic operations:

```
# Create floating-point variables
x = 3.5
y = 2.0

# Perform arithmetic operations
sum_result = x + y
difference_result = x - y
product_result = x * y
quotient_result = x / y
```

You can use the same arithmetic operators (+, -, *, /, //, %) as with integer numbers to perform calculations with floating-point numbers. However, it's

important to note that when performing division (/), Python will always return a floating-point result, even if the operands are integers.

```
# Print the results
print("Sum:", sum_result)
print("Difference:", difference_result)
print("Product:", product_result)
print("Quotient:", quotient_result)
```

In this example:

- We create two floating-point variables x and y.
- We perform simple arithmetic operations using the +, -, *, and / operators.
- We print the results of the calculations.

Output:

```
Sum: 5.5
Difference: 1.5
Product: 7.0
Quotient: 1.75
```

You can perform various arithmetic operations on floating-point numbers in Python, and Python will return floating-point results for division operations.

Multiple Assignment

Multiple assignment in Python allows you to assign multiple variables in a single line, each with its corresponding value. This can be done using tuple

unpacking or list unpacking. Here's how you can do multiple assignment:

1 **Tuple Unpacking:** You can use a tuple on the right-hand side of the assignment operator to assign values to multiple variables.

```
# Tuple unpacking
x, y, z = 10, 20, 30

print("x:", x)  # Output: 10
print("y:", y)  # Output: 20
print("z:", z)  # Output: 30
```

2 **List Unpacking:** You can use a list on the right-hand side of the assignment operator to assign values to multiple variables.

```
# List unpacking
[a, b, c] = [10, 20, 30]

print("a:", a)  # Output: 10
print("b:", b)  # Output: 20
print("c:", c)  # Output: 30
```

3 **Extended Unpacking:** You can use extended unpacking with the * operator to assign the remaining elements of an iterable to a single variable.

```
# Extended unpacking
first, *rest = [1, 2, 3, 4, 5]

print("first:", first)  # Output: 1
print("rest:", rest)    # Output: [2, 3, 4, 5]
```

4 **Swapping Variables:** Multiple assignment is commonly used to swap the values of variables without needing a temporary variable.

```
# Swapping variables
```

```
x = 10
y = 20

x, y = y, x

print("x:", x)  # Output: 20
print("y:", y)  # Output: 10
```

Multiple assignment is a convenient feature in Python that allows you to write concise and readable code when working with multiple values simultaneously.

Int to string

To convert an integer to a string in Python, you can use the str() function. Here's how you can do it:

```
# Convert an integer to a string
number = 123
string_number = str(number)

# Print the string representation
print(string_number)
```

Output:

```
12
3
```

In this example, the str() function is used to convert the integer 123 to a string representation "123". The resulting string can then be used in string operations, printed, or stored in a variable for further processing.

Int from string

To convert a string to an integer in Python, you can use the int() function. Here's how you can do it:

```
# Convert a string to an integer
string_number = "123"
number = int(string_number)
```

```
# Print the integer value
print(number)
```

Output:

```
12
3
```

In this example, the int() function is used to convert the string "123" to an integer value 123. The resulting integer can then be used in arithmetic operations, comparisons, or stored in a variable for further processing.

Int to floating point

To convert integers to floating-point numbers in Python, you can simply use the float() function. Here's how you can do it:

```
# Convert an integer to a floating-point number
integer_number = 123
float_number = float(integer_number)

# Print the floating-point number
print(float_number)
```

Output:

```
123.0
```

In this example, the float() function is used to convert the integer 123 to a floating-point number 123.0. The resulting floating-point number can then be used in arithmetic operations, comparisons, or stored in a variable for further processing.

4 Python List

Introduction

In Python, a list is a built-in data structure that represents a collection of items in a specific order. Lists are mutable, meaning that you can modify their elements after they have been created. Lists can contain elements of different data types, and they can grow or shrink dynamically as needed.

Here's a simple example of a Python list:

```
```

```
# Creating a list
my_list = [1, 2, 3, 4, 5]

# Accessing elements of the list
print(my_list[0])  # Output: 1
print(my_list[2])  # Output: 3

# Modifying elements of the list
my_list[1] = 10
print(my_list)  # Output: [1, 10, 3, 4, 5]

# Adding elements to the list
my_list.append(6)
print(my_list)  # Output: [1, 10, 3, 4, 5, 6]

# Removing elements from the list
my_list.remove(3)
print(my_list)  # Output: [1, 10, 4, 5, 6]
```

You might need to use a list in Python when:

1. **Storing Multiple Values:** You need to store multiple values of possibly different data types in a single variable.
2. **Maintaining Order:** You need to preserve the order of elements in the collection.
3. **Dynamic Size:** You need a data structure that can dynamically grow or shrink as elements are added or removed.
4. **Mutable Operations:** You need to perform mutable operations such as adding, removing, or modifying elements in the collection.
5. **Indexing and Slicing:** You need to access elements by their index or perform slicing operations to extract subsets of elements.

Lists are incredibly versatile and are widely used in Python programming for tasks such as storing collections of items, representing sequences of data, managing sets of values, and more. They provide a flexible and efficient way to work with collections of data in Python.

Access Elements by Index

In Python, you can access elements in a list by their index. List indexing starts at 0, meaning the first element of the list has an index of 0, the second element has an index of 1, and so on. You can also use negative indices to access elements from the end of the list, where -1 represents the last element, -2 represents the second-to-last element, and so forth. Here's how you can access elements by index in a Python list:

```
# Define a list
my_list = ['apple', 'banana', 'cherry', 'date', 'elderberry']

# Access elements by index
print(my_list[0])  # Output: 'apple' (first element)
print(my_list[2])  # Output: 'cherry' (third element)
print(my_list[-1]) # Output: 'elderberry' (last element)
print(my_list[-2]) # Output: 'date' (second-to-last element)
```

In this example:

- We define a list my_list containing five elements.
- We use square brackets [] to access elements by their index within the list.
- Positive indices refer to elements starting from the beginning of the list, while negative indices refer to elements starting from the end of the list.
- We print the values of the elements at index 0, 2, -1, and -2 to demonstrate accessing elements by index.

Modify Elements

To modify elements in a Python list, you can directly assign new values to specific elements using their indices. Lists in Python are mutable, which means you can change their elements after they have been created. Here's how you can modify elements in a Python list:

```
# Define a list
my_list = ['apple', 'banana', 'cherry', 'date', 'elderberry']

# Modify elements in the list
my_list[1] = 'blueberry'  # Modify the element at index 1
my_list[-2] = 'grape'     # Modify the second-to-last element

# Print the modified list
print(my_list)  # Output: ['apple', 'blueberry', 'cherry', 'grape', 'elderberry']
```

In this example:

- We define a list my_list containing five elements.
- We use square brackets [] to access elements by their indices within the list.
- We assign new values to elements at specific indices to modify them.
- We print the modified list to verify the changes.

You can modify elements in a list using any valid Python expression. This means you can assign new values of the same type, different types, or even the result of an expression involving other elements in the list. Lists provide a flexible and powerful way to manage collections of data in Python.

Add Elements

To add elements to a Python list, you can use various methods such as append(), insert(), or concatenation (+ operator). Here's how you can add elements to a list using these methods:

1 **Using append() method:** The append() method adds a single element to the end of the list.

```
# Define a list
my_list = ['apple', 'banana', 'cherry']

# Add a single element to the end of the list
my_list.append('date')

# Print the modified list
```

```
print(my_list)  # Output: ['apple', 'banana', 'cherry', 'date']
```

2 **Using insert() method:** The insert() method inserts a single element at a specified index in the list.

```
# Define a list
my_list = ['apple', 'banana', 'cherry']

# Insert a single element at a specified index
my_list.insert(1, 'blueberry')  # Insert 'blueberry' at index 1

# Print the modified list
print(my_list)  # Output: ['apple', 'blueberry', 'banana', 'cherry']
```

3 **Using concatenation (+ operator):** You can concatenate two lists to create a new list containing elements from both lists.

```
# Define a list
my_list = ['apple', 'banana', 'cherry']

# Create another list of elements to add
new_elements = ['date', 'elderberry']

# Concatenate the two lists to add elements
my_list += new_elements

# Print the modified list
print(my_list)  # Output: ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

In each example, we first define a list (my_list). Then, we use one of the methods (append(), insert(), or concatenation) to add elements to the list. Finally, we print the modified list to verify the changes.

These methods provide flexible ways to add elements to a list in Python, allowing you to easily extend the list with new data as needed.

Insert Elements

To insert elements in the middle of a Python list, you can use the insert() method. The insert() method allows you to specify the index where you want to insert the new element. Here's how you can insert elements into the middle of a list:

```
```

```
# Define a list
my_list = ['apple', 'banana', 'cherry', 'date']

# Insert an element in the middle of the list
my_list.insert(2, 'blueberry')  # Insert 'blueberry' at index 2

# Print the modified list
print(my_list)  # Output: ['apple', 'banana', 'blueberry', 'cherry', 'date']
```

In this example:

- We define a list my_list containing four elements.
- We use the insert() method to insert the element 'blueberry' at index 2.
- The index 2 indicates that the new element will be inserted after the second element ('banana') and before the third element ('cherry').
- We print the modified list to verify the changes.

The insert() method modifies the original list in place by shifting existing elements to make room for the new element. It's a convenient way to insert elements at specific positions within a list.

Sort Permanently

To sort a Python list permanently using the sort() method, you can simply call the sort() method on the list. The sort() method sorts the elements of the list in ascending order by default. Here's how you can use the sort() method to sort a list permanently:

```
# Define a list
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

# Sort the list permanently
my_list.sort()

# Print the sorted list
print("Sorted list:", my_list)  # Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
```

In this example:

- We define a list my_list containing unsorted elements.

- We call the sort() method on the list my_list to sort its elements permanently.
- The sort() method modifies the original list by sorting its elements in ascending order.
- We print the sorted list to verify the changes.

5 Python Tuple

Introduction

A Python tuple is an immutable, ordered collection of elements. Immutable means that once a tuple is created, its elements cannot be changed or modified. Tuples are similar to lists, but they are enclosed in parentheses () instead of square brackets [].

You can create a tuple in Python by enclosing elements within parentheses (). Here are a few examples of creating tuples:

```
# Creating an empty tuple
empty_tuple = ()

# Creating a tuple with elements
my_tuple = (1, 2, 3, 4, 5)

# Creating a tuple with elements of different types
mixed_tuple = (1, "apple", 3.14, True)

# Creating a single-element tuple (note the comma)
single_element_tuple = (42,)

# You can also create a tuple without parentheses (not recommended for readability)
```

```
another_tuple = 1, 2, 3
```

Tuples are commonly used for grouping data that belongs together, like coordinates, records from a database, or returning multiple values from a function. Because tuples are immutable, they provide a certain level of safety when dealing with data that should not be changed accidentally.

Loop

Looping through all values in a Python tuple is similar to looping through a list. You can use a for loop to iterate over each element in the tuple. Here's how you can do it:

```
my_tuple = (1, 2, 3, 4, 5)

# Loop through all values in the tuple
for value in my_tuple:
    print(value)
```

This loop iterates over each element in the my_tuple tuple and prints each value.

Alternatively, you can also use indexing to loop through a tuple if you need access to the index:

```
my_tuple = (1, 2, 3, 4, 5)

# Loop through all values in the tuple with index
for index in range(len(my_tuple)):
    value = my_tuple[index]
    print(value)
```

However, using the first method with a for loop directly iterating over the tuple elements is generally more Pythonic and preferred.

length

In Python, you can use the len() function to get the length (number of elements) of a tuple. Here's how you can use it:

```
my_tuple = (1, 2, 3, 4, 5)

# Get the length of the tuple
```

```
length = len(my_tuple)

print("Length of the tuple:", length)
```

Output:

```
Length of the tuple: 5
```

The len() function returns the number of elements in the tuple, in this case, 5. This function can be used with tuples as well as other sequences like lists, strings, and dictionaries.

max/min

In Python, you can use the max() and min() functions to find the maximum and minimum elements, respectively, in a tuple. Here's how you can use them:

```
my_tuple = (3, 7, 1, 9, 2, 6)

# Find the maximum element in the tuple
maximum = max(my_tuple)
print("Maximum element:", maximum)

# Find the minimum element in the tuple
minimum = min(my_tuple)
print("Minimum element:", minimum)
```

Output:

```
Maximum element: 9
Minimum element: 1
```

Both max() and min() functions return the maximum and minimum elements of the tuple, respectively. These functions work similarly for other iterable data types like lists and strings as well.

tuple to string

To convert a tuple to a string in Python, you can use string manipulation techniques. One common approach is to use the join() method along with a string concatenation or formatting to convert the elements of the tuple into a string. Here's how you can do it:

Using join() method with string concatenation:

```
my_tuple = (1, 2, 3, 4, 5)

# Convert tuple to string
tuple_string = ', '.join(str(element) for element in my_tuple)

print("Tuple as string:", tuple_string)
```

Using join() method with string formatting:

```
my_tuple = (1, 2, 3, 4, 5)

# Convert tuple to string with formatting
tuple_string = ', '.join(map(str, my_tuple))

print("Tuple as string:", tuple_string)
```

Both examples will output:

```
Tuple as string: 1, 2, 3, 4, 5
```

In both examples, the join() method is used to concatenate the elements of the tuple into a single string. In the first example, we use a generator expression along with string concatenation inside the join() method. In the second example, we use map() to convert each element of the tuple to a string, and then use string formatting inside the join() method.

6 Python Dictionary

Introduction

A Python dictionary is an unordered collection of key-value pairs. Each key in a dictionary is unique and immutable, and it maps to a corresponding value. You can think of a dictionary like a real-world dictionary, where each word (key) has a definition (value).

You can create a dictionary in Python using curly braces {} and specifying key-value pairs separated by colons :. Here's how you can create a dictionary:

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with key-value pairs
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Creating a dictionary using the dict() constructor
another_dict = dict(name='Alice', age=25, city='Los Angeles')
```

In the examples above:

- empty_dict is an empty dictionary.
- my_dict contains three key-value pairs: 'name': 'John', 'age': 30, and 'city': 'New York'.
- another_dict is created using the dict() constructor, where keys and values are specified as keyword arguments.

You can access, modify, add, or remove elements from a dictionary using its keys. Dictionaries are widely used for storing and retrieving data efficiently, especially when you need to access values by their associated keys.

