Python Keywords and Identifiers

Keywords – Keywords are reserved words in Python that the Python interpreter uses to recognise the program's structure. The keyword can't be used as a variable name, function name, or identifier. Except for True and False, all keywords in Python are written in lower case.

Example of Keywords –

False, class, finally, is, return, None, continue, for lambda, try, True, def, from, nonlocal, while, and, del, global, not, with, as, elif, if, or, yield, assert, else, import, pass, break, except, in, raise etc.

Identifiers – An identifier is a name given to a variable, function, class, module, or other object. The identification is made up of a series of digits and underscores. The identification should begin with a letter or an Underscore and then be followed by a digit. A-Z or a-z, an UnderScore (_), and a numeral are the characters (0-9). Special characters (#, @, $, percent,!) should not be used in identifiers.

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _.
2. An identifier cannot start with a digit
3. Keywords cannot be used as identifiers
4. We cannot use special symbols like !, @, #, $, % etc. in our identifier
5. Identifier can be of any length
6. Python is a case-sensitive language.

Example of Identifier

var1
_var1
_1_var
var_1

Variables

A variable is a memory location where you store a value in a programming language. In Python, a variable is formed when a value is assigned to it. Declaring a variable in Python does not require any further commands.

Constants

A constant is a kind of variable that has a fixed value. Constants are like containers that carry information that cannot be modified later.

Declaring and assigning value to a constant

NAME = "Rajesh Kumar"
AGE = 20

Datatype

In Python, each value has a datatype. Data types are basically classes, and variables are instances (objects) of these classes, because everything in Python programming is an object.

Python has a number of different data types. The following are some of the important datatypes.

1. Numbers
2. Sequences
3. Sets
4. Maps

a. Number Datatype

Numerical Values are stored in the Number data type. There are four categories of number datatype –

1. Int – Int datatype is used to store the whole number values. Example : x=500
2. Float – Float datatype is used to store decimal number values. Example : x=50.5
3. Complex – Complex numbers are used to store imaginary values. Imaginary values are denoted with 'j' at the end of the number. Example : x=10 + 4j
4. Boolean – Boolean is used to check whether the condition is True or False. Example : x = 15 > 6 type(x)

b. Sequence Datatype

A sequence is a collection of elements that are ordered and indexed by positive integers. It's made up of both mutable and immutable data types. In Python, there are three types of sequence data types:

1. String – Unicode character values are represented by strings in Python. Because Python does not have a character data type, a single character is also treated as a string. Single quotes (' ') or double quotes (" ") are used to enclose strings.
2. List – A list is a sequence of any form of value. The term "element" or "item" refers to a group of values. These elements are indexed in the same way as an array is. List is enclosed in square brackets. Example : dob = [19,"January",1995]

1. Tuples – A tuple is an immutable or unchanging collection. It is arranged in a logical manner, and the values can be accessed by utilizing the index values. A tuple can also have duplicate values. Tuples are enclosed in (). Example : newtuple = (15,20,20,40,60,70)

c. Sets Datatype

A set is a collection of unordered data and does not have any indexes. In Python, we use curly brackets to declare a set. Set does not have any duplicate values. To declare a set in python we use the curly brackets.

Example : newset = {10, 20, 30}

d. Mapping

This is an unordered data type. Mappings include dictionaries.

Dictionaries

In Python, Dictionaries are used generally when we have a huge amount of data. A dictionary is just like any other collection array. A dictionary is a list of strings or numbers that are not in any particular sequence and can be changed. The keys are used to access objects in a dictionary. Curly brackets are used to declare a dictionary. Example : d = {1:'Ajay','key':2}

Operators

Operators are symbolic representations of computation. They are used with operands, which can be either values or variables. On different data types, the same operators can act differently. When operators are used on operands, they generate an expression.

Operators are categorized as –

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators

Mathematical operations such as addition, subtraction, multiplication, and division are performed using arithmetic operators.

Operator	Meaning	Expression	Result
+	Addition	20 + 20	40
–	Subtraction	30 – 10	20
*	Multiplication	10 * 100	1000
/	Division	30 / 10	20
//	Integer Division	25 // 10	2
%	Remainder	25 % 10	5
**	Raised to power	3 ** 2	9

Assignment Operator

When assigning values to variables, assignment operators are used.

Operator	Expression	Equivalent to
=	x=10	x = 10
+=	x += 10	x = x + 10
-=	x -= 10	x = x – 10
*=	x *= 10	x = x * 10
/=	x /= 10	x = x / 10

Comparison Operator

The values are compared using comparison operators or relational operators. Depending on the criteria, it returns True or False.

Operator	Meaning	Expression	Result
>	Greater Than	20 > 10	True
		20 < 50	False
<	Less Than	20 < 10	False
		10 < 40	True
==	Equal To	5 == 5	True
		5 == 6	False
!=	Not Equal to	67 != 45	True
		35 != 35	False

Logical Operator

Logical operators are used to combine the two or more then two conditional statements –

Operator	Meaning	Expression	Result
And	And Operator	True and True	True
		True and False	False
Or	Or Operator	True or False	True
		False or False	False
Not	Not Operator	Not False	True
		Not True	False

Type Conversion

Type conversion is the process of converting the value of one data type (integer, text, float, etc.) to another data type. There are two types of type conversion in Python.

1. Implicit Type Conversion
2. Explicit Type Conversion

<u>Implicit Type Conversion</u>

Python automatically changes one data type to another via implicit type conversion. There is no need for users to participate in this process.