Loop Pairs

You can loop through all key-value pairs in a Python dictionary using a for loop. Python dictionaries provide several methods to access keys, values, or both. Here's how you can loop through all key-value pairs in a dictionary:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Loop through all key-value pairs
```

```
for key, value in my_dict.items():
    print(key, '=', value)
```

In this example, my_dict.items() returns a view object that provides a dynamic view of the dictionary's key-value pairs. The for loop iterates over each key-value pair, and key and value are assigned to each key-value pair respectively. Inside the loop, you can access both the key and value of each pair and perform any desired operations.

Output:

```
name = John
age = 30
city = New York
```

Using .items() is the most common and efficient way to loop through all key-value pairs in a dictionary in Python.

Access

You can access values in a Python dictionary using the keys as indices. Here are a few methods to access values in a dictionary:

1 **Using square brackets []:**

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Accessing values using keys
name = my_dict['name']
age = my_dict['age']
city = my_dict['city']

print("Name:", name)
print("Age:", age)
print("City:", city)
```

2 Using the get() method:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Accessing values using get() method
```

```
name = my_dict.get('name')
age = my_dict.get('age')
city = my_dict.get('city')

print("Name:", name)
print("Age:", age)
print("City:", city)
```

Both methods will output:

```
Name: John
Age: 30
City: New York
```

In both examples, we access the values in the dictionary my_dict using the keys 'name', 'age', and 'city'. Using square brackets is straightforward and raises a KeyError if the key is not found in the dictionary. Using the get() method allows you to provide a default value if the key is not found, which can help avoid KeyError exceptions.

Chapter 3 :

Operators

1 Python Arithmetic Operators

Introduction

In Python, arithmetic operators are used to perform mathematical operations. Here's a list of arithmetic operators in Python:

1 Addition: +

2 Subtraction: -

3 Multiplication: *

4 Division: /

5 Floor Division (integer division): //

6 Modulus (remainder): %

7 Exponentiation: **

You can use these operators to perform arithmetic operations on numbers in Python.

Usage

Here are examples illustrating the use of each arithmetic operator in Python:

1 **Addition (+):**

```
a = 5
b = 3
result = a + b
print("Addition:", result)  # Output: Addition: 8
```

2 **Subtraction (-):**

```
a = 7
b = 4
result = a - b
print("Subtraction:", result)  # Output: Subtraction: 3
```

3 **Multiplication (*):**

```
a = 6
b = 5
result = a * b
print("Multiplication:", result)  # Output: Multiplication: 30
```

4 **Division (/):**

```
a = 10
b = 3
result = a / b
print("Division:", result)  # Output: Division: 3.3333333333333335
```

5 **Floor Division (//):**

```
a = 10
b = 3
result = a // b
print("Floor Division:", result)  # Output: Floor Division: 3
```

6 **Modulus (%):**

```
a = 10
b = 3
result = a % b
print("Modulus:", result)  # Output: Modulus: 1
```

7 **Exponentiation ():****

```
a = 2
b = 3
result = a ** b
print("Exponentiation:", result)  # Output: Exponentiation: 8
```

These examples demonstrate the use of each arithmetic operator in Python with simple numerical values.

Handle Divide By Zero

To handle division by zero and multiple exceptions in Python, you can use the try and except blocks. Here's how you can handle these scenarios:

1 **Handling Divide By Zero:** You can catch the ZeroDivisionError exception when dividing by zero.

2 **Handling Multiple Exceptions:** You can catch multiple exceptions by specifying them in a tuple after except.

Here's an example illustrating both scenarios:

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Result of division:", result)
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    except (TypeError, ValueError):
        print("Error: Invalid input or type mismatch.")
```

```
    except Exception as e:
        print("An unexpected error occurred:", e)

# Example usage
divide_numbers(10, 0)  # Division by zero
divide_numbers(10, 'a')  # Type mismatch
```

In this example:

- The try block attempts to perform the division operation.
- If a ZeroDivisionError occurs (division by zero), the first except block is executed.
- If a TypeError or ValueError occurs (for example, if b is not a valid number), the second except block is executed.
- If any other unexpected error occurs, it's caught by the last except block, which captures all exceptions.
- The as keyword is used to capture the exception object, which can be useful for debugging.

You can customize the error messages and handling logic based on your requirements within each except block.

Integer Division

In Python, integer division is performed using the double-slash operator //. This operator divides one number by another and returns the integer result, discarding any fractional part.

Here's an example:

```
result = 10 // 3
print(result)  # Output: 3
```

In this example, 10 // 3 performs integer division, resulting in 3.

Now, regarding the remainder operator %, it returns the remainder of the division operation. It's useful when you want to find out if one number is divisible by another, and if not, what the remainder is.

Here's an example:

```
remainder = 10 % 3
```

```
print(remainder)  # Output: 1
```

In this example, 10 % 3 returns 1 because 10 divided by 3 leaves a remainder of 1.

You might use the remainder operator in various scenarios, such as checking if a number is even or odd (number % 2 == 0 for even numbers), or for cycling through a list of elements in a round-robin manner. It's a versatile operator that finds use in many mathematical and programming scenarios.

Common Divisors

You can find the common divisors of two numbers in Python by iterating through the range of numbers from 1 to the minimum of the two numbers and checking if both numbers are divisible by that number without remainder. Here's how you can do it:

```
def common_divisors(num1, num2):
    # Find the minimum of the two numbers
    min_num = min(num1, num2)

    # Initialize a list to store common divisors
    common_divisors_list = []

    # Iterate through the range from 1 to min_num
    for i in range(1, min_num + 1):
        # Check if both numbers are divisible by i without remainder
        if num1 % i == 0 and num2 % i == 0:
          common_divisors_list.append(i)

    return common_divisors_list

# Example usage
num1 = 12
num2 = 18
result = common_divisors(num1, num2)
```

```
print("Common divisors of", num1, "and", num2, ":", result)
```

In this example:

- The common_divisors function takes two integers num1 and num2 as input.
- It finds the minimum of the two numbers to determine the range of the loop.
- It initializes an empty list common_divisors_list to store the common divisors.
- It iterates through the range from 1 to the minimum of the two numbers.
- For each number i, it checks if both num1 and num2 are divisible by i without remainder.
- If they are, i is added to the common_divisors_list.
- Finally, it returns the list of common divisors.

You can use this function to find the common divisors of any two numbers by passing them as arguments to the function.

Sum of squares

To find the sum of squares of the first n natural numbers in Python, you can use a simple formula and compute the sum directly. The formula to find the sum of squares of the first n natural numbers is given by:

[\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}]

Here's how you can implement it in Python:

```
def sum_of_squares_of_first_n(n):
    return n * (n + 1) * (2 * n + 1) // 6
```

```
# Example usage
n = 5
result = sum_of_squares_of_first_n(n)
print("Sum of squares of the first", n, "natural numbers:", result)
```

In this implementation:

- The sum_of_squares_of_first_n function takes an integer n as input.
- It computes the sum of squares using the formula (\frac{n(n+1)(2n+1)}{6}).
- The result is returned.

You can adjust the value of n to find the sum of squares of the first n natural numbers for any value of n.

2 Python Bitwise Operators

Introduction

In Python, bitwise operators are used to perform bitwise operations on integers. Here's a list of bitwise operators in Python:

1 **Bitwise AND (&):** Performs a bitwise AND operation on the corresponding bits of two integers.

2 **Bitwise OR (|):** Performs a bitwise OR operation on the corresponding bits of two integers.

3 **Bitwise XOR (^):** Performs a bitwise XOR (exclusive OR) operation on the corresponding bits of two integers.

4 **Bitwise NOT (~):** Performs a bitwise NOT (complement) operation, which inverts all the bits of an integer.

5 **Left Shift (<<):** Shifts the bits of an integer to the left by a specified number of positions, filling the shifted positions with zeros.

6 **Right Shift (>>):** Shifts the bits of an integer to the right by a specified number of positions, filling the shifted positions with zeros or the sign bit (for signed integers).

These operators work at the bit level, manipulating individual bits within integer values. They are commonly used in low-level programming, hardware manipulation, and optimization tasks.

Usage

Here are examples illustrating the use of each bitwise operator in Python:

1 **Bitwise AND (&):**

```
a = 5  # 0101 in binary
b = 3  # 0011 in binary

result = a & b
print("Bitwise AND:", result)  # Output: 1 (0001 in binary)
```

2 **Bitwise OR (|):**

```
a = 5  # 0101 in binary
b = 3  # 0011 in binary

result = a | b
print("Bitwise OR:", result)  # Output: 7 (0111 in binary)
```

3 **Bitwise XOR (^):**

```
a = 5  # 0101 in binary
b = 3  # 0011 in binary

result = a ^ b
print("Bitwise XOR:", result)  # Output: 6 (0110 in binary)
```

4 **Bitwise NOT (~):**

```
a = 5  # 0101 in binary

result = ~a
print("Bitwise NOT:", result)  # Output: -6 (1010 in binary for 32-bit integers)
```

5 **Left Shift (<<):**

```
a = 5  # 0101 in binary

result = a << 2
print("Left Shift:", result)  # Output: 20 (10100 in binary)
```

6 **Right Shift (>>):**

```
a = 10  # 1010 in binary

result = a >> 1
print("Right Shift:", result)  # Output: 5 (0101 in binary)
```

These examples demonstrate the use of each bitwise operator in Python. You can experiment with different values to see how the operators behave with various inputs.

Power of 2

You can determine whether a number is a power of two using bitwise operators in Python. A number is a power of two if and only if it has exactly one bit set in its binary representation. Here's how you can do it:

```
def is_power_of_two(num):
    # A number is a power of two if it is greater than 0 and has exactly one bit set
    return num > 0 and (num & (num - 1)) == 0

# Example usage
num1 = 4
num2 = 6

print(num1, "is a power of two:", is_power_of_two(num1))  # Output: True
print(num2, "is a power of two:", is_power_of_two(num2))  # Output: False
```

In this example:

- The is_power_of_two function checks if a given number is a power of two.
- It first checks if the number is greater than 0 (since negative numbers cannot be powers of two).
- Then, it checks if the number has exactly one bit set by performing a bitwise AND operation between the number and its predecessor (i.e., num & (num - 1)). If the result is 0, it

means that only one bit is set, indicating that the number is a power of two.

- The function returns True if the number is a power of two, and False otherwise.
- The example usage demonstrates checking whether numbers 4 and 6 are powers of two. Adjust the numbers as needed to test other values.

Sum odd even digit

You can sum the odd and even digits of a number in Python by iterating through each digit of the number, checking if it's odd or even, and then summing them separately. Bitwise operators are not typically used for this task as it's more straightforward to accomplish with arithmetic and modulus operators. Here's how you can do it:

```
def sum_odd_even_digits(num):
    odd_sum = 0
    even_sum = 0

    # Iterate through each digit of the number
    while num > 0:
        digit = num % 10
        if digit % 2 == 0:
            even_sum += digit
        else:
            odd_sum += digit
        num //= 10

    return odd_sum, even_sum

# Example usage
number = 123456

odd_sum, even_sum = sum_odd_even_digits(number)
print("Sum of odd digits:", odd_sum)
print("Sum of even digits:", even_sum)
```

In this example:

- The sum_odd_even_digits function takes a number (num) as input.

- It initializes variables odd_sum and even_sum to store the sums of odd and even digits, respectively.
- It iterates through each digit of the number by repeatedly dividing the number by 10 and extracting the last digit using the modulus operator.
- It checks if each digit is odd or even using the modulus operator (%).
- It adds the digit to the corresponding sum based on whether it's odd or even.
- The function returns the sums of odd and even digits.
- The example usage demonstrates summing the odd and even digits of the number 123456. You can pass any integer to the function to sum the odd and even digits of that number.

3 Python Comparison Operators

Introduction

In Python, comparison operators are used to compare values and return Boolean results (True or False) based on the comparison. Here's a list of comparison operators in Python:

1 **Equal to (==):** Returns True if the operands are equal.

1 **Not equal to (!=):** Returns True if the operands are not equal.

3 **Greater than (>):** Returns True if the left operand is greater than the right operand.

4 **Less than (<):** Returns True if the left operand is less than the right operand.

5 **Greater than or equal to (>=):** Returns True if the left operand is greater than or equal to the right operand.

6 **Less than or equal to (<=):** Returns True if the left operand is less than or equal to the right operand.

These operators are commonly used in conditional statements, loops, and other contexts where comparison of values is needed.

Usage

Here are examples illustrating the use of each comparison operator in Python:

1 Equal to (==):

```
a = 5
b = 5

result = a == b
print("Is", a, "equal to", b, "?", result)  # Output: Is 5 equal to 5 ? True
```

2 Not equal to (!=):

```
a = 5
b = 6

result = a != b
print("Is", a, "not equal to", b, "?", result)  # Output: Is 5 not equal to 6 ? True
```

3 Greater than (>):

```
a = 6
b = 5

result = a > b
print("Is", a, "greater than", b, "?", result)  # Output: Is 6 greater than 5 ? True
```

4 Less than (<):

```
a = 5
b = 6

result = a < b
print("Is", a, "less than", b, "?", result)  # Output: Is 5 less than 6 ? True
```

5 Greater than or equal to (>=):

```
a = 6
b = 6
```

```
result = a >= b
print("Is", a, "greater than or equal to", b, "?", result)  # Output: Is 6 greater than or equal to 6 ? True
```

6 **Less than or equal to (<=):**

```
a = 5
b = 6

result = a <= b
print("Is", a, "less than or equal to", b, "?", result)  # Output: Is 5 less than or equal to 6 ? True
```

These examples demonstrate the use of each comparison operator in Python. You can change the values of a and b to test different comparisons.

4 Python Logical Operators

Introduction

In Python, logical operators are used to combine multiple conditions and produce a single Boolean result (True or False). Here's a list of logical operators in Python:

1 **Logical AND (and):** Returns True if both operands are True.

2 **Logical OR (or):** Returns True if at least one of the operands is True.

3 **Logical NOT (not):** Returns True if the operand is False, and False if the operand is True.

These operators are commonly used in conditional statements, loops, and other contexts where logical operations are needed.

Usage

Here are examples illustrating the use of each logical operator in Python:

1 **Logical AND (and):**

```
a = True
b = False

result = a and b
```

```
print("Logical AND:", result)  # Output: Logical AND: False
```

2 **Logical OR (or):**

```
a = True
b = False

result = a or b
print("Logical OR:", result)  # Output: Logical OR: True
```

3 **Logical NOT (not):**

```
a = True

result = not a
print("Logical NOT:", result)  # Output: Logical NOT: False
```

These examples demonstrate the use of each logical operator in Python. You can change the values of a and b to test different logical operations.

Combine

You can combine logical operators to create more complex conditions in Python by using parentheses to group expressions and applying logical operators (and, or, not) as needed. Here's an example that demonstrates combining logical operators:

```
# Example: Check if a number is between 10 and 20, or outside the range 30 to 40

def check_number_range(num):
    if (num >= 10 and num <= 20) or (num < 30 or num > 40):
        return True
    else:
        return False

# Example usage
number1 = 15
number2 = 25
number3 = 35
```

```
print("Is", number1, "between 10 and 20 or outside the range 30 to 40?",
check_number_range(number1))  # Output: True
print("Is", number2, "between 10 and 20 or outside the range 30 to 40?",
check_number_range(number2))  # Output: False
print("Is", number3, "between 10 and 20 or outside the range 30 to 40?",
check_number_range(number3))  # Output: True
```

In this example:

- The check_number_range function takes a number (num) as input and checks if it meets the specified conditions.
- The condition is defined using logical operators:
- (num >= 10 and num <= 20) checks if the number is between 10 and 20.
- (num < 30 or num > 40) checks if the number is outside the range 30 to 40.
- The or operator is used to combine these two conditions.
- The and operator is used within the first condition to check if the number satisfies both the lower and upper bounds of the range.

- Parentheses are used to group expressions and ensure the correct evaluation of the conditions.
- The function returns True if the number meets the conditions and False otherwise.
- The example usage demonstrates checking whether three different numbers meet the specified conditions. Adjust the numbers and conditions as needed for your specific requirements.

Greatest of three numbers

You can create a Python program to find the greatest of three numbers using logical operators (and and or) to compare the numbers. Here's how you can do it:

```
def find_greatest(num1, num2, num3):
```

```
    if num1 >= num2 and num1 >= num3:
        return num1
    elif num2 >= num1 and num2 >= num3:
        return num2
    else:
        return num3

# Example usage
number1 = 10
number2 = 20
number3 = 15

greatest = find_greatest(number1, number2, number3)
print("The greatest of the three numbers is:", greatest)
```

In this program:

- The find_greatest function takes three numbers (num1, num2, num3) as input.
- It compares the numbers using logical operators (and and or) to find the greatest among them.
- If num1 is greater than or equal to both num2 and num3, it returns num1 as the greatest.
- If num2 is greater than or equal to both num1 and num3, it returns num2 as the greatest.
- Otherwise, it returns num3 as the greatest.
- The example usage demonstrates finding the greatest of three numbers (10, 20, and 15). You can adjust the values of number1, number2, and number3 to find the greatest of any three numbers.

5 Python Ternary Operators

Introduction

The ternary operator in Python is a conditional expression that evaluates a condition and returns one of two values depending on whether the condition is true or false. It is also known as the conditional expression. The syntax of the ternary operator in Python is as follows:

```
x = <value_if_true> if <condition> else <value_if_false>
```

Here's how the ternary operator works:

- <condition> is evaluated first. If it is true, <value_if_true> is returned; otherwise, <value_if_false> is returned.
- The ternary operator is an expression, not a statement, so it can be used within larger expressions or assignments.

Here's an example of using the ternary operator in Python:

```
x = 10
y = 20

max_value = x if x > y else y
print("Maximum value:", max_value)  # Output: Maximum value: 20
```

In this example:

- The condition x > y is evaluated first. If it is true, x is assigned to max_value; otherwise, y is assigned.
- Since x (10) is not greater than y (20), the value of y (20) is assigned to max_value.
- The ternary operator makes the code concise and readable, especially in situations where you need to choose between two values based on a condition.

Usage

Here are examples illustrating the use of the ternary operator in Python:

1 **Assigning Maximum Value:**

```
```

```
x = 10
y = 20

max_value = x if x > y else y
print("Maximum value:", max_value)  # Output: Maximum value: 20
```

2 Checking Even or Odd:

```
num = 15

result = "Even" if num % 2 == 0 else "Odd"
print(num, "is", result)  # Output: 15 is Odd
```

3 Assigning Absolute Value:

```
number = -10

absolute_value = number if number >= 0 else -number
print("Absolute value:", absolute_value)  # Output: Absolute value: 10
```

4 Checking for Validity:

```
username = "john_doe"

message = "Valid" if len(username) >= 8 else "Invalid"
print("Username is", message)  # Output: Username is Valid
```

5 Handling Division by Zero:

```
dividend = 10
divisor = 0

result = dividend / divisor if divisor != 0 else "Error: Division by Zero"
print("Result:", result)  # Output: Result: Error: Division by Zero
```

These examples demonstrate the versatility of the ternary operator in Python for making concise conditional assignments or expressions.

Largest

You can create a Python program to find the largest of two numbers entered by the user using ternary operators as follows:

```
# Input two numbers from the user
```

```
number1 = float(input("Enter first number: "))
number2 = float(input("Enter second number: "))

# Determine the largest number using ternary operator
largest = number1 if number1 > number2 else number2

# Display the result
print("The largest number is:", largest)
```

In this program:

- The input function is used to get two numbers as input from the user.
- Ternary operator is used to determine the largest of the two numbers.
- If number1 is greater than number2, number1 is assigned to largest; otherwise, number2 is assigned.
- The result is displayed to the user.

Chapter 4 :

Statements

1 Python if

Introduction

In Python, the if statement is a control flow statement that allows you to execute a block of code based on whether a specified condition evaluates to True. It is used for decision-making in programming.

The syntax of the if statement in Python is as follows:

```
if condition:
    # Code block to execute if the condition is True
    statement1
    statement2
    ...
```

Here's how the if statement works:

- The condition is evaluated. If it is True, the code block following the if statement is executed. If it is False, the code block is skipped.
- The code block under the if statement is typically indented to indicate that it belongs to the if statement. The indentation is usually four spaces, but it can be any consistent whitespace.
- Optionally, you can include an else statement to specify a block of code to execute if the condition is False. Additionally, you can use elif (short for "else if") to specify additional conditions to check.

Here's an example of using the if statement in Python:

```
x = 10

if x > 5:
    print("x is greater than 5")
```

In this example:

- The condition x > 5 is evaluated. Since x is 10, which is greater than 5, the condition is True.
- Therefore, the code block under the if statement is executed, and the message "x is greater than 5" is printed to the console.

Syntax

The if statement in Python can be used in various ways to create conditional logic. Here are the different forms of if statements along with their syntax:

1 **Basic if statement:**

```
if condition:
    # Code block to execute if the condition is True
    statement1
    statement2
    ...
```

2 if-else statement:

```
if condition:
    # Code block to execute if the condition is True
    statement1
    statement2
    ...
else:
    # Code block to execute if the condition is False
    statement3
    statement4
    ...
```

3 Chained if-elif-else statement:

```
if condition1:
    # Code block to execute if condition1 is True
    statement1
    statement2
    ...
elif condition2:
    # Code block to execute if condition2 is True
    statement3
    statement4
    ...
elif condition3:
    # Code block to execute if condition3 is True
    statement5
    statement6
    ...
else:
    # Code block to execute if all conditions are False
    statement7
    statement8
    ...
```

4 Nested if statement:

```
if condition1:
```

```
    if condition2:
        # Code block to execute if both condition1 and condition2 are True
        statement1
        statement2
        ...
```

In each form of the if statement, you can include one or more conditions and corresponding code blocks to execute based on the evaluation of those conditions. The else and elif clauses are optional and can be used to specify alternate code blocks to execute when the conditions are False.

Even Odd

You can check whether a given number is even or odd in Python using an if statement with the modulo operator (%). Here's how you can do it:

```
def check_even_odd(number):
    if number % 2 == 0:
        print(number, "is even.")
    else:
        print(number, "is odd.")

# Example usage
number = 7
check_even_odd(number)
```

In this program:

- The check_even_odd function takes a number (number) as input.
- It checks if the number is even or odd using the modulo operator (%). If the remainder when dividing the number by 2 is 0, the number is even; otherwise, it is odd.
- If the number is even (number % 2 == 0), it prints a message indicating that the number is even. Otherwise, it prints a message indicating that the number is odd.
- The example usage demonstrates checking whether the number 7 is even or odd. You can pass any integer to the function to check if it's even or odd.

Largest/smallest

You can read in three integers from the user, then use if statements to determine the largest and smallest integers among them. Here's how you can do it:

```
# Read in three integers from the user
num1 = int(input("Enter first integer: "))
num2 = int(input("Enter second integer: "))
num3 = int(input("Enter third integer: "))

# Assume the first number is both the largest and smallest initially
largest = num1
smallest = num1

# Determine the largest integer
if num2 > largest:
    largest = num2
if num3 > largest:
    largest = num3

# Determine the smallest integer
if num2 < smallest:
    smallest = num2
if num3 < smallest:
    smallest = num3

# Print the largest and smallest integers
print("Largest integer:", largest)
print("Smallest integer:", smallest)
```

In this program:

- We first read in three integers (num1, num2, num3) from the user using the input function, and convert them to integers using the int function.
- We initialize the variables largest and smallest to the value of num1, assuming initially that num1 is both the largest and smallest integer.
- We then use if statements to compare num2 and num3 with largest to determine the largest integer, and similarly compare them with smallest to determine the smallest integer.

- Finally, we print the largest and smallest integers to the console.

Positive or negative

You can create a Python program to check whether a given integer is positive or negative using an if statement like this:

```
# Take input from the user
num = int(input("Enter an integer: "))

# Check if the number is positive, negative, or zero
if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

In this code:

- We take an integer input from the user using the input() function.
- Then, we convert the input into an integer using the int() function.
- We use the if, elif, and else statements to check whether the number is positive, negative, or zero.
- If the number is greater than 0, it's positive. If it's less than 0, it's negative. Otherwise, it's zero.

2 Python while

Syntax

The syntax of a while loop in Python is as follows:

```
while condition:
```

```
    # Code block to be executed repeatedly
    # as long as the condition is True
```

In this syntax:

- condition is an expression that is evaluated before each iteration of the loop. If the condition evaluates to True, the loop body is executed. If the condition evaluates to False, the loop terminates.
- The colon (:) at the end of the while statement indicates the start of the indented block of code that will be executed repeatedly as long as the condition is True.
- The code inside the loop block is indented to signify that it belongs to the loop body.
- The loop continues to execute the code block repeatedly until the condition becomes False. If the condition never becomes False, you can end up with an infinite loop.

break

You can use the break statement to exit a while loop prematurely in Python. Here's how you can use it:

```
while condition:
    # Code block to be executed repeatedly
    # as long as the condition is True
    if some_condition:
        break  # Exit the loop if some condition is met
```

In this code:

- Inside the while loop, there's an if statement that checks for a certain condition (some_condition).
- If some_condition evaluates to True, the break statement is executed.

- When the break statement is encountered, it immediately exits the while loop, regardless of whether the loop's condition is still True.
- After the break statement is executed, the program continues to execute the code that follows the loop.

continue

You can use the continue statement to skip the current iteration of a while loop and proceed to the next iteration. Here's how you can use it:

```
while condition:
    # Code block to be executed repeatedly
    # as long as the condition is True

    if some_condition:
        continue  # Skip the rest of the loop's code and start the next iteration

    # Code here will be skipped if some_condition is True
```

In this code:

- Inside the while loop, there's an if statement that checks for a certain condition (some_condition).
- If some_condition evaluates to True, the continue statement is executed.
- When the continue statement is encountered, it skips the remaining code in the loop's block for the current iteration and moves on to the next iteration of the loop.
- If some_condition is False, the code after the if block will be executed normally for that iteration.

3 Python for loop

Introduction

In Python, a for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object. The for loop executes a block of code multiple times, once for each item in the sequence or iterable.

The syntax of a for loop in Python is as follows:

```
for item in sequence:
    # Code block to be executed for each item in the sequence
```

In this syntax:

- item is a variable that takes on the value of each item in the sequence during each iteration of the loop.
- sequence is the sequence of elements over which the loop iterates. It can be any iterable object, such as a list, tuple, string, or range.
- The colon (:) at the end of the for statement indicates the start of the indented block of code that will be executed for each item in the sequence.
- The code inside the loop block is indented to signify that it belongs to the loop body.
- During each iteration of the loop, the item variable takes on the value of the next element in the sequence, and the code block inside the loop is executed with that value.

Here's a simple example of using a for loop to iterate over a list:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

In this example, the for loop iterates over each item in the fruits list, and during each iteration, the fruit variable takes on the value of the current item, which is then printed.

Syntax

In Python, the for loop can be used in different ways depending on the type of iterable you are iterating over or the specific requirement of the loop. Here are some common variations of for loop statements:

1 Iterating over a sequence (list, tuple, string, etc.):

```
sequence = [1, 2, 3, 4, 5]
for item in sequence:
    # Code block to be executed for each item in the sequence
```

2 Iterating over a range of numbers:

```
for i in range(start, stop, step):
    # Code block to be executed for each value of i in the range
```

Here, start is the starting value of the range (inclusive), stop is the ending value of the range (exclusive), and step is the step size between each value (default is 1).

3 Iterating over a sequence with index using enumerate():

```
sequence = ["a", "b", "c", "d"]
for index, value in enumerate(sequence):
    # Code block to be executed for each (index, value) pair
```

4 Iterating over multiple sequences simultaneously using zip():

```
sequence1 = [1, 2, 3]
sequence2 = ["a", "b", "c"]
for item1, item2 in zip(sequence1, sequence2):
    # Code block to be executed for each corresponding pair of items
```

5 Iterating over dictionary keys, values, or items:

```
dictionary = {"a": 1, "b": 2, "c": 3}
# Iterating over keys
for key in dictionary:
    # Code block to be executed for each key
# Iterating over values
for value in dictionary.values():
```

```
    # Code block to be executed for each value
# Iterating over key-value pairs
for key, value in dictionary.items():
    # Code block to be executed for each (key, value) pair
```

These are some of the common ways you can use the for loop in Python, but there can be more variations depending on the specific requirements of your code.

Loop on array

In Python, you can print each element in an array (or list) using a for loop. Here's how you can do it:

```
# Define an array (or list)
my_array = [1, 2, 3, 4, 5]

# Iterate over each element in the array using a for loop
for element in my_array:
    print(element)
```

In this code:

- We define an array (or list) called my_array containing some elements [1, 2, 3, 4, 5].
- We use a for loop to iterate over each element in the array.
- During each iteration, the element variable takes on the value of the current element in the array.
- Inside the loop, we print each element using the print() function.

This will print each element of the array on a new line. If you want to print them on the same line, you can use the end parameter of the print() function, like this:

```
# Print each element on the same line
for element in my_array:
    print(element, end=" ")
```

This will print each element separated by a space on the same line.

4 Python pass

Introduction

In Python, the pass statement is a null operation. It doesn't do anything, and it acts as a placeholder when a statement is syntactically required but you don't want to execute any code.

You might use the pass statement in the following situations:

1 **Placeholder for future code:** You can use pass as a placeholder when you want to indicate that a block of code will be implemented later, but you want to have a syntactically correct placeholder in the meantime.

```
if condition:
    # To be implemented later
    pass
```

2 **Empty function or class:** When you define a function or a class that doesn't contain any code yet, you can use pass to indicate that it's intentionally empty.

```
def my_function():
    pass

class MyClass:
    pass
```

3 **Implementing a placeholder for a loop or conditional block:** Sometimes, when designing your code, you might want to include a loop or conditional block that doesn't do anything yet. In such cases, you can use pass as a placeholder.