Example :

```
x = 5
y=2.5
z = x / z
```

In the above example, x is containing integer value, y is containing float value and in the variable z will automatically contain float value after execution.

<u>Explicit Type Conversion</u>

Users transform the data type of an object to the required data type using Explicit Type Conversion. To do explicit type conversion, we employ predefined functions such as int(), float(), str(), and so on. Because the user casts (changes) the data type of the objects, this form of conversion is also known as typecasting.

Example : Birth_day = str(Birth_day)

Python Input and Output

Python Output Using print() function

To output data to the standard output device, we use the print() method (screen).

Example :

a = “Hello World!”

print(a)

Python User input

In python, input() function is used to take input from the users.

This function allows the user to take input from the keyboard as a string.

a = input(“Enter name”) #if a is “sagar”, the user entered sagar

Flow of Control

There are three control flow statements in Python – if, for and while.

Decision Making Statement

In programming languages, decision-making statements determine the program’s execution flow.

Python has the following decision-making statements:

1. if statement
2. if..else statements
3. if-elif ladder

If Statement

The if statement is used to test a condition: if the condition is true, a set of statements is executed (called the if-block).

Flow of Control

Syntax -

```
If test expression:
    statement(s)
```

If...else statement

The if/else statement is a control flow statement that allows you to run a block of code only if a set of conditions are satisfied.

Syntax -

```
if test expression:
    Body of if
else:
    Body of else
```

if-elif ladder

Elif stands for “else if.” It enables us to check for several expressions at the same time. If the if condition is False, the next elif block’s condition is checked, and so on. The body of else is executed if all of the conditions are False.

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else: Body of else
```

Nested if statements

An if...elif...else sentence can be nestled inside another if...elif...else statement. In computer programming, this is referred to as nesting.

For Loop

The for statement allows you to specify how many times a statement or compound statement should be repeated. A for statement’s body is executed one or more times until an optional condition is met.

Syntax -

for val in sequence:
    Body of for

While Statement

The while statement allows you to repeatedly execute a block of statements as long as a condition is true. A while statement is an example of what is called a looping statement. A while statement can have an optional else clause.

Syntax -

while test_expression:
    Body of while

Data Types

Basic Data Types

1. **Integers (int)**:
   - **Definition**: Whole numbers without a decimal point.
   - **Example**: 5, 100, -42
   - **Usage**: Used for counting, indexing, and mathematical operations.
   - **Range**: Integers can be positive, negative, or zero. Python supports very large integers, limited by available memory.
2. **Floating-Point Numbers (float)**:
   - **Definition**: Numbers that contain a decimal point.
   - **Example**: 3.14, 0.99, -7.5
   - **Usage**: Used for precise measurements, calculations involving fractions, and scientific calculations.
   - **Precision**: Floating-point numbers are approximate and have a limited precision based on the number of bits used to store them. They are subject to rounding errors.
3. **Strings (str)**:
   - **Definition**: Sequences of characters enclosed in single (') or double quotes (").
   - **Example**: "Hello", 'World', "123"
   - **Usage**: Used for text processing, displaying messages, and handling user input.
4. **Booleans (bool)**:
   - **Definition**: Represents truth values, either True or False.
   - **Example**: True, False
   - **Usage**: Used for conditional statements and logical operations.
5. **Complex Numbers (complex)**

- **Definition**: Numbers that have both a real and an imaginary part. The imaginary part is indicated by the letter j or J.
- **Format**: A complex number is written as real_part + imaginary_part * j.
- **Example**: 3 + 4j, -2 - 5j

Examples

Integer:

```
number = 10
print(type(number))  # Output: <class 'int'>
```

2. **Floating-Point**:

```
pi = 3.14159
print(type(pi))  # Output: <class 'float'>
```

3. **Complex**:

```
z = 2 + 3j
print(type(z))  # Output: <class 'complex'>
```

Compound Data Types

1. **Lists (list)**:
- **Definition**: Ordered, mutable collections of items enclosed in square brackets ([]). Items can be of different types.
- **Example**: [1, 2, 3, 4], ['apple', 'banana', 'cherry']
- **Usage**: Used to store multiple items in a single variable and to perform operations on those items.
2. **Tuples (tuple)**:
- **Definition**: Ordered, immutable collections of items enclosed in parentheses (()). Items can be of different types.
- **Example**: (1, 2, 3, 4), ('red', 'green', 'blue')
- **Usage**: Used to store multiple items in a fixed order where the data shouldn't change.
3. **Dictionaries (dict)**:
- **Definition**: Unordered collections of key-value pairs enclosed in curly braces ({}). Keys are unique, and values can be of any type.
- **Example**: {'name': 'Alice', 'age': 25, 'city': 'New York'}
- **Usage**: Used to store and retrieve data efficiently using keys.
4. **Sets (set)**:
- **Definition**: Unordered collections of unique items enclosed in curly braces ({}).
- **Example**: {1, 2, 3}, {'apple', 'banana'}
- **Usage**: Used to store unique items and perform mathematical set operations like union, intersection, and difference.

Special Data Types

1. **NoneType (None)**:
- **Definition**: Represents the absence of a value or a null value.
- **Example**: None
- **Usage**: Used to signify that a variable has no value or to represent missing or undefined data.

In Python, None is a special constant that represents the absence of a value or a null value. It's a unique data type, NoneType, and is used in various scenarios to indicate that something is undefined or missing.

Mapping(dictionary)

In Python, a dictionary is a built-in data structure that allows you to store and manage data in key-value pairs. It's a type of mapping where each key is associated with a value, making it easy to look up data based on a unique identifier.

Dictionaries (dict):
- **Definition**: Collections of key-value pairs where the data can be modified after creation.
- **Operations**: You can add, update, or remove key-value pairs.
- **Example**:

```
student = {"name": "Alice", "age": 18}
student["age"] = 19  # Updates the value
student["grade"] = "A"  # Adds a new key-value pair
```

3. **Sets (set)**:
- **Definition**: Unordered collections of unique elements that can be modified.
- **Operations**: You can add or remove elements from a set.
- **Example**:

```
numbers = {1, 2, 3}
numbers.add(4)  # Adds a new element
numbers.remove(2)  # Removes an element
```

Immutable Data Types

Immutable data types are those whose values cannot be changed after they are created. Any operation that seems to modify the object actually creates a new object.

Tuples (tuple):

- **Definition**: Ordered collections of elements, similar to lists but immutable.
- **Operations**: Any modification creates a new tuple object.
- **Example**:
  coordinates = (10, 20)
  new_coordinates = coordinates + (30,) **# Creates a new tuple**

Expressions and Statements

- **Expression**: An expression is a combination of values, variables, operators, and functions that evaluates to a single value. Expressions can be as simple as 5 + 3 or as complex as ((2 * 3) + (4 / 2)). Expressions always return a result.
- **Example**:
  result = 5 * (2 + 3) **# 5 * 5 = 25, so the expression evaluates to 25**
- **Statement**: A statement is a complete unit of execution that performs an action. Statements include assignments, loops, conditionals, and function calls. Statements do not return a value but execute an operation.
- **Example**:
  x = 10 **# This is an assignment statement**
  print(x) **# This is a print statement**

Precedence of Operators

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence. For instance, multiplication (*) has higher precedence than addition (+), so in the expression 2 + 3 * 4, the multiplication is performed first, giving 2 + 12, which results in 14.

- **Example**:
  result = 10 + 3 * 2 **# The multiplication is done first, so 10 + (3 * 2) = 16**

Evaluation of an Expression

When an expression is evaluated, Python performs operations based on operator precedence and associativity rules. Parentheses can be used to override default precedence and force specific order of operations.

- **Example**:
  value = (8 + 2) * 5 **# Parentheses ensure that 8 + 2 is evaluated first, so (10 * 5) = 50**

Type Conversion

Type conversion allows you to change the data type of a value. This can be done implicitly by Python or explicitly by the programmer.

- **Explicit Conversion**: Done using functions like int(), float(), and str() to convert between types.
- **Example**:
  num_str = "123"
  num_int = int(num_str) **# Converts the string "123" to the integer 123**
- **Implicit Conversion**: Python automatically converts types when necessary, such as converting integers to floats during arithmetic operations involving both types.
- **Example**:
  result = 10 + 2.5 **# The integer 10 is implicitly converted to a float, so the result is 12.5**

Understanding these concepts helps you write more effective and error-free code by ensuring expressions are evaluated correctly and data types are managed properly.

Flow of Control

The flow of control refers to the order in which the individual statements, instructions, or function calls are executed or evaluated in a programming language. In Python, this flow is managed using different constructs like sequences, conditions, and loops.

Use of Indentation

Python uses indentation (whitespace) to define blocks of code. Unlike some other programming languages that use braces or keywords, Python's indentation is crucial for defining the structure and flow of the program. Proper indentation ensures that code blocks (such as those following conditional statements or loops) are correctly associated with their control statements.

- **Example**:

```
if True:
    print("This is inside the if block")  # Indented block
print("This is outside the if block")  # Not indented
```

Conditional Statements: if, if-else, if-elif-else

Conditional statements in Python are used to execute specific blocks of code based on certain conditions. Let's explore the different types of conditional statements, along with flowcharts and examples of simple programs.

Conditional Statements

1. **if Statement** The if statement checks a condition and executes the associated block of code if the condition is true.

o **Example**:

```
temperature = 30
if temperature > 25:
    print("It's a hot day!")
```

2. **if-else Statement** The if-else statement provides an alternative block of code to execute if the condition is false.

o **Example**:

```
temperature = 20
if temperature > 25:
    print("It's a hot day!")
else:
    print("It's a cool day!")
```

3. **if-elif-else Statement** The if-elif-else statement allows multiple conditions to be checked in sequence. The first true condition's block is executed, and if none are true, the else block is executed.

o **Example**:

```
temperature = 10
if temperature > 30:
    print("It's a very hot day!")
elif temperature > 20:
    print("It's a warm day!")
elif temperature > 10:
    print("It's a cool day!")
else:
    print("It's a cold day!")
```

Iterative Statement:

Iterative statements allow you to execute a block of code repeatedly based on certain conditions. In Python, the primary iterative statements are for loops and while loops. Here's how they work:

for Loop

The for loop iterates over a sequence (like a list, tuple, or string) or a range of numbers. It's useful when you know in advance how many times you want to repeat a block of code.

- **Example**:

```
for i in range(5):
    print(i)  # Prints numbers from 0 to 4
```

range() Function

The range() function generates a sequence of numbers and is commonly used with for loops. It can take up to three arguments: start, stop, and step.

- **Example**:

```
for i in range(2, 10, 2):
    print(i)  # Prints 2, 4, 6, 8
```

while Loop

The while loop executes as long as its condition remains true. It's useful when you don't know beforehand how many times you'll need to repeat the code.

- **Example**:

```
count = 0
while count < 5:
    print(count)
    count += 1  # Increment count
```

break and continue Statements

- **break**: Exits the loop immediately, regardless of the loop condition.
  - **Example**:

```
for i in range(10):
    if i == 5:
        break  # Exits the loop when i is 5
    print(i)
```

- **continue**: Skips the current iteration and continues with the next iteration of the loop.
  - **Example**:

```
for i in range(10):
    if i % 2 == 0:
        continue  # Skips even numbers
    print(i)  # Prints only odd numbers
```

Nested Loops

Nested loops involve placing one loop inside another. They are useful for working with multi-dimensional data or generating patterns.

- **Example**:

```
for i in range(3):
    for j in range(3):
        print(f"({i}, {j})", end=" ")
    print()  # New line after inner loop
```

Strings

Strings are sequences of characters used to represent text in Python. They are one of the most commonly used data types and are essential for handling textual data. Here's a brief overview of strings and some of their key operations.

Introduction to Strings

A string is a collection of characters enclosed in single quotes ('), double quotes ("), or triple quotes (''' or """). Strings are immutable, meaning once created, their content cannot be changed.

- **Example**:

```
message = "Hello, World!"
```

String Operations

1. **Concatenation:** Concatenation combines two or more strings into one. This is done using the + operator.
   - **Example**:

first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name) **# Output: John Doe**

2. **Repetition:** Repetition allows you to repeat a string a certain number of times using the * operator.
- **Example**:
  echo = "Hello! " * 3
  print(echo) **# Output: Hello! Hello! Hello!**

4. **Slicing:** Slicing extracts a part of the string. You use indexing to specify the start and end positions. The syntax is string[start:end].
- **Example**:
  phrase = "Hello, World!"
  slice1 = phrase[0:5] **# Extracts 'Hello'**
  slice2 = phrase[7:] **# Extracts 'World!'**
  print(slice1) **# Output: Hello**
  print(slice2) **# Output: World!**
- vowels is a string containing all vowel characters.
- For each character in text, check if it is in the vowels string.
- Increment count if a vowel is found.

Built-in String Functions/Methods in Python

Python strings come with a variety of built-in methods that make text manipulation easy and efficient. Here's a rundown of some commonly used string methods:

- **len()**
- **Usage**: Returns the number of characters in the string.
- **Example**:
  text = "Hello"
  print(len(text)) **# Output: 5**
- **capitalize()**
- **Usage**: Capitalizes the first character of the string and makes all other characters lowercase.
- **Example**:
  text = "hello world"
  print(text.capitalize()) **# Output: Hello world**
- **title()**
- **Usage**: Capitalizes the first letter of each word in the string.
- **Example**:
  text = "hello world"
  print(text.title()) **# Output: Hello World**
- **lower()**
- **Usage**: Converts all characters in the string to lowercase.
- **Example**:
  text = "HELLO"
  print(text.lower()) **# Output: hello**
- **upper()**
- **Usage**: Converts all characters in the string to uppercase.
- **Example**:
  text = "hello"
  print(text.upper()) **# Output: HELLO**
- **count()**
- **Usage**: Counts the occurrences of a substring within the string.

- **Example**:
  text = "hello hello"
  print(text.count("hello")) **# Output: 2**
- **find()**
- **Usage**: Returns the lowest index where the substring is found, or -1 if not found.
- **Example**:
  text = "hello world"
  print(text.find("world")) **# Output: 6**
- **index()**
- **Usage**: Returns the lowest index where the substring is found, raises ValueError if not found.
- **Example**:
  text = "hello world"
  print(text.index("world")) **# Output: 6**
- **endswith()**
- **Usage**: Checks if the string ends with the specified substring.
- **Example**:
  text = "hello world"
  print(text.endswith("world")) **# Output: True**
- **startswith()**
- **Usage**: Checks if the string starts with the specified substring.
- **Example**:
  text = "hello world"
  print(text.startswith("hello")) **# Output: True**
- **isalnum()**
- **Usage**: Returns True if all characters in the string are alphanumeric (letters and numbers).
- **Example**:
  text = "hello123"
  print(text.isalnum()) **# Output: True**
- **isalpha()**
- **Usage**: Returns True if all characters in the string are alphabetic.
- **Example**:
  text = "hello"
  print(text.isalpha()) **# Output: True**
- **isdigit()**
- **Usage**: Returns True if all characters in the string are digits.
- **Example**:
  text = "12345"
  print(text.isdigit()) **# Output: True**
- **islower()**
- **Usage**: Returns True if all characters in the string are lowercase.
- **Example**:
  text = "hello"
  print(text.islower()) **# Output: True**
- **isupper()**
- **Usage**: Returns True if all characters in the string are uppercase.
- **Example**:
  text = "HELLO"
  print(text.isupper()) **# Output: True**
- **isspace()**
- **Usage**: Returns True if all characters in the string are whitespace.

- **Example**:
  text = " "
  print(text.isspace()) **# Output: True**
- **lstrip()**
- **Usage**: Removes leading whitespace or specified characters.
- **Example**:
  text = " hello"
  print(text.lstrip()) **# Output: hello**
- **rstrip()**
- **Usage**: Removes trailing whitespace or specified characters.
- **Example**:
  text = "hello "
  print(text.rstrip()) **# Output: hello**
- **strip()**
- **Usage**: Removes leading and trailing whitespace or specified characters.
- **Example**:
  text = " hello "
  print(text.strip()) **# Output: hello**
- **replace()**
- **Usage**: Replaces occurrences of a substring with another substring.
- **Example**:
  text = "hello world"
  print(text.replace("world", "Python")) **# Output: hello Python**
- **join()**
- **Usage**: Joins elements of an iterable (e.g., list) into a single string with a specified separator.
- **Example**:
  words = ["Hello", "world"]
  print(" ".join(words)) **# Output: Hello world**
- **partition()**
- **Usage**: Splits the string into a 3-tuple containing the part before the separator, the separator itself, and the part after.
- **Example**:
  text = "hello world"
  print(text.partition(" ")) **# Output: ('hello', ' ', 'world')**
- **split()**
- **Usage**: Splits the string into a list of substrings based on a separator.
- **Example**:
  text = "hello world"
  print(text.split()) **# Output: ['hello', 'world']**

Functions in Python

A function can be defined as the organized block of reusable code which can be called whenever required. In other words, Python allows us to divide a large program into the basic building blocks known as function.

Python provide us various inbuilt functions like range() or print(). Although, the user can able to create functions which can be called user-defined functions.

Creating Function

In python, a function can be created by using def keyword.

Syntax:

```
def my_function():
   Statements
   return statement
```

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

Example:

```
def sample():                    #function definition
        print ("Hello world")

sample()            #function calling
```

Modules in Python

In python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module.

Modules in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to import the specific module.

Creating Module

Example: demo.py

```
# Python Module example
def sum(a,b):
        return a+b
def sub(a,b):
        return a-b
def mul(a,b):
        return a*b
def div(a,b):
        return a/b
```

In the above example we have defined 4 functions sum(), sub(), mul() and div() inside a module named demo.

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. **import statement**
2. **from-import statement**

Python Modules

Python modules are a great way to organize and reuse code across different programs. Think of a module as a file containing Python code that defines functions, classes, and variables you can use in your own programs. By importing these modules, you can make use of their functionality without having to rewrite code.

"Modules" in Python are files that contain Python code. It can contain functions, classes, and variables, and you can use them in your scripts. It help to reuse code and keep the project organized.

Types of Python modules

1. **Built-in Modules**

These are already available in Python. Example:

- math
- os
- random
- sys

2. **User-defined Modules**

These modules are created by the user. For example, if you write functions and code in a file and use it in another program, it becomes your user-defined module.
import statement to use the module
import is used to access a module in Python.
Importing a Module Using import <module>
When you use the import statement, you're telling Python to load a module so you can use its functions, classes, or variables. For example:

```
import math

# Using a function from the math module
result = math.sqrt(16)
print(result)  # Output: 4.0
```

In this example:

- import math loads the math module.
- You access functions in the math module using the dot notation (math.sqrt()), where sqrt is a function that calculates the square root of a number.

Using From Statement to Import Specific Components
If you only need specific functions or variables from a module, you can use the from <module> import <item> syntax. This approach makes your code cleaner and more efficient by importing only what you need.

```
from math import sqrt, pi

# Using the imported functions and variables directly
result = sqrt(25)
print(result)  # Output: 5.0

print(pi)  # Output: 3.141592653589793
```

In this example:

- from math import sqrt, pi imports only the sqrt function and pi constant from the math module.
- You can then use sqrt() and pi directly without prefixing them with math.

Importing and Using the Math Module
The math module in Python provides various mathematical functions and constants that make it easier to perform complex calculations. By importing the math module, you gain access to functions and constants such as pi, e, and mathematical functions like sqrt(), ceil(), and sin(). Here's a quick guide on how to use these features:
Importing the Math Module
First, you need to import the math module:

```
import math
```

Using Constants

- **math.pi**: The mathematical constant π (pi), approximately equal to 3.14159.
- **math.e**: The mathematical constant e, approximately equal to 2.71828.

Example:

```
import math
```

print("Value of pi:", math.pi) # Output: Value of pi: 3.141592653589793
print("Value of e:", math.e)   # Output: Value of e: 2.718281828459045

Using Functions

- **math.sqrt(x)**: Returns the square root of x.
  print("Square root of 16:", math.sqrt(16)) # Output: Square root of 16: 4.0
- **math.ceil(x)**: Returns the smallest integer greater than or equal to x.
  print("Ceiling of 4.2:", math.ceil(4.2)) # Output: Ceiling of 4.2: 5
- **math.floor(x)**: Returns the largest integer less than or equal to x.
  print("Floor of 4.7:", math.floor(4.7)) # Output: Floor of 4.7: 4
- **math.pow(x, y)**: Returns x raised to the power of y.
  print("2 to the power of 3:", math.pow(2, 3)) # Output: 2 to the power of 3: 8.0
- **math.fabs(x)**: Returns the absolute value of x.
  print("Absolute value of -5.3:", math.fabs(-5.3)) # Output: Absolute value of -5.3: 5.3
- **math.sin(x), math.cos(x), math.tan(x)**: Return the sine, cosine, and tangent of x (where x is in radians).