```
for item in my_list:
```

```
    # Placeholder for future
code
    pass

if condition:
    # Placeholder for future
code
    pass
```

Using pass allows your code to remain syntactically correct while providing a clear indication that the block of code is intentionally left empty or will be implemented later.

5 Python break

Introduction

In Python, the break statement is used to exit (or "break out of") a loop prematurely. When a break statement is encountered inside a loop (such as a for loop or a while loop), the loop is immediately terminated, and the program execution continues from the statement immediately following the loop.

Here's a basic example of how break statement works in a loop:

```
for i in range(5):
    print(i)
    if i == 2:
        break
```

In this example, the loop iterates over the range from 0 to 4. When i becomes equal to 2, the if condition i == 2 becomes true, and the break

statement is executed. As a result, the loop is terminated prematurely, and the program execution continues after the loop.

Output:

```
0
1
2
```

After i reaches 2, the loop is terminated, and the statement print("Loop terminated") (if present) would be executed next.

The break statement is often used in combination with conditional statements (if statements) inside loops to exit the loop based on certain conditions. It provides a way to terminate the loop prematurely when a specific condition is met, which can be useful for optimization or for handling special cases.

Syntax

In Python, the break statement is used within loops to exit the loop prematurely. Here are some common ways to use the break statement with different types of loops:

1 **Using break in a for loop:**

```
for item in iterable:
    # Code block
    if condition:
        break
```

2 **Using break in a while loop:**

```
while condition:
    # Code block
    if condition:
        break
```

In both cases:

- The break statement is used to exit the loop immediately when a certain condition is met.
- It can be placed inside a conditional statement (if statement) to define the condition under which the loop should be terminated.
- Once the break statement is executed, the loop is terminated, and the program execution continues from the statement immediately following the loop.

Here's an example demonstrating the usage of break in both types of loops:

```
# Using break in a for loop
for i in range(5):
    print(i)
    if i == 2:
        break

# Using break in a while loop
i = 0
while i < 5:
    print(i)
    if i == 2:
        break
    i += 1
```

Output:

```
0
1
2
0
1
2
```

In both cases, the loop terminates prematurely when i becomes equal to 2 due to the break statement.

break for

In Python, you can use the break statement within a for loop to exit the loop prematurely based on a certain condition. Here's how you can use the break statement in a for loop:

```
# Example: Using break in a for loop
for item in iterable:
    # Code block
    if condition:
        break
```

In this syntax:

- iterable is the sequence or iterable object over which the loop iterates.
- item is the loop variable that takes on the value of each item in the iterable during each iteration.
- The if statement with condition inside the loop block checks if a certain condition is met.
- If the condition evaluates to True, the break statement is executed, causing the loop to terminate immediately, and the program execution continues from the statement immediately following the loop.

Here's a simple example demonstrating the usage of break in a for loop:

```
# Example: Using break in a for loop
fruits = ["apple", "banana", "cherry", "date"]
for fruit in fruits:
    print(fruit)
    if fruit == "cherry":
        break
```

Output:

```
apple
banana
cherry
```

In this example, the loop terminates prematurely when the fruit variable becomes equal to "cherry" due to the break statement.

break nested for

In Python, you can use the break statement within a nested for loop to exit the inner loop prematurely based on a certain condition. Here's how you can use the break statement in a nested for loop:

```
# Example: Using break in a nested for loop
for outer_item in outer_iterable:
    # Outer loop code block
    for inner_item in inner_iterable:
        # Inner loop code block
        if condition:
      break
    if condition:
        break
```

In this syntax:

- outer_iterable is the sequence or iterable object over which the outer loop iterates.
- inner_iterable is the sequence or iterable object over which the inner loop iterates.
- outer_item and inner_item are the loop variables that take on the value of each item in the outer_iterable and inner_iterable, respectively, during each iteration.
- The if statement with condition inside the inner loop block checks if a certain condition is met. If the condition evaluates to True, the break statement is executed, causing the inner loop to terminate immediately.
- Similarly, you can use the break statement in the outer loop to exit both the inner and outer loops prematurely if needed.

Here's a simple example demonstrating the usage of break in a nested for loop:

```
# Example: Using break in a nested for loop
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for element in row:
        print(element)
        if element == 5:
```

```
        break
    else:
        continue
    break
```

Output:

```
1
2
3
4
5
```

In this example, the inner loop terminates prematurely when the element variable becomes equal to 5 due to the break statement. The outer loop then terminates immediately because of the break statement outside the inner loop.

break while

In Python, you can use the break statement within a while loop to exit the loop prematurely based on a certain condition. Here's how you can use the break statement in a while loop:

```
# Example: Using break in a while loop
while condition:
    # Code block
    if condition:
        break
```

In this syntax:

- condition is the expression that determines whether the loop should continue executing or not.
- The while loop continues to execute as long as the condition evaluates to True.

- Inside the loop block, the if statement with condition checks if a certain condition is met.
- If the condition evaluates to True, the break statement is executed, causing the loop to terminate immediately, and the program execution continues from the statement immediately following the loop.

Here's a simple example demonstrating the usage of break in a while loop:

```
# Example: Using break in a while loop
i = 0
while i < 5:
    print(i)
    if i == 2:
        break
    i += 1
```

Output:

```
0
1
2
```

In this example, the loop terminates prematurely when i becomes equal to 2 due to the break statement.

6 Python continue

Introduction

In Python, the continue statement is used inside loops (such as for loops and while loops) to skip the rest of the code inside the loop for the current iteration and continue with the next iteration of the loop.

Here's a basic syntax of the continue statement:

```
for item in iterable:
    # Code block
    if condition:
        continue
    # More code
```

Or:

```
while condition:
    # Code block
    if condition:
        continue
    # More code
```

In this syntax:

- iterable is the sequence or iterable object over which the loop iterates in the case of a for loop, or condition is the expression that determines whether the loop should continue executing in the case of a while loop.
- The if statement with condition inside the loop block checks if a certain condition is met.
- If the condition evaluates to True, the continue statement is executed, causing the rest of the code inside the loop for the current iteration to be skipped, and the loop proceeds to the next iteration.

Here's a simple example demonstrating the usage of the continue statement in a for loop:

```
# Example: Using continue in a for loop
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output:

```
0
1
3
4
```

In this example, when i becomes equal to 2, the continue statement is executed, and the rest of the code inside the loop for that iteration (i.e., print(i)) is skipped. The loop proceeds to the next iteration.

Syntax

In Python, the continue statement is used within loops (such as for loops and while loops) to skip the rest of the code inside the loop for the current iteration and continue with the next iteration of the loop. The syntax for using the continue statement varies slightly depending on the type of loop:

1 **Using continue in a for loop:**

```
for item in iterable:
    # Code block
    if condition:
        continue
    # More code
```

2 **Using continue in a while loop:**

```
while condition:
    # Code block
    if condition:
        continue
    # More code
```

In both cases:

- iterable is the sequence or iterable object over which the loop iterates in the case of a for loop, or condition is the expression that determines whether the loop should continue executing in the case of a while loop.

- The if statement with condition inside the loop block checks if a certain condition is met.
- If the condition evaluates to True, the continue statement is executed, causing the rest of the code inside the loop for the current iteration to be skipped, and the loop proceeds to the next iteration.

Here's a simple example demonstrating the usage of the continue statement in both types of loops:

```
# Example: Using continue in a for loop
for i in range(5):
    if i == 2:
        continue
    print(i)

# Example: Using continue in a while loop
i = 0
while i < 5:
    if i == 2:
        i += 1
        continue
    print(i)
    i += 1
```

Output:

```
0
1
3
4
0
1
3
4
```

In both examples, when i becomes equal to 2, the continue statement is executed, causing the rest of the code inside the loop for that iteration to be skipped, and the loop proceeds to the next iteration.

continue for

To use the continue statement within a for loop in Python, you can place it inside the loop's block to skip the remaining code for the current iteration and move to the next iteration. Here's how you can use the continue statement in a for loop:

```
# Example: Using continue in a for loop
for item in iterable:
    # Code block
    if condition:
        continue
    # More code
```

In this syntax:

- iterable is the sequence or iterable object over which the loop iterates.
- item is the loop variable that takes on the value of each item in the iterable during each iteration.
- The if statement with condition inside the loop block checks if a certain condition is met.
- If the condition evaluates to True, the continue statement is executed, causing the rest of the code inside the loop for the current iteration to be skipped, and the loop proceeds to the next iteration.

Here's a simple example demonstrating the usage of the continue statement in a for loop:

```
# Example: Using continue in a for loop
fruits = ["apple", "banana", "cherry", "date"]
for fruit in fruits:
    if fruit == "banana":
        continue
    print(fruit)
```

Output:

```
apple
cherry
date
```

In this example, when the fruit variable becomes equal to "banana", the continue statement is executed, causing the rest of the code inside the loop for that iteration to be skipped. The loop proceeds to the next iteration, and the remaining fruits are printed.

continue while

To use the continue statement within a while loop in Python, you can place it inside the loop's block to skip the remaining code for the current iteration and move to the next iteration. Here's how you can use the continue statement in a while loop:

```
# Example: Using continue in a while loop
while condition:
    # Code block
    if condition:
        continue
    # More code
```

In this syntax:

- condition is the expression that determines whether the loop should continue executing.
- The while loop continues to execute as long as the condition evaluates to True.
- Inside the loop block, the if statement with condition checks if a certain condition is met.
- If the condition evaluates to True, the continue statement is executed, causing the rest of the code inside the loop for the current iteration to be skipped, and the loop proceeds to the next iteration.

Here's a simple example demonstrating the usage of the continue statement in a while loop:

```
# Example: Using continue in a while loop
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

Output:

```
1
2
4
5
```

In this example, when i becomes equal to 3, the continue statement is executed, causing the rest of the code inside the loop for that iteration to be skipped. The loop proceeds to the next iteration, and the remaining values of i are printed.

Chapter 5 :

Functions

1 Python function

Introduction

In Python, a function is a block of reusable code that performs a specific task. Functions provide a way to organize code into manageable pieces, improve code reusability, and make the code more readable.

Here's the syntax to define and use a function in Python:

1 **Defining a function:**

```
def function_name(parameter1, parameter2, ...):
```

```
    # Function body
    # Code to perform the task
    return result
```

- def keyword is used to define a function.
- function_name is the name of the function.
- (parameter1, parameter2, ...) is the list of parameters (if any) that the function accepts. Parameters are optional.
- : colon indicates the beginning of the function body.
- The function body consists of the code that performs the task.
- return result statement (optional) is used to return a value from the function. If there's no return statement, the function returns None by default.

2 **Using a function:**

```
result = function_name(argument1, argument2, ...)
```

- function_name is the name of the function to call.
- (argument1, argument2, ...) is the list of arguments (if any) to pass to the function. Arguments are optional.
- The function is called with the specified arguments, and the result (if any) is stored in the result variable.

Here's an example of defining and using a simple function in Python:

```
# Function definition
def add_numbers(x, y):
    # Function body: adds two numbers
    result = x + y
    return result

# Using the function
sum_result = add_numbers(3, 5)
print("Sum:", sum_result)  # Output: Sum: 8
```

In this example:

- We define a function add_numbers that takes two parameters x and y, adds them together, and returns the result.
- We call the function add_numbers with arguments 3 and 5, and store the result in the variable sum_result.
- We print the result to the console.

Parameters

In Python, function parameters are variables that are specified in the function definition and are used to pass values to the function when it is called. Parameters allow functions to accept input values and perform operations on them.

Here's the syntax to define and use parameters in Python functions:

1 **Defining parameters:**

```
def function_name(parameter1, parameter2, ...):
    # Function body
    # Code that uses parameter1, parameter2, ...
```

- def keyword is used to define a function.
- function_name is the name of the function.
- (parameter1, parameter2, ...) is the list of parameters that the function accepts. Parameters are optional.
- Each parameter in the list is a variable that holds the value passed to the function.
- Parameters are separated by commas.

2 **Using parameters:**

```
result = function_name(argument1, argument2, ...)
```

- function_name is the name of the function to call.
- (argument1, argument2, ...) is the list of arguments that are passed to the function. Arguments are optional.
- Arguments are the actual values that are passed to the function when it is called.

- The function uses these arguments to perform its task.

Here's an example demonstrating the usage of parameters in Python functions:

```
# Function definition with parameters
def greet(name):
    # Function body: greets the user with the provided name
    print("Hello,", name, "!")

# Using the function with an argument
greet("Alice")  # Output: Hello, Alice!
```

In this example:

- We define a function greet that takes a single parameter name.
- When the function is called with the argument "Alice", the value "Alice" is passed to the parameter name.
- Inside the function, the parameter name holds the value "Alice", and the function prints "Hello, Alice!" to the console.

Keyword Parameters

Keyword parameters (also known as keyword arguments) in Python are parameters that are passed to a function by specifying their corresponding parameter names along with the values. Keyword parameters allow you to provide arguments to a function in any order, making the function call more readable and flexible.

Here's the syntax to define and use keyword parameters in Python functions:

1 **Defining keyword parameters:**

```
def function_name(param1=default_value1, param2=default_value2, ...):
    # Function body
    # Code that uses param1, param2, ...
```

- def keyword is used to define a function.
- function_name is the name of the function.

- (param1=default_value1, param2=default_value2, ...) is the list of parameters with their default values (if any). Keyword parameters are optional.
- Each parameter in the list is defined with its default value using the syntax param=default_value.
- Default values are used if no value is provided for the corresponding parameter during the function call.

2 **Using keyword parameters:**

```
result = function_name(param1=value1, param2=value2, ...)
```

- function_name is the name of the function to call.
- param1=value1, param2=value2, etc., are the keyword arguments passed to the function.
- Each keyword argument consists of the parameter name followed by the value assigned to it.
- Keyword arguments can be provided in any order.

Here's an example demonstrating the usage of keyword parameters in Python functions:

```
# Function definition with keyword parameters
def greet(name, message="Hello"):
    # Function body: greets the user with the provided name and message
    print(message, name, "!")

# Using the function with keyword arguments
greet(message="Hi", name="Bob")  # Output: Hi Bob!
```

In this example:

- We define a function greet that takes two parameters: name and message.
- The parameter message has a default value "Hello".
- When the function is called with keyword arguments message="Hi" and name="Bob", these arguments are passed to the corresponding parameters.
- The function uses the provided values to print the greeting message "Hi Bob!" to the console.

Default value

In Python, you can add default values to function parameters by specifying the default values directly in the function definition. Here's the syntax to add default values to function parameters:

```
def function_name(param1=default_value1, param2=default_value2, ...):
    # Function body
    # Code that uses param1, param2, ...
```

In this syntax:

- param1=default_value1, param2=default_value2, etc., are the parameters with their corresponding default values.
- If no value is provided for a parameter during the function call, the default value specified in the function definition is used.

Here's an example demonstrating how to add default values to function parameters:

```
# Function definition with default parameter values
def greet(name, message="Hello"):
    # Function body: greets the user with the provided name and message
    print(message, name, "!")

# Using the function without providing the default parameter value
greet("Alice")  # Output: Hello Alice!

# Using the function with providing a custom parameter value
greet("Bob", "Hi")  # Output: Hi Bob!
```

In this example:

- We define a function greet with two parameters: name and message.
- The parameter message has a default value "Hello".
- When the function is called without providing a value for the message parameter (greet("Alice")), the default value "Hello" is used, and the function prints "Hello Alice!".
- When the function is called with a custom value for the message parameter (greet("Bob", "Hi")), the provided value

"Hi" is used instead of the default value, and the function prints "Hi Bob!".

Return Values

In Python, you can return values from a function using the return statement. The return statement is used to exit the function and specify the value(s) that the function should return to the caller. Here's the syntax to return values from a Python function:

```
def function_name(parameter1, parameter2, ...):
    # Function body
    # Code that performs the task
    return value1, value2, ...
```

In this syntax:

- def keyword is used to define a function.
- function_name is the name of the function.
- (parameter1, parameter2, ...) is the list of parameters that the function accepts. Parameters are optional.
- The return statement is used to exit the function and return one or more values to the caller.
- value1, value2, ... are the values that the function returns. You can return multiple values separated by commas, or a single value.

Here's an example demonstrating how to return values from a Python function:

```
# Function definition with parameters and return statement
```

```
def add_numbers(x, y):
    # Function body: adds two numbers
    result = x + y
    return result

# Using the function and storing the returned value
sum_result = add_numbers(3, 5)
print("Sum:", sum_result)  # Output: Sum: 8
```

In this example:

- We define a function add_numbers that takes two parameters x and y, adds them together, and returns the result using the return statement.
- When the function is called with arguments 3 and 5, the function calculates the sum (3 + 5 = 8) and returns the result 8.
- The returned value is stored in the variable sum_result, and we print it to the console.

2 Python Function Recursion

Introduction

Function recursion in Python refers to the concept of a function calling itself, either directly or indirectly. Recursion is a powerful technique used in programming where a function solves a problem by breaking it down into smaller, similar subproblems and calling itself to solve each subproblem. Recursion continues until it reaches a base case, which is a simple case that can be solved directly without further recursion.

Here's a basic example of a recursive function to calculate the factorial of a number:

```
def factorial(n):
    if n == 0:
        return 1
    else:
```

```
        return n * factorial(n - 1)

# Example usage
result = factorial(5)
print(result)  # Output: 120 (5! = 5 * 4 * 3 * 2 * 1 = 120)
```

In this example:

- The factorial function calculates the factorial of a non-negative integer n.
- The base case is when n is equal to 0, where the factorial is defined as 1.
- For any other value of n, the function recursively calls itself with n - 1 until it reaches the base case.
- The result is computed by multiplying n with the factorial of n - 1.

Recursion is useful in situations where a problem can be divided into smaller, identical subproblems that can be solved more easily. Common examples include:

1 **Tree-based problems:** Problems involving trees, such as traversing a binary tree or finding the height of a tree, are often naturally suited for recursion.

2 **Divide and conquer algorithms:** Problems that can be divided into smaller, similar subproblems, such as sorting algorithms like merge sort and quicksort, can be efficiently solved using recursion.

3 **Dynamic programming:** Recursive solutions are often used in dynamic programming to solve problems by breaking them down into overlapping subproblems.

Recursion can make code more concise and elegant for certain types of problems. However, it can also lead to performance issues and stack overflow errors if not implemented carefully, especially for problems with deep recursion or overlapping subproblems. In such cases, iteration or other optimization techniques may be preferred.

Pros and cons

Recursion and iteration are both techniques used in programming to solve problems, and each has its own set of advantages and disadvantages. Here are some pros and cons of recursion and iteration in Python:

Recursion:

Pros:

1 **Elegant solution:** Recursion often leads to concise and elegant solutions for problems that can be divided into smaller, identical subproblems.

2 **Readability:** Recursive solutions closely mimic the problem's definition, making the code easier to understand and maintain, especially for problems involving tree structures or mathematical sequences.

3 **Divide and conquer:** Recursion is well-suited for divide-and-conquer algorithms, where a problem can be broken down into smaller, similar subproblems that are easier to solve.

Cons:

1 **Performance overhead:** Recursive function calls involve additional overhead, such as function call stack management, which can lead to slower execution compared to iterative solutions, especially for deep recursion.

2 **Stack overflow:** Recursive solutions can lead to stack overflow errors if not implemented carefully, particularly for problems with deep recursion or when the base case is not reached.

3 **Difficulty in debugging:** Recursive functions can be more difficult to debug due to their indirect nature, making it challenging to trace the execution flow.

Iteration:

Pros:

1 **Efficiency:** Iterative solutions often have better performance compared to recursive solutions, especially for problems with large inputs or deep recursion levels, as they typically involve less overhead.

2 **Control over execution:** Iteration provides more explicit control over the execution flow, making it easier to understand and debug, particularly for complex algorithms.

3 **Tail optimization:** Some programming languages, including Python, support tail call optimization, which allows certain tail-recursive functions to be optimized into iterative form, eliminating the risk of stack overflow.

Cons:

1 **Complexity:** Iterative solutions can sometimes be more complex and less intuitive, especially for problems that naturally lend themselves to recursive solutions.

2 **Verbose code:** Iterative solutions may require more lines of code and additional variables to maintain loop state, leading to less concise code compared to recursive solutions for certain problems.

3 **Difficulty with certain problems:** Some problems may be inherently difficult to solve iteratively, especially those involving tree traversal or backtracking, where recursion provides a more natural and intuitive approach.

In summary, both recursion and iteration have their advantages and disadvantages, and the choice between them depends on factors such as problem complexity, performance requirements, and personal preference. While recursion often leads to elegant and concise solutions for certain problems, iteration may offer better performance and explicit control over execution flow in others. It's essential to consider these factors when deciding which approach to use in a given situation.

Sum

You can find the sum of natural numbers using recursion in Python with a function that calls itself until reaching the base case. Here's how you can do it:

```
def sum_of_natural_numbers(n):
    if n <= 1:
        return n
    else:
        return n + sum_of_natural_numbers(n - 1)

# Test the function
n = 5
print("Sum of first", n, "natural numbers is:", sum_of_natural_numbers(n))
```

In this code:

- The base case is when n becomes 1 or less, in which case we return n.
- Otherwise, we return n plus the sum of natural numbers from 1 to n-1, which we obtain by calling the function recursively with n-1.
- The function keeps calling itself with decreasing values of n until it reaches the base case, at which point the recursion stops.

String Reverse

You can reverse a string using recursion in Python by recursively swapping the characters at the beginning and end of the string until the entire string is reversed. Here's how you can implement it:

```
def reverse_string(s):
    if len(s) <= 1:
        return s
    else:
        return reverse_string(s[1:]) + s[0]

# Test the function
input_string = "hello"
```

```
print("Original string:", input_string)
print("Reversed string:", reverse_string(input_string))
```

In this code:

- The base case is when the length of the string is less than or equal to 1. In this case, the string itself is returned.
- Otherwise, the function returns the result of recursively calling reverse_string on the substring starting from the second character (s[1:]) and then appending the first character (s[0]) at the end.
- The function keeps calling itself with substrings until it reaches the base case, at which point the recursion stops and the reversed string is constructed.

Chapter 6 :

Object Oriented

1 Python Modules

Introduction

A Python module is a file containing Python code, which can define functions, classes, and variables. The purpose of a Python module is to organize related code into a reusable and shareable unit. Modules provide a way to structure large Python projects into smaller, manageable components, making it easier to maintain, test, and collaborate on code.

Here are some key purposes and benefits of using Python modules:

1 **Code Organization:** Modules help organize code by grouping related functionality together. This makes it easier to understand and maintain the codebase.

2 **Encapsulation:** Modules provide a way to encapsulate code into separate namespaces. This helps prevent naming conflicts and allows you to use the same names for variables and functions in different modules without causing conflicts.

3 **Reusability:** Modules can be reused in multiple projects or within the same project. Once defined, a module can be imported into other Python scripts or modules, allowing you to reuse code without duplication.

4 **Abstraction:** Modules allow you to abstract away implementation details and expose only the necessary interfaces to users. This promotes a clean and modular design, where each module can be treated as a black box with well-defined inputs and outputs.

5 **Scoping:** Modules define their own scope, which helps avoid polluting the global namespace. Variables and functions defined within a module are accessible only within that module unless explicitly imported or exported.

6 **Namespacing:** Modules provide a way to organize and manage namespaces in Python. Each module has its own namespace, which helps avoid naming conflicts and provides a clean separation of concerns.

Overall, Python modules play a crucial role in structuring and organizing Python code, promoting modularity, reusability, and maintainability. They enable developers to build complex applications by breaking them down into smaller, more manageable components.

Create

To create a Python module, you simply need to create a Python file (.py) containing the code you want to include in the module. Here's a step-by-step guide on how to create a Python module:

1 **Create a Python File:** Create a new file with a .py extension. This file will contain the code for your module. For example, you can name it mymodule.py.

2 **Write Module Code:** Write the Python code for your module in the created file. This can include function definitions, class definitions, variable assignments, and any other Python code you want to include in the module.

3 **Define Module Interface:** Decide which functions, classes, and variables you want to make accessible to users of your module. These will be the public interface of your module.

4 **Save the File:** Save the Python file containing your module code.

5 **Use the Module:** You can now import and use your module in other Python scripts or modules by using the import statement.

Here's an example of a simple Python module named mymodule.py:

```
# mymodule.py

def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b

PI = 3.14159
```

To use this module in another Python script, you would import it like this:

```
# main.py

import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.add(2, 3))       # Output: 5
print(mymodule.PI)              # Output: 3.14159
```

As for the file/folder structure convention for a Python module, it's common practice to organize related modules into packages. A package is a directory that contains one or more Python modules and an __init__.py file. Here's an example of a typical file/folder structure for a Python module:

```
my_package/
│
├── __init__.py
├── module1.py
├── module2.py
└── subpackage/
    │
    ├── __init__.py
    ├── submodule1.py
    └── submodule2.py
```

In this structure:

- my_package is the top-level package directory.
- __init__.py files indicate that the directories are Python packages and can contain module code.
- module1.py and module2.py are Python modules directly within the my_package package.
- subpackage is a subpackage directory within the my_package package.
- submodule1.py and submodule2.py are Python modules within the subpackage subpackage.

This structure allows you to organize your modules into logical groups and namespaces, making it easier to manage and maintain larger codebases.

import Syntax

In Python, the import statement is used to include various Python modules into your code. There are several syntaxes you can use depending on what you want to achieve:

1 **Importing the entire module:**

```
import module_name
```

Example:

```
import math
```

2 **Importing with an alias:**

```
import module_name as alias
```

Example:

```
import numpy as np
```

3 **Importing specific attributes from a module:**

```
from module_name import attribute_name1, attribute_name2, ...
```

Example:

```
from math import sqrt, pi
```

4 **Importing all attributes from a module:**

```
from module_name import *
```

This imports all attributes defined in the module, but it's generally discouraged because it can lead to namespace pollution and make it unclear where certain functions or classes come from.

Example:

```
from math import
*
```

5 **Importing a submodule:**

```
import package_name.module_name
```

Example:

```
import matplotlib.pyplot as plt
```

These are the primary ways to use the import statement in Python. Each has its use case depending on the requirements of your program and your coding style preferences.

import all function

To import all functions and attributes from a module in Python, you can use the * wildcard character with the import statement. However, it's generally discouraged because it can lead to namespace pollution and make it unclear where certain functions or classes come from.

Here's how you can import all functions from a module:

```
from module_name import *
```

Replace module_name with the name of the module you want to import from.

Example:

```
from math import
*
```

This imports all functions and attributes from the math module into the current namespace. However, it's recommended to import only the specific functions or attributes you need from a module to keep your code clean and avoid potential naming conflicts.

Alias

In Python, you can give a module, a function, or a class an alias using the as keyword. This is particularly useful when you want to shorten long module names, clarify the purpose of a function or class, or avoid naming conflicts. Here's how you can use as to give an alias:

1 **Alias for a module:**

```
import module_name as alias
```

Example:

```
import numpy as np
```

2 Alias for a function or a class:

```
from module_name import function_name_or_class_name as alias
```

Example:

```
from math import sqrt as square_root
```

In the above examples, np is an alias for the numpy module, and square_root is an alias for the sqrt function from the math module.

Using aliases can make your code more readable and concise, especially when dealing with long module names or when you need to clarify the purpose of certain functions or classes.

Importing Specific Functions

You can import specific Python functions using the import statement along with the from keyword. Here's the syntax:

```
from module_name import function_name1, function_name2, ...
```

Replace module_name with the name of the module containing the functions you want to import, and list the function names you want to import separated by commas.

Example:

Let's say you want to import the sqrt and cos functions from the math module:

```
from math import sqrt, cos
```

Now you can use these functions directly in your code without prefixing them with the module name:

```
print(sqrt(4))  # Output: 2.0
print(cos(0))   # Output: 1.0
```

This syntax is useful when you only need specific functions from a module and don't want to import the entire module. It can also make your code more readable by clearly indicating which functions are being used.

Importing Classes

To import classes from a module in Python, you can use the import statement along with the from keyword, similar to how you import functions. Here's the syntax:

```
from module_name import ClassName1, ClassName2, ...
```

Replace module_name with the name of the module containing the classes you want to import, and list the class names you want to import separated by commas.

Example:

Let's say you have a module named my_module containing two classes Dog and Cat, and you want to import both classes:

```
from my_module import Dog, Cat
```

Now you can create objects of these classes directly in your code:

```
my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")
```

This syntax is useful when you only need specific classes from a module and don't want to import the entire module. It can also make your code more readable by clearly indicating which classes are being used.

Importing Multiple Classes

To import multiple classes from a Python module, you can use the from keyword followed by the module name, and then list the class names you want to import, separated by commas. Here's the syntax:

```
from module_name import ClassName1, ClassName2, ...
```

Replace module_name with the name of the module containing the classes you want to import, and list the class names you want to import separated by commas.

Example:

Let's say you have a module named my_module containing two classes Dog and Cat, and you want to import both classes:

```
from my_module import Dog, Cat
```

Now you can create objects of these classes directly in your code:

```
my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")
```

This syntax allows you to import multiple classes from a module in a single line, making your code more concise and readable.

Importing an Entire Module

If you want to import an entire module from another Python module, you can use the simple import statement. Here's how:

```
import module_name
```

Replace module_name with the name of the module you want to import.

Example:

Let's say you have a module named my_module and you want to import the entire module:

```
import my_module
```

After importing, you can access anything defined in my_module by prefixing it with my_module. For example:

```
my_module.some_function()
my_instance = my_module.SomeClass()
```

This approach imports the entire module, including all functions, classes, variables, etc. It's the simplest way to import an entire module, but it can lead to namespace pollution if the module contains a lot of definitions.

Importing All Classes

To import all classes from a Python module, you can use the from keyword with the * wildcard character. Here's the syntax:

```
from module_name import *
```

Replace module_name with the name of the module containing the classes you want to import.

Example:

Let's say you have a module named my_module containing several classes (Class1, Class2, etc.), and you want to import all of them:

```
from my_module import *
```

After this import statement, you can directly use any class defined in my_module without prefixing it with my_module. For example:

```
obj1 =
Class1()
obj2 =
Class2()
```

However, be cautious when using this approach because it can lead to namespace pollution and make it unclear where certain classes come from, especially if the module contains a large number of classes. It's generally recommended to import specific classes or use an alias to avoid these issues.

Importing a Module

To import a module into another Python module, you can use the import statement. Here's the syntax:

```
import module_name
```

Replace module_name with the name of the module you want to import.

Example:

Let's say you have two Python modules, module1.py and module2.py, and you want to import module1 into module2:

In module2.py:

```
import module1
```

After importing, you can access anything defined in module1 by prefixing it with module1. For example:

```
module1.some_function()
```

If you want to access specific attributes (functions, classes, variables) from module1, you can do so by directly referencing them after importing:

```
from module1 import some_function, SomeClass

some_function()
obj = SomeClass()
```

This imports only some_function and SomeClass from module1, making them accessible directly without prefixing with module1..

Alternatively, if you want to import all attributes from module1 into module2, you can use the * wildcard character:

```
from module1 import *
```

However, be cautious when using this approach as it can lead to namespace pollution and make it unclear where certain attributes come from. It's generally recommended to import specific attributes or use an alias to avoid these issues.

2 Python class

Introduction

In Python, a class is a blueprint for creating objects (instances). It defines the properties (attributes) and behaviors (methods) that all objects created from it will have. Classes are fundamental to object-oriented programming (OOP), a programming paradigm that models real-world entities as objects with attributes and behaviors.

Here's a breakdown of key concepts related to Python classes:

1 **Attributes:** These are variables that store data associated with the class or its instances.

2 **Methods:** These are functions defined within a class that can perform operations on the class's data.

3 **Instances:** These are individual objects created from a class. Each instance has its own set of attributes and can call the class's methods.

4 **Inheritance:** This is a feature of OOP that allows a class (called a subclass or derived class) to inherit attributes and methods from another class (called a superclass or base class). It promotes code reusability and allows for creating specialized classes based on existing ones.

When do we need Python classes

1 **Encapsulation:** Classes allow you to encapsulate data and functionality together. This means that data and the operations that manipulate it are bundled together within the class, promoting modularity and making the code easier to understand and maintain.

2 **Abstraction:** Classes provide a way to model real-world entities with their attributes and behaviors. They abstract away the details of implementation and allow you to work with higher-level concepts.

3 **Code Reusability:** By defining classes and using inheritance, you can reuse code across different parts of your program. This promotes code organization, reduces redundancy, and makes it easier to manage complex systems.

4 **Polymorphism:** Classes support polymorphism, which allows different objects to be treated as instances of the same class, even if they are of different types. This promotes flexibility and makes it easier to work with heterogeneous collections of objects.

In summary, Python classes are essential for building modular, maintainable, and scalable software. They provide a way to structure code, encapsulate data and behavior, promote code reuse, and model real-world entities in a flexible and understandable manner.

__init__()

In Python, __init__() is a special method (also known as a constructor) that is automatically called when a new instance of a class is created. It is used to initialize the attributes of the newly created object. The __init__() method is optional, but it is commonly used to set up the initial state of an object.

Here's the syntax for defining the __init__() method within a class:

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
        # Initialization code here
```

In the __init__() method:

- self: The first parameter of the __init__() method is always self, which refers to the current instance of the class. It is used to access the attributes and methods of the instance within the class.

- parameter1, parameter2, ...: These are the parameters that you want to pass when creating an instance of the class. You can define any number of parameters here, depending on the initialization requirements of your class.

Here's an example to illustrate the usage of the __init__() method:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating instances of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Accessing attributes of the instances
print(person1.name)  # Output: Alice
print(person1.age)   # Output: 30

print(person2.name)  # Output: Bob
print(person2.age)   # Output: 25
```

In this example, the __init__() method initializes the name and age attributes of each Person object when it is created. When you create a new Person object (person1 and person2), you provide values for name and age, which are then assigned to the corresponding attributes of the object.

Instance

To make an instance from a Python class, you need to follow these steps:

1 **Define the class:** Define the blueprint for the objects you want to create by writing a class definition.

2 **Instantiate the class:** Use the class name followed by parentheses () to create an instance of the class.

Here's a simple example:

```
# Define the class
class MyClass:
    def __init__(self, parameter1, parameter2):
        self.parameter1 = parameter1
        self.parameter2 = parameter2

# Instantiate the class
my_instance = MyClass(value1, value2)
```

Let's break down this example:

- We define a class named MyClass.
- Inside the class, there is an __init__() method that initializes the instance attributes (parameter1 and parameter2) with the values passed as arguments.
- We then create an instance of MyClass by calling MyClass(value1, value2). This invokes the __init__() method with the specified values for parameter1 and parameter2, creating a new instance of the class.
- The newly created instance is assigned to the variable my_instance.

Now, you can use my_instance to access the attributes and methods of the MyClass object:

```
print(my_instance.parameter1)
print(my_instance.parameter2)
```

Replace value1 and value2 with the values you want to initialize the instance with. This is how you make an instance from a Python class.

Attributes

To access an attribute from a Python class, you use dot notation (.) followed by the attribute name. Here's how you do it:

1 First, you need to create an instance of the class.

2 Then, you can use dot notation to access the attributes of that instance.

Here's an example:

```
class MyClass:
    def __init__(self, attribute):
        self.attribute = attribute

# Create an instance of MyClass
my_instance = MyClass("value")
```

```
# Access the attribute using dot notation
print(my_instance.attribute)
```

In this example:

- We define a class named MyClass with an __init__() method that initializes an attribute (self.attribute) with the value passed as an argument.
- We create an instance of MyClass called my_instance and pass the value "value" to the constructor.
- We then use dot notation (my_instance.attribute) to access the value of the attribute attribute of the my_instance object.

This will print "value", which is the value assigned to the attribute attribute of the my_instance object.

Calling Methods

To call methods defined in a Python class, you follow these steps:

1 Create an instance of the class.

2 Use dot notation (.) followed by the method name to call the method on that instance.

Here's a simple example:

```
class MyClass:
    def __init__(self, attribute):
        self.attribute = attribute

    def my_method(self):
        print("Hello from my_method!")

# Create an instance of MyClass
my_instance = MyClass("value")

# Call the method using dot notation
my_instance.my_method()
```

In this example:

- We define a class named MyClass with an __init__() method that initializes an attribute (self.attribute) with the value passed as an argument, and a my_method() method.
- We create an instance of MyClass called my_instance and pass the value "value" to the constructor.
- We then call the my_method() method on the my_instance object using dot notation (my_instance.my_method()).

This will print "Hello from my_method!", indicating that the method has been called successfully on the instance of the class.

3 Python class Inheritance

Introduction

Class inheritance in Python allows a class (subclass) to inherit attributes and methods from another class (superclass). This means that the subclass can access and use the attributes and methods of the superclass without redefining them. Inheritance creates a parent-child relationship between classes, where the subclass inherits the characteristics of the superclass and can also have its own additional attributes and methods.

Syntax for Class Inheritance:

```
class Superclass:
    # Attributes and methods

class Subclass(Superclass):
    # Additional attributes and methods
```

In the above syntax:

- Subclass is the subclass that inherits from Superclass.
- Subclass can access all attributes and methods of Superclass and can also define its own additional attributes and methods.

When and Why Do We Need Inheritance

1 **Code Reusability:** Inheritance allows you to reuse code by defining common attributes and methods in a superclass. Subclasses can then inherit these common characteristics without having to redefine them, reducing code duplication.

2 **Modularity and Extensibility:** Inheritance promotes modularity by organizing code into logical hierarchies. Subclasses can extend the functionality of the superclass by adding new methods or overriding existing ones, thus providing flexibility and extensibility to the codebase.

3 **Abstraction and Encapsulation:** Inheritance helps in creating abstract classes that define a common interface or behavior for a group of related classes. It allows you to encapsulate common functionality in the superclass, making the code more manageable and understandable.

4 **Polymorphism:** Inheritance facilitates polymorphism, which allows objects of different classes to be treated uniformly based on their common superclass. This enables you to write code that operates on objects of the superclass type but can also handle objects of the subclass type, providing flexibility and code reuse.

5 **Specialization:** Subclasses can specialize or customize the behavior of the superclass by adding new functionality or modifying existing functionality. This allows you to tailor classes to specific requirements while maintaining a consistent interface across related classes.

Overall, inheritance is a powerful mechanism in object-oriented programming that promotes code reuse, modularity, and extensibility,

making it easier to manage and maintain complex systems. However, it should be used judiciously to avoid creating overly complex class hierarchies and tight coupling between classes.

Syntax

In Python, creating inheritance between classes involves defining a new class that inherits from an existing class. Here's the syntax to create inheritance in Python:

```
class BaseClass:
    # Attributes and methods of the base class

class DerivedClass(BaseClass):
    # Additional attributes and methods of the derived class
```

In the above syntax:

- BaseClass is the name of the existing class from which you want to inherit.
- DerivedClass is the name of the new class that you're creating, which will inherit from BaseClass.
- DerivedClass is said to be a subclass of BaseClass, and BaseClass is said to be the superclass or parent class of DerivedClass.

By inheriting from BaseClass, DerivedClass gains access to all the attributes and methods defined in BaseClass. Additionally, you can define new attributes and methods specific to DerivedClass within its own class definition.

Here's an example to illustrate inheritance in Python:

```
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):  # Dog inherits from Animal
    def sound(self):
```

```
        print("Woof")

class Cat(Animal):  # Cat inherits from Animal
    def sound(self):
        print("Meow")

# Create instances of subclasses
dog = Dog()
cat = Cat()

# Call methods from the superclass and subclasses
dog.sound()  # Output: Woof
cat.sound()  # Output: Meow
```

In this example, Dog and Cat are subclasses of Animal. They inherit the sound() method from Animal but provide their own implementation of the method. When you call sound() on instances of Dog and Cat, it prints the sound specific to each subclass.

__init__()

To define and use the __init__() method for a child class (subclass) in Python inheritance, you can override the __init__() method of the parent class (superclass). This allows you to customize the initialization process for instances of the child class while still leveraging the initialization logic of the parent class if needed. Here's how you can do it:

```
class ParentClass:
    def __init__(self, parent_attribute):
        self.parent_attribute = parent_attribute
        print("ParentClass initialized with:", self.parent_attribute)
```

```
class ChildClass(ParentClass):
    def __init__(self, parent_attribute, child_attribute):
        # Call the __init__() method of the parent class
        super().__init__(parent_attribute)
        self.child_attribute = child_attribute
        print("ChildClass initialized with:", self.child_attribute)

# Create an instance of the child class
child_obj = ChildClass("Parent Data", "Child Data")
```

In the above example:

- ParentClass defines an __init__() method that initializes an attribute parent_attribute.
- ChildClass is a subclass of ParentClass and defines its own __init__() method.
- Inside ChildClass's __init__() method, super().__init__(parent_attribute) calls the __init__() method of the parent class (ParentClass) to initialize the parent_attribute.
- After initializing the parent attribute, the __init__() method of ChildClass initializes its own attribute child_attribute.
- When you create an instance of ChildClass, both ParentClass's and ChildClass's __init__() methods are called, in the order defined in the inheritance hierarchy.

This approach ensures that the initialization logic defined in the parent class is executed before the initialization logic of the child class, allowing for proper initialization of attributes in both classes.

Overriding Methods

In Python, you can override methods from the parent class in a child class by defining a method with the same name in the child class. When an

instance of the child class calls the method, the overridden method in the child class will be executed instead of the method from the parent class. Here's how you can override methods from the parent class in Python:

```
class ParentClass:
    def some_method(self):
        print("This is a method from ParentClass")

class ChildClass(ParentClass):
    def some_method(self):
        print("This is an overridden method from ChildClass")

# Create an instance of the child class
child_obj = ChildClass()

# Call the overridden method
child_obj.some_method()  # Output: This is an overridden method from ChildClass
```

In the above example:

- ParentClass defines a method called some_method().
- ChildClass is a subclass of ParentClass and defines its own version of the some_method() method.
- When you call some_method() on an instance of ChildClass, it calls the overridden method from ChildClass, not the method from ParentClass.

This demonstrates how you can override methods from the parent class in Python. Overriding methods allows child classes to provide their own implementations of methods inherited from the parent class, providing flexibility and customization in object-oriented programming.

4 Python Abstract Base Classes

Introduction

Abstract Base Classes (ABCs) in Python are classes that cannot be instantiated directly but are meant to be subclassed. They define a set of abstract methods that must be implemented by concrete subclasses. ABCs provide a way to define a common interface or behavior for a group of related classes while enforcing a contract that specifies which methods must be implemented by subclasses.

To use Abstract Base Classes in Python, you need to import the abc module from the standard library, which provides the ABC class and other utilities for defining and working with abstract classes. Here's how you can define and use Abstract Base Classes in Python:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# Attempting to instantiate the Shape class directly will raise an error
# shape = Shape()  # This will raise TypeError: Can't instantiate abstract class Shape with abstract methods area, perimeter

# Instantiate a subclass of Shape
rectangle = Rectangle(5, 4)
print("Area:", rectangle.area())  # Output: 20
```

```
print("Perimeter:", rectangle.perimeter())  # Output: 18
```

In the above example:

- Shape is an abstract base class that defines two abstract methods: area() and perimeter().
- The abstractmethod decorator from the abc module is used to mark these methods as abstract, indicating that they must be implemented by concrete subclasses.
- Rectangle is a concrete subclass of Shape that implements both area() and perimeter() methods.
- You cannot instantiate the Shape class directly because it is abstract and contains abstract methods. Attempting to do so will raise a TypeError.
- You can instantiate Rectangle, which is a concrete subclass of Shape, and use its methods.

Abstract Base Classes provide a way to define a common interface or behavior for a group of related classes, ensuring that subclasses implement the required methods. They are useful for enforcing contracts and promoting code clarity and maintainability in object-oriented programming.