```
import math

angle = math.pi / 4  # 45 degrees in radians
print("Sine of 45 degrees:", math.sin(angle))  # Output: Sine of 45 degrees: 0.7071067811865475
print("Cosine of 45 degrees:", math.cos(angle))  # Output: Cosine of 45 degrees:
0.7071067811865476
print("Tangent of 45 degrees:", math.tan(angle))  # Output: Tangent of 45 degrees:
0.9999999999999999
```

Example: built-in module

```
import math
print(math.sqrt(16)) # Output: 4.0
```

Example: User-Defined Module

Suppose, you have a file my_module.py which contains this code:

```
# my_module.py
def greet(name):
   return f”Hello, {name}!”
```

Now you can import it into your script like this:

```
import my_module
print(my_module.greet(“Rajveer”)) # Output: Hello, Rajveer!
```

Using from ... import

You can also import just a specific function or variable from a module:

```
from math import sqrt
print(sqrt(25)) # Output: 5.0
```

Benefits of Python modules

1. Code Reusability: Code written once can be used again and again.
2. Organized Code: Dividing the code into separate files makes it easier to read and understand.
3. Use of built-in tools: The work can be made simpler and faster with the built-in modules available in Python.

Datetime module:

Python’s datetime module is used to manage time and date. It simplifies timestamps, dates, times, and time calculations.

Below the main parts of datetime module and their usage:

using datetime module

Importing the datetime module:

import datetime

1. **Getting the current date and time**

Example:

```
import datetime
# Current date and time
current_datetime = datetime.datetime.now()
print("Current date and time:", current_datetime)
```

Output:

Current date and time: 2024-12-17 12:34:56.789012

2. **Getting only the date**

Example:

```
import datetime
# today's date
current_date = datetime.date.today()
print("Today's date:", current_date)
```

Output:

Today's date: 2024-12-17

3. **Setting custom date and time**

You can set your own date and time.

Example:

```
import datetime
# Custom date and time
custom_datetime = datetime.datetime(2024, 12, 25, 10, 30, 45)
print("Custom date and time:", custom_datetime)
```

Output:

Custom date and time: 2024-12-25 10:30:45

4. **Extracting separate parts of date and time**

Example:

```
import datetime
now = datetime.datetime.now()
print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
print("hour:", now.hour)
print("minute:", now.minute)
print("Second:", now.second)
```

5. **Date and Time Difference**

You can find the time difference by using timedelta.

Example:

```
import datetime
# today's date
today = datetime.date.today()
# custom date
future_date = datetime.date(2024, 12, 25)
# find the difference
difference = future_date – today
print("Difference:", difference.days, "days")
```

Output:

Gap: 8 days

6. **Formatting the date and time**

use of strftime
You can format the date and time as per your choice.
Example:
```
import datetime
now = datetime.datetime.now()
formatted_date = now.strftime("%d-%m-%Y %H:%M:%S")
print("Formatted date and time:", formatted_date)
```
Formatting Codes:
code meaning
%d days (01-31)
%m month (01-12)
%Y Year (four digits)
%H Hours (in 24-hour format)
%M minutes
%S seconds

7. **Parsing Strings**

Using strptime:
Example:
```
import datetime
date_str = "25-12-2024"
parsed_date = datetime.datetime.strptime(date_str, "%d-%m-%Y")
print("Parsed Date:", parsed_date)
```

8. **Time Handling**

Getting time only:
```
import datetime
# time only
time_now = datetime.datetime.now().time()
print("Current time:", time_now)
```
9.UTC Time
Example:
```
import datetime
utc_now = datetime.datetime.utcnow()
print("Current UTC time:", utc_now)
```
Main class of datetime module:
class use
date Working only with dates
time working only with time
datetime Working with date and time
timedelta for time interval
Calendar modules:
The calendar module in Python is used to perform calendar related tasks. With the help of this module you can create calendars, calculate dates and days, and get information about the year or month.
Using the calendar module
First you need to import the module:
```
import calendar
```

1. **Show month calendar**

Example:
```
import calendar
# Show calendar of any month
```

print(calendar.month(2024, 12)) # Calendar for December 2024

Output:

December 2024
Mo Tu We Th Fr Sa Su
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

math module in python

Python's math module is used to make mathematical calculations simpler and faster. It comes with a host of pre-built functions that allow you to perform advanced calculations, trigonometry, logarithms, and other mathematical operations with ease.

using the math module

Importing the math module:

import math

Main functions of math module

1. **Basic Mathematical Functions**

function description example
math.ceil(x) Rounds the number upward math.ceil(4.2) → 5
math.floor(x) rounds the number down math.floor(4.8) → 4
math.fabs(x) gives the magnitude of the number math.fabs(-5) → 5.0
math.factorial(x) gives the factorial of the number math.factorial(5) → 120
math.gcd(a, b) Greatest Common Factor (GCD) of two numbers math.gcd(12, 18) → 6

2. **Exponential and Logarithm Functions**

function description example
math.exp(x) calculates e^x math.exp(2) → 7.389
math.log(x, base) Finds the logarithm (base is optional) math.log(8, 2) → 3.0
math.log10(x) Finds the logarithm of base 10 math.log10(100) → 2.0
math.pow(x, y) finds x to the power y math.pow(2, 3) → 8.0
math.sqrt(x) finds the square root math.sqrt(16) → 4.0

3. **Trigonometry Functions**

function description example
math.sin(x) sine of angle in radians math.sin(math.pi/2) → 1.0
math.cos(x) cosine of angle in radians math.cos(0) → 1.0
math.tan(x) Tangent of angle in radians math.tan(math.pi/4) → 1.0
math.asin(x) inverse sine math.asin(1) → 1.5708
math.acos(x) inverse cosine math.acos(1) → 0.0
math.atan(x) inverse tangent math.atan(1) → 0.7854
math.degrees(x) converts radians to degrees math.degrees(math.pi) → 180.0
math.radians(x) converts degrees to radians math.radians(180) → 3.14159

4. **Float and Fraction Functions**

function description example
math.modf(x) returns decimal and integer parts of the number math.modf(4.5) → (0.5, 4.0)
math.trunc(x) returns integer division of a number math.trunc(4.9) → 4

5. **Special Functions**

function description example
math.pi Value of π (pi) math.pi → 3.141592653589793
math.e Value of e (Natural Logarithm Base) math.e → 2.718281828459045

math.inf Value of Infinity math.inf → ∞
math.nan Value of NaN (Not a Number) math.nan → NaN
examples of math module
Example 1: to calculate