ABC as concept

Abstract classes in Python serve as templates for other classes. They are not meant to be instantiated directly but instead are designed to be subclassed. Abstract classes define a blueprint for how subclasses should be structured

and what methods they should implement. They typically contain one or more abstract methods, which are methods that are declared but not implemented in the abstract class itself. Subclasses must provide concrete implementations for these abstract methods.

Here's how to understand abstract classes as a concept in Python:

1 **Cannot be instantiated directly:** Abstract classes cannot be instantiated directly because they contain one or more abstract methods that are not implemented. Attempting to create an instance of an abstract class will result in an error.

2 **Provide a template for subclasses:** Abstract classes provide a blueprint or template for how subclasses should be structured. They define common methods or attributes that subclasses are expected to implement or use.

3 **Contain one or more abstract methods:** Abstract methods are methods declared in an abstract class but not implemented. They serve as placeholders for methods that must be implemented by subclasses. Subclasses must provide concrete implementations for these abstract methods.

4 **Enforce a contract:** Abstract classes enforce a contract between the abstract class and its subclasses. Subclasses must adhere to the interface defined by the abstract class, implementing all abstract methods to fulfill the contract.

5 **Promote code reuse and maintainability:** Abstract classes promote code reuse by providing a common interface or behavior that can be shared among multiple subclasses. They help in organizing code and promoting maintainability by defining a clear structure for subclasses to follow.

Overall, abstract classes in Python provide a powerful mechanism for defining common interfaces and promoting code clarity, reusability, and maintainability. They encourage good design practices by enforcing

contracts between classes and facilitating the creation of well-structured object-oriented systems.

ABC subclass

Subclassing an Abstract Base Class (ABC) in Python involves creating a new class that inherits from the ABC and provides concrete implementations for its abstract methods. This allows you to define a common interface or behavior specified by the ABC and ensure that subclasses adhere to this interface by implementing the required methods. Here's how you can subclass an ABC and use it in Python:

1 **Defining an ABC:** First, you define an abstract base class by subclassing ABC from the abc module and using the @abstractmethod decorator to mark methods as abstract.

2 **Creating concrete subclasses:** Then, you create concrete subclasses that inherit from the ABC and provide implementations for its abstract methods.

3 **Using the subclasses:** Finally, you can instantiate and use the concrete subclasses, which now adhere to the interface defined by the ABC.

Here's an example illustrating subclassing an ABC in Python:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.radius

# Create instances of concrete subclasses
rectangle = Rectangle(5, 4)
circle = Circle(3)

# Use the concrete subclasses
print("Rectangle Area:", rectangle.area())  # Output: 20
print("Rectangle Perimeter:", rectangle.perimeter())  # Output: 18
print("Circle Area:", circle.area())  # Output: 28.26
print("Circle Circumference:", circle.perimeter())  # Output: 18.84
```

In this example:

- Shape is an abstract base class (ABC) that defines two abstract methods: area() and perimeter().
- Rectangle and Circle are concrete subclasses of Shape that provide implementations for the abstract methods.
- Both subclasses adhere to the interface defined by the Shape ABC, ensuring that they provide area() and perimeter() methods.
- You can instantiate and use instances of the concrete subclasses, which now have well-defined behavior specified by the ABC.

5 Python Operator Overloading

Introduction

Operator overloading in Python refers to the ability to redefine the behavior of built-in operators (+, -, *, /, etc.) for user-defined objects. By overloading operators, you can customize how objects of a class behave when operated with built-in operators, allowing for more natural and intuitive syntax.

To use operator overloading in Python, you need to define special methods, also known as magic methods or dunder methods (due to their double underscore prefix and suffix), that correspond to the operators you want to overload. These special methods are automatically called when the corresponding operator is used with objects of your class.

Here's an example of how to use operator overloading in Python:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Point(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"({self.x}, {self.y})"

# Create two Point objects
p1 = Point(1, 2)
p2 = Point(3, 4)

# Use the overloaded operators
print("Addition:", p1 + p2)  # Output: (4, 6)
print("Subtraction:", p1 - p2)  # Output: (-2, -2)
print("Scalar multiplication:", p1 * 2)  # Output: (2, 4)
```

In this example:

- We define a Point class to represent 2D points with x and y coordinates.
- We overload the +, -, and * operators by defining special methods __add__, __sub__, and __mul__, respectively.
- When we use the +, -, and * operators with Point objects, Python automatically calls the corresponding special methods.
- We also define a __str__ method to customize the string representation of Point objects when using print().

Operator overloading allows you to write more expressive and concise code by providing natural syntax for operations on user-defined objects. It's a powerful feature of Python's object-oriented programming model that enables customization of behavior to suit the needs of your classes.

Addition

To overload the addition operator (+) for numerical operations in Python, you need to define the special method __add__() within your class. This method will be automatically called when the addition operator is used with instances of your class. Here's how to overload the addition operator for numerical operations in Python:

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        # Check if 'other' is an instance of Number
        if isinstance(other, Number):
          # If 'other' is an instance of Number, perform addition
          return Number(self.value + other.value)
        else:
          # If 'other' is not an instance of Number, raise TypeError
          raise TypeError("Unsupported operand type(s) for +: '{}' and
'{}'".format(
                type(self).__name__, type(other).__name__))

    def __str__(self):
        return str(self.value)

# Create instances of Number
```

```
num1 = Number(5)
num2 = Number(10)

# Use the overloaded addition operator
result = num1 + num2
print("Result:", result)  # Output: 15
```

In this example:

- We define a Number class that represents a numerical value.
- We define the __add__() special method, which will be called when the addition operator (+) is used with instances of the Number class.
- Inside the __add__() method, we check if the other operand is an instance of the Number class. If it is, we perform the addition of the values and return a new Number object with the result.
- If the other operand is not an instance of the Number class, we raise a TypeError to indicate that the operation is not supported.
- We define a __str__() method to provide a string representation of Number objects when they are printed.

You can now use the addition operator (+) with instances of the Number class, and Python will automatically call the __add__() method to perform the addition operation.

Subtraction

To overload the subtraction operator (-) for numerical operations in Python, you need to define the special method __sub__() within your class. This method will be automatically called when the subtraction operator is used with instances of your class. Here's how to overload the subtraction operator for numerical operations in Python:

```
class Number:
    def __init__(self, value):
        self.value = value

    def __sub__(self, other):
```

```
        # Check if 'other' is an instance of Number
        if isinstance(other, Number):
            # If 'other' is an instance of Number, perform subtraction
            return Number(self.value - other.value)
        else:
            # If 'other' is not an instance of Number, raise TypeError
            raise TypeError("Unsupported operand type(s) for -: '{}' and
'{}'".format(
                type(self).__name__, type(other).__name__))

    def __str__(self):
        return str(self.value)

# Create instances of Number
num1 = Number(10)
num2 = Number(5)

# Use the overloaded subtraction operator
result = num1 - num2
print("Result:", result)  # Output: 5
```

In this example:

- We define a Number class that represents a numerical value.
- We define the __sub__() special method, which will be called when the subtraction operator (-) is used with instances of the Number class.
- Inside the __sub__() method, we check if the other operand is an instance of the Number class. If it is, we perform the subtraction of the values and return a new Number object with the result.
- If the other operand is not an instance of the Number class, we raise a TypeError to indicate that the operation is not supported.
- We define a __str__() method to provide a string representation of Number objects when they are printed.

You can now use the subtraction operator (-) with instances of the Number class, and Python will automatically call the __sub__() method to perform the subtraction operation.

Multiplication

To overload the multiplication operator (*) for numerical operations in Python, you need to define the special method __mul__() within your class. This method will be automatically called when the multiplication operator is used with instances of your class. Here's how to overload the multiplication operator for numerical operations in Python:

```
class Number:
    def __init__(self, value):
        self.value = value

    def __mul__(self, other):
        # Check if 'other' is an instance of Number
        if isinstance(other, Number):
            # If 'other' is an instance of Number, perform multiplication
            return Number(self.value * other.value)
        else:
            # If 'other' is not an instance of Number, raise TypeError
            raise TypeError("Unsupported operand type(s) for *: '{}' and
'{}'".format(
                type(self).__name__, type(other).__name__))

    def __str__(self):
        return str(self.value)

# Create instances of Number
num1 = Number(5)
num2 = Number(10)

# Use the overloaded multiplication operator
result = num1 * num2
print("Result:", result)  # Output: 50
```

In this example:

- We define a Number class that represents a numerical value.
- We define the __mul__() special method, which will be called when the multiplication operator (*) is used with instances of the Number class.
- Inside the __mul__() method, we check if the other operand is an instance of the Number class. If it is, we perform the

multiplication of the values and return a new Number object with the result.
- If the other operand is not an instance of the Number class, we raise a TypeError to indicate that the operation is not supported.
- We define a __str__() method to provide a string representation of Number objects when they are printed.

You can now use the multiplication operator (*) with instances of the Number class, and Python will automatically call the __mul__() method to perform the multiplication operation.

Chapter 7 :

Advanced

1 Python File

Access Modes

Python file access modes are used to specify the mode in which a file is opened. Each mode determines the operations that can be performed on the file, such as reading, writing, appending, or creating a new file. Here are the commonly used file access modes in Python:

1 **'r':** Open for reading (default). If the file does not exist or cannot be opened, an IOError will be raised.

2 **'w':** Open for writing. If the file already exists, its contents will be overwritten. If the file does not exist, it will be created.

3 **'a':** Open for appending. The file pointer is positioned at the end of the file. New data will be written to the end of the file. If the file does not exist, it will be created.

4 **'r+':** Open for reading and writing. The file pointer is positioned at the beginning of the file. If the file does not exist or cannot be opened, an IOError will be raised.

5 **'w+':** Open for reading and writing. If the file already exists, its contents will be overwritten. If the file does not exist, it will be created.

6 **'a+':** Open for reading and appending. The file pointer is positioned at the end of the file. New data will be written to the end of the file. If the file does not exist, it will be created.

Here's how you can use these file access modes in Python:

```
# Open a file in read mode
with open('file.txt', 'r') as file:
    content = file.read()
    print(content)

# Open a file in write mode
with open('file.txt', 'w') as file:
    file.write('Hello, world!')

# Open a file in append mode
with open('file.txt', 'a') as file:
    file.write('\nAppending new content')

# Open a file in read and write mode
with open('file.txt', 'r+') as file:
    content = file.read()
    file.write('\nAdding new content')

# Open a file in read and write mode (creating a new file if it doesn't exist)
with open('new_file.txt', 'w+') as file:
    file.write('This is a new file')

# Open a file in read and append mode (creating a new file if it doesn't exist)
with open('new_file.txt', 'a+') as file:
    file.write('\nThis is appended content')
```

Remember to always close the file using the with statement or by explicitly calling the close() method after you are done working with it. This ensures that any resources used by the file are properly released.

Handler

In Python, a file handler (also referred to as a file object or file descriptor) is an object that represents an open file. It provides methods and attributes that allow you to interact with the file, such as reading from it, writing to it, or manipulating its contents.

File handlers are typically obtained by calling the open() function, which returns a file handler associated with the specified file. You can then use this file handler to perform various operations on the file.

Here's how to use a file handler in Python:

```
# Open a file in read mode and obtain a file handler
file_handler = open('file.txt', 'r')

# Read the entire contents of the file
content = file_handler.read()
print(content)

# Close the file handler when done
file_handler.close()
```

In the above example:

- We use the open() function to open a file named 'file.txt' in read mode ('r'). This returns a file handler that represents the opened file.
- We then use the read() method of the file handler to read the entire contents of the file and store it in the variable content.
- Finally, we close the file handler by calling its close() method. It's important to close file handlers when they are no longer needed to free up system resources and ensure that any buffered data is flushed to the file.

Additionally, Python supports a context manager (with statement) for file handling, which automatically closes the file handler when you exit the block. This is the recommended way to work with files in Python:

```
```

```
# Open a file using a context manager
with open('file.txt', 'r') as file_handler:
    content = file_handler.read()
    print(content)
```

In this example, the file handler is automatically closed when the code block exits, regardless of whether an exception occurs. Using a context manager helps ensure that file resources are properly managed and avoids the need for explicit calls to close().

Append

To append data to a file using a file handler in Python, you can open the file in append mode ('a') and then use the file handler's write() method to add the desired content to the end of the file. Here's how you can do it:

```
# Open the file in append mode and obtain a file handler
with open('file.txt', 'a') as file_handler:
    # Write data to the file
    file_handler.write('\nNew data to be appended')
```

In this example:

- We open the file 'file.txt' in append mode ('a') using a context manager (with statement), which automatically closes the file handler when done.
- Inside the context manager block, we use the file handler's write() method to append the string 'New data to be appended' to the file. The \n character is used to add a new line before the appended data.

After executing this code, the file 'file.txt' will contain the original contents, followed by the new data appended at the end. It's worth noting that the file will be created if it does not already exist. If you want to append data to an existing file, ensure it's present in the specified location.

Create

To create and write content to a file using a file handler in Python, you can open the file in write mode ('w') and then use the file handler's write() method to add the desired content. Here's how you can do it:

```
```

```
# Open the file in write mode and obtain a file handler
with open('file.txt', 'w') as file_handler:
    # Write data to the file
    file_handler.write('This is some content that we are writing to the file.\n')
    file_handler.write('This is another line of content.\n')
```

In this example:

- We open the file 'file.txt' in write mode ('w') using a context manager (with statement), which automatically closes the file handler when done.
- Inside the context manager block, we use the file handler's write() method to write the desired content to the file. Each call to write() adds the specified string to the file.

After executing this code, the file 'file.txt' will contain the content that we wrote to it. If the file already exists, its contents will be overwritten. If you want to append content to an existing file or preserve its current contents, you should use append mode ('a') instead of write mode ('w').

File size

You can use the os.path.getsize() function from the os module to get the size of a file in bytes. Here's how you can use it:

```
import os

# Get the size of the file
file_size = os.path.getsize('file.txt')

print("File size:", file_size, "bytes")
```

In this example:

- We import the os module.
- We use the os.path.getsize() function, passing the path to the file ('file.txt' in this case) as an argument.
- The function returns the size of the file in bytes, which we store in the variable file_size.
- We print the size of the file to the console.

This method allows you to quickly retrieve the size of a file without needing to open it or read its contents.

Move File

To move a file from one directory to another in Python, you can use the shutil.move() function from the shutil module. Here's how you can use it:

```
import shutil

# Source and destination file paths
source_file = 'source_directory/file.txt'
destination_file = 'destination_directory/file.txt'

# Move the file from the source directory to the destination directory
shutil.move(source_file, destination_file)

print("File moved successfully!")
```

In this example:

- We import the shutil module.
- We define the paths of the source file and the destination file.
- We use the shutil.move() function, passing the source file path and the destination file path as arguments.
- The function moves the file from the source directory to the destination directory.
- Finally, we print a message to indicate that the file has been moved successfully.

Make sure to replace 'source_directory/file.txt' and 'destination_directory/file.txt' with the actual paths of the source and destination directories respectively, as well as the filenames you want to move.

Delete File

To delete a file in Python, you can use the os.remove() function from the os module. Here's how you can use it:

```
import os
```

```
# File path
file_path = 'file_to_delete.txt'

# Check if the file exists before attempting to delete it
if os.path.exists(file_path):
    # Delete the file
    os.remove(file_path)
    print("File deleted successfully!")
else:
    print("File does not exist.")
```

In this example:

- We import the os module.
- We define the path of the file we want to delete.
- We use the os.path.exists() function to check if the file exists before attempting to delete it.
- If the file exists, we use the os.remove() function, passing the file path as an argument, to delete the file.
- If the file does not exist, we print a message indicating that the file does not exist.

Make sure to replace 'file_to_delete.txt' with the actual path of the file you want to delete.

2 Python Text File

Read

To read from a file in Python, you can use the open() function to open the file and obtain a file handler, and then use methods like read(), readline(), or readlines() to read the content from the file. Here's how you can do it:

1 **Reading the entire contents of a file at once using read():**

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
    # Read the entire contents of the file
    content = file_handler.read()
    print(content)
```

2 **Reading one line at a time using readline():**

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
    # Read one line at a time
    line = file_handler.readline()
    while line:
        print(line, end='')  # end='' to avoid adding extra newlines
        line = file_handler.readline()
```

3 **Reading all lines into a list using readlines():**

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
    # Read all lines into a list
    lines = file_handler.readlines()
    for line in lines:
        print(line, end='')  # end='' to avoid adding extra newlines
```

In each example:

- We use the open() function with the file name 'file.txt' and the mode 'r' to open the file in read mode.
- We use a context manager (with statement) to automatically close the file handler after it's been opened.
- We use one of the file handler's methods (read(), readline(), or readlines()) to read the content from the file.
- We print the content to the console.

Choose the appropriate method based on your specific requirements. If you want to process the entire content of the file at once, use read(). If you want to process the file line by line, use readline(). If you want to store all lines in a list, use readlines().

Path

In Python, you can obtain both relative and absolute file paths using various methods. Here are some common approaches:

1 **Using the os.path Module:**

- The os.path.abspath() function returns the absolute path of a file.
- The os.path.relpath() function returns the relative path of a file from a specified directory (or the current working directory if not specified).

```
import os

# Absolute path
absolute_path = os.path.abspath('file.txt')
print("Absolute path:", absolute_path)

# Relative path from the current working directory
relative_path = os.path.relpath('file.txt')
print("Relative path:", relative_path)
```

2 **Using the pathlib Module (Python 3.4 and later):**

- The pathlib.Path.resolve() method returns the absolute path of a file.
- The pathlib.Path.relative_to() method returns the relative path of a file from a specified directory.

```
from pathlib import Path

# Absolute path
absolute_path = Path('file.txt').resolve()
print("Absolute path:", absolute_path)

# Relative path from the current working directory
relative_path = Path('file.txt').relative_to(Path.cwd())
print("Relative path:", relative_path)
```

3 **Using the os.getcwd() Function (to get the current working**

directory):

- The os.getcwd() function returns the current working directory.
- You can combine this with relative file paths to get the absolute paths.

```
import os

# Current working directory
cwd = os.getcwd()

# Relative file path
relative_path = 'file.txt'

# Absolute path
absolute_path = os.path.join(cwd, relative_path)
print("Absolute path:", absolute_path)
```

Choose the method that best fits your requirements and coding style. The pathlib module is recommended for its object-oriented interface and improved readability.

Line

To access lines of a file in Python, you can use various methods provided by file objects, such as readline(), readlines(), or iterating over the file object itself. Here's how you can do it:

1 **Using readline():**

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
    # Read one line at a time
    line = file_handler.readline()
    while line:
        # Process the line
        print(line.strip())  # strip() to remove leading and trailing whitespace
        # Read the next line
        line = file_handler.readline()
```

2 **Using readlines():**

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
```

```
    # Read all lines into a list
    lines = file_handler.readlines()
    # Iterate over the lines
    for line in lines:
        # Process each line
        print(line.strip())  # strip() to remove leading and trailing whitespace
```

3 Iterating over the file object:

```
# Open the file in read mode and obtain a file handler
with open('file.txt', 'r') as file_handler:
    # Iterate over the file object
    for line in file_handler:
        # Process each line
        print(line.strip())  # strip() to remove leading and trailing whitespace
```

In each example:

- We use the open() function with the file name 'file.txt' and the mode 'r' to open the file in read mode.
- We use a context manager (with statement) to automatically close the file handler after it's been opened.
- We use one of the methods (readline(), readlines(), or iterating over the file object) to access lines from the file.
- We process each line as needed, such as stripping leading and trailing whitespace using strip(), and then print it to the console.

Write

To write a line to a file using Python, you can use the write() method of the file object. Here's how you can do it:

```
# Open the file in write mode and obtain a file handler
with open('output.txt', 'w') as file_handler:
    # Write a line to the file
    file_handler.write("This is a line written to the file.\n")
```

In this example:

- We use the open() function with the file name 'output.txt' and the mode 'w' to open the file in write mode. If the file does not exist, it will be created. If the file already exists, its contents will be overwritten.
- We use a context manager (with statement) to automatically close the file handler after it's been opened.
- We use the write() method of the file handler to write the specified line to the file. The '\n' character is used to add a newline at the end of the line.

After executing this code, the file 'output.txt' will contain the line "This is a line written to the file." followed by a newline character.

3 Python Exceptions

Introduction

An exception in Python is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an exceptional condition arises, such as an error or unexpected behavior, Python raises an exception to handle the situation.

Exceptions are used to manage errors and unexpected situations in Python programs. They provide a mechanism for handling errors gracefully and preventing program crashes. Instead of abruptly terminating the program when an error occurs, exceptions allow you to handle errors in a controlled

manner, such as by displaying an error message, logging the error, or taking corrective action.

You should use exceptions in Python to:

1 Handle errors: Exceptions allow you to detect and handle errors that occur during the execution of your program. This helps prevent program crashes and provides a way to gracefully recover from errors.

2 Provide feedback to users: Exceptions allow you to provide informative error messages to users, helping them understand what went wrong and how to resolve the issue.

3 Log errors: Exceptions can be logged to record information about errors that occur during program execution. This can be useful for debugging and troubleshooting.

4 Take corrective action: Exceptions can be caught and handled by executing specific code to address the error condition, such as retrying an operation, using default values, or prompting the user for input.

Here's an example of using exceptions to handle a division by zero error:

```
try:
    result = 10 / 0  # Attempting division by zero
except ZeroDivisionError:
    print("Error: Division by zero")
```

In this example:

- The code inside the try block attempts to divide 10 by zero, which would result in a ZeroDivisionError.
- The except block catches the ZeroDivisionError exception and prints an error message indicating that division by zero occurred.
- By handling the exception, the program continues to execute normally without crashing.

try-except

In Python, you use the try and except blocks to handle exceptions. The basic syntax is as follows:

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType:
    # Code to handle the exception
    # ...
```

Here's a breakdown of the syntax:

- The try block contains the code that may raise an exception. It's the block where you expect the potential error to occur.
- The except block specifies the type of exception that you want to catch and handle. If an exception of the specified type (or a subclass of it) occurs within the try block, the corresponding except block is executed.
- You can have multiple except blocks to handle different types of exceptions, or a single except block to catch all exceptions. If no specific exception type is specified, it will catch all exceptions (not recommended unless you have a good reason).

Here's an example demonstrating the use of try-except blocks:

```
try:
    # Code that may raise an exception
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print("Result:", result)
except ValueError:
    # Handle ValueError (e.g., invalid input)
    print("Please enter valid integers.")
except ZeroDivisionError:
    # Handle ZeroDivisionError (e.g., division by zero)
    print("Error: Division by zero is not allowed.")
except Exception as e:
    # Handle any other type of exception
```

```
    print("An error occurred:", e)
```

In this example:

- The try block attempts to perform division operation between two numbers entered by the user.
- If a ValueError occurs (e.g., if the user inputs non-integer values), the corresponding except ValueError block handles it.
- If a ZeroDivisionError occurs (e.g., if the user inputs 0 as the second number), the corresponding except ZeroDivisionError block handles it.
- If any other type of exception occurs, it will be caught by the except Exception block, and the error message will be printed.
- If no exception occurs within the try block, the except block(s) will be skipped, and the program will continue execution after the try-except statement.

else

In Python, you can use a try...else block to handle exceptions in a way that distinguishes between the code that may raise an exception and the code that should run if no exception occurs. The else block is executed only if no exception is raised in the try block. Here's the syntax:

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType:
    # Code to handle the exception
    # ...
else:
    # Code to execute if no exception occurs
    # ...
```

Here's a breakdown of the syntax:

- The try block contains the code that may raise an exception. This is the block where you expect the potential error to occur.

- The except block specifies the type of exception that you want to catch and handle. If an exception of the specified type (or a subclass of it) occurs within the try block, the corresponding except block is executed.

- The else block contains code that should run if no exception occurs in the try block. It is optional and is executed only if no exception is raised.

Here's an example demonstrating the use of try...else blocks:

```
try:
    # Code that may raise an exception
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ValueError:
    # Handle ValueError (e.g., invalid input)
    print("Please enter valid integers.")
except ZeroDivisionError:
    # Handle ZeroDivisionError (e.g., division by zero)
    print("Error: Division by zero is not allowed.")
else:
    # Code to execute if no exception occurs
    print("Result:", result)
```

In this example:

- The try block attempts to perform a division operation between two numbers entered by the user.
- If a ValueError occurs (e.g., if the user inputs non-integer values), the corresponding except ValueError block handles it.
- If a ZeroDivisionError occurs (e.g., if the user inputs 0 as the second number), the corresponding except ZeroDivisionError block handles it.
- If no exception occurs within the try block, the else block is executed, and the result is printed.

finally

In Python, the finally clause is used in conjunction with the try statement to define a block of code that is always executed, regardless of whether an exception occurs or not. The finally block is commonly used to perform cleanup actions, such as closing files or releasing resources, ensuring that these actions are executed even if an exception is raised. Here's the syntax:

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType:
    # Code to handle the exception
    # ...
finally:
    # Code that is always executed
    # ...
```

Here's a breakdown of the syntax:

- The try block contains the code that may raise an exception. This is the block where you expect the potential error to occur.
- The except block specifies the type of exception that you want to catch and handle. If an exception of the specified type (or a subclass of it) occurs within the try block, the corresponding except block is executed.
- The finally block contains code that is always executed, regardless of whether an exception occurs in the try block or not.

Here's an example demonstrating the use of the finally clause:

```
try:
    # Open a file
    file_handler = open('file.txt', 'r')
    # Read data from the file
    data = file_handler.read()
    print("Data from file:", data)
except FileNotFoundError:
    print("File not found.")
finally:
    # Close the file (cleanup action)
```

```
    file_handler.close()
    print("File closed.")
```

In this example:

- The try block attempts to open a file 'file.txt' for reading and read its contents.
- If a FileNotFoundError occurs (e.g., if the file does not exist), the corresponding except FileNotFoundError block handles it.
- The finally block ensures that the file is closed using the close() method, even if an exception occurs or not. This cleanup action is executed regardless of whether an exception is raised, ensuring that the file is properly closed and resources are released.

4 Python Testing

Introduction

Unit testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure they behave as expected. In Python, unit testing is commonly performed using the built-in unittest module or third-party libraries like pytest.

Here's an overview of unit testing in Python:

1 **When to use it:** Unit testing is typically performed during the development phase of a software project. It is used to validate the behavior of individual units or functions in the codebase. Unit tests help ensure that

each unit of code works correctly in isolation before integrating them into the larger system. Unit testing is an essential part of the Test-Driven Development (TDD) process, where tests are written before the actual code implementation.

2 **Why it's needed:**

- **Identify bugs early:** Unit tests can catch bugs early in the development process, making them easier and cheaper to fix.
- **Ensure code quality:** Unit tests help maintain code quality by providing a safety net for refactoring and code changes. They ensure that existing functionality remains intact as the codebase evolves.
- **Facilitate collaboration:** Unit tests serve as documentation for how a piece of code should behave. They make it easier for developers to understand and collaborate on the codebase.
- **Promote confidence:** Having a comprehensive suite of unit tests gives developers confidence that their code works as intended. It allows them to make changes with the assurance that existing functionality won't be inadvertently broken.

3 **How to write unit tests:** In Python, unit tests are written using the unittest framework or other testing libraries like pytest. You define test cases as subclasses of unittest.TestCase and write test methods that verify the behavior of individual functions or classes. Test methods typically use assertions to check expected outcomes against actual results.

Here's a simple example of a unit test using unittest:

```
import unittest

def add(x, y):
    return x + y

class TestAddFunction(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)
```

```
    def test_add_negative_numbers(self):
        self.assertEqual(add(-2, -3), -5)

if __name__ == '__main__':
    unittest.main()
```

In this example, we define a simple add function and write unit tests to verify its behavior for different input scenarios.

Overall, unit testing is an essential practice in software development, including Python, as it helps ensure code quality, maintainability, and reliability of the software product.

pytest install

You can install pytest using pip, the package installer for Python. Here's the command to install pytest:

```
pip install pytest
```

You can run this command in your terminal or command prompt to install pytest. Make sure you have pip installed and configured properly in your Python environment.

Once pytest is installed, you can use it to write and run tests for your Python code.

pytest test cases

To create test cases using pytest in Python, you typically organize your test code in separate Python files and use functions prefixed with test_ to define individual test cases. Here's a step-by-step guide on how to create test cases using pytest:

1 **Install pytest:** If you haven't already installed pytest, you can do so using pip:

```
pip install pytest
```

2 **Create a Python file for your test code:** Create a new Python file (e.g., test_example.py) where you'll write your test cases. This file should contain functions that define your test cases.

3 **Write test functions:** Write test functions using the test_ prefix to indicate that they are test cases. Within these functions, use assertions to verify the expected behavior of the code being tested.

Here's an example of a test file (test_example.py) with some test cases:

```
# test_example.py

def add(x, y):
    return x + y

def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_negative_numbers():
    assert add(-2, -3) == -5
```

In this example, we have two test functions (test_add_positive_numbers and test_add_negative_numbers) that test the add function for different scenarios.

4 **Run pytest:** To run your tests, navigate to the directory containing your test file(s) in your terminal or command prompt, and run the pytest command:

```
pytest
```

pytest will automatically discover and run any test functions defined in files with names that start with test_. It will display the results of the tests, including any failures or errors.

You can also specify the name of the test file(s) or directories containing test files to run specific tests:

```
pytest test_example.py
```

This command runs only the tests defined in test_example.py.