```
import math
radius = 5
area = math.pi * math.pow(radius, 2)
print("Area of circle:", area) # Output: 78.53981633974483
```

Escape Sequence Characters:
Sequence of characters after backslash '\' [Escape Sequence Characters]
Escape Sequence Characters comprises of more than one character but represents one character when
used within the string.
Examples: \n (new line), \t (tab), \' (single quote), \\ (backslash), etc

Data Structures in python
Python can able to create different types of applications like web, desktop, Data Science, Artificial Intelligence and etc… for creating that kind of application mostly possible using data. Data is playing an important role that means data stored inefficiently as well as access in a timely. This process will be done using a technique called Data Structures.
Data Structures
Data Structure is a process of manipulates the data. Manipulate is a process of organizing, managing and storing the data in different ways. It is also easier to get the data and modify the data. The data structure allows us to store the data, modify the data and also allows to compare the data to others. Also, it allows performing some operations based on data.
Types of Data Structures in python
Generally, all the programming language allows doing the programming for these data structures. In python also having some inbuilt operations to do that. Also in python, it divided the data structures into two types.

- Built-in Data Structures
- User-Defined Data Structures

Built-in Data Structures
Python having some implicit data structure concepts to access and store the data. The following are the implicit or Built-in Data structures in python.

- List
- Tuple
- Dict
- Set

List, Tuple, Set, and Dictionary
List, Tuple, Set, and Dictionary are core data structures in Python, and understanding them is essential for writing efficient, clean code. Each one serves a unique role: use lists and tuples to store sequences, sets to keep unordered unique items, and dictionaries to manage key-value pairs. When you know how and when to use each, you can streamline your Python development.
Python's List, Tuple, Set, and Dictionary
List
A list in Python is a collection of elements that is ordered and mutable. Lists are created using square brackets [] and can hold elements of different data types.

Example:
my_list = [1, 'apple', True]
print(my_list)
Advantages:

- Mutable – Elements can be added, removed, or modified.
- Ordered – Elements maintain their order in the list.

Disadvantages:

- Slower performance for operations like insertion and deletion compared to sets and dictionaries.

Use Cases and Applications:

- Storing collections of similar items.
- Iterating over elements.

Tuple
A tuple in Python is a collection of elements that is ordered and immutable. Tuples are created using parentheses () and can also hold elements of different data types.
Example:
my_tuple = (1, 'banana', False)
print(my_tuple)
Advantages:

- Immutable – Elements cannot be changed after creation.
- Faster access time compared to lists for read-only operations.

Disadvantages:

- Cannot be modified once created.

Use Cases and Applications:

- Used for fixed collections of elements where immutability is desired.
- Returning multiple values from a function.

Set
A set in Python is a collection of unique elements that is unordered and mutable. Sets are created using curly braces {} or the set() function.
Example:
my_set = {1, 2, 3}
print(my_set)
Advantages:

- Contains unique elements only, eliminating duplicates.
- Fast membership testing and operations like intersection and union.

Disadvantages:

- Not ordered – Elements are not stored in a specific order.

Use Cases and Applications:

- Removing duplicates from a list.
- Checking for membership or common elements between sets.

Dictionary
You create dictionaries in Python using curly braces {} and colons : to define key-value pairs.
They let you store unordered and mutable collections of data.
Example:
my_dict = {'key1': 'value1', 'key2': 2}
print(my_dict)

Advantages:

- Fast lookups based on keys.
- Key-value pair structure allows for easy data retrieval.

Disadvantages:

- Not ordered – No guarantee on the order of key-value pairs.

Use Cases and Applications:

- Storing data with a key for easy retrieval.
- Mapping unique identifiers to values for quick access.

Key Differences: List, Tuple, Set, and Dictionary

List	Tuple	Set	Dictionary
Mutable	Immutable	Mutable	Mutable
Defined with square brackets []	Defined with parentheses ()	Defined with curly braces {}	Defined with curly braces {} (key-value pairs)
Supports item assignment and deletion	Does not support item assignment or deletion	Supports item addition and deletion	Supports item addition, deletion, and modification of values
Has more built-in methods like append(), extend()	Has fewer built-in methods compared to list	Has methods like add(), remove(), and discard()	Has methods like keys(), values(), items(), and get()
Can contain duplicate elements	Can contain duplicate elements	Cannot contain duplicate elements	Keys must be unique, but values can be duplicated
Ordered collection	Ordered collection	Unordered collection	Ordered collection (as
Can be used as keys in dictionaries	Can be used as keys in dictionaries if they contain only hashable elements	Cannot be used as keys in dictionaries	Keys are used as identifiers in dictionaries
Slower iteration compared to tuples	Faster iteration than lists	Faster iteration than lists	Moderately fast iteration depending on the number of keys
Can store mixed data types	Can store mixed data types	Can store mixed data types	Can store mixed data types (keys and values)
More memory consumption due to mutability	Less memory consumption due to immutability	More memory-efficient compared to lists	Memory-efficient, especially for large datasets
Commonly used for variable-sized collections	Preferred for fixed-size collections or when immutability is required	Preferred for collections where uniqueness is important	Used to store key-value pairs for efficient data lookup
Can contain any data type	Can contain any data type	Can contain any data type	Keys must be hashable, values can be any data type
Can contain nested lists	Can contain nested tuples	Can contain nested sets or frozensets	Can contain nested dictionaries
More flexible in terms of operations	More secure in terms of data integrity	Efficient for membership tests	Efficient for quick lookups and data

List	Tuple	Set	Dictionary association
Commonly used for data that needs frequent updates	Used for data that should not change over time	Used for data that must have unique elements	Commonly used for key-value mappings

Practical Implementation

Lists in Python

Lists are versatile data structures in Python that can hold heterogeneous elements.

Practical Implementation Example:

Let's create a list of fruits and print each fruit:

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

Best Practices and Optimization Tips:

Use list comprehension for concise and efficient code:

```
squared_nums = [num2 for num in range(1, 6)]
```

Common Pitfalls and Solutions:

A common pitfall is modifying a list while iterating over it. To avoid this, create a copy of the list:

```
numbers = [1, 2, 3]
for num in numbers[:]:
    numbers.append(num * 2)
```

Tuples in Python

Tuples are immutable sequences in Python, typically used to represent fixed collections of items.

Practical Implementation Example:

Creating a tuple of coordinates:

```
coordinates = (10, 20)
x, y = coordinates
print(f'x: {x}, y: {y}')
```

Best Practices and Optimization Tips:

Use tuples as keys in dictionaries for efficient lookups:

```
point = (3, 4)
distances = {(0, 0): 0, (1, 1): 1}
distance_to_origin = distances.get(point, -1)
```

Common Pitfalls and Solutions:

Attempting to modify a tuple will result in an error. If mutability is required, consider using a list.

Sets in Python

Use sets in Python when you need to store unique elements, test for membership, or eliminate duplicates without caring about order.

Practical Implementation Example:

Creating a set of unique letters in a word:

```
word = 'hello'
```

```
unique_letters = set(word)
print(unique_letters)
```
Best Practices and Optimization Tips:
Use set operations like intersection, union, and difference for efficient manipulation of data:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection = set1 & set2
```
Common Pitfalls and Solutions:
Accidentally mutating a set during iteration can lead to unexpected behavior. To prevent this, operate on a copy of the set.

Dictionaries in Python
Dictionaries are key-value pairs that allow for fast lookups and mappings between items.
Practical Implementation Example:
Creating a dictionary of phone contacts:

```
contacts = {'Alice': '555-1234', 'Bob': '555-5678'}
print(contacts['Alice'])
```
Best Practices and Optimization Tips:
Use dictionary comprehensions for concise creation of dictionaries:

```
squared_dict = {num: num2 for num in range(1, 6)}
```

Array
An array is defined as a container that stores the collection of items at contiguous memory locations. The array is an idea of storing multiple items of the same type together and it makes it easier to calculate the position of each element. It is used to store multiple values in a single variable.
Creating an Array
For creating an array in Python, we need to import the array module.
After importing, the array module we just have to use the array function which is used to create the arrays in Python.
Syntax

```
import array as arr

arrayName = arr.array(code for data type, [array and its items])
```

Code for Data Types which are used in array Function

Code Type	Python Type	Full Form	Size(in Bytes)
u	unicode character	Python Unicode	2
b	int	Signed Char	1
B	int	Unsigned Char	1
h	int	Signed Short	2
l	int	Signed Long	4
L	int	Unsigned Long	4
q	int	Signed Long Long	8
Q	int	Unsigned Long Long	8
H	int	Unsigned Short	2

Code Type	Python Type	Full Form	Size(in Bytes)
f	float	Float	4
d	float	Double	8
i	int	Signed Int	2
I	int	Unsigned Int	2

Example

Here, we have created an array with the name myArr and printed it to get the output.

```
import array as arr

myArr = arr.array('d', [20, 35, 55, 65])

print(myArr)
```

Explanation

Adding Element to Array

As we all know, arrays are mutable in nature. So we can add an element to the existing array.

We can use 2 different methods to add the elements to the existing array. These methods are:

- .append(): This is used to add a single element to the existing array. By using the append method the element gets added at the end of the existing array.
- .extend(): This is used to add an array to the existing array. By using the extend method the array that we have added to the extend function gets merged at the end of the existing array. And, finally, we got the updated array where we have the combination of the original and the new array.

Let's checkout with the help of an example of how these functions are used to add elements at the end of the array.

Syntax

```
import array as arr
arrayName = arr.array(code for data type, [array and its items])
##sYNTAX OF APPEND FUNCTION
arrayName.append(single element to be passed inside the array)
# SYNTAX OF EXTEND FUNCTION
arrayName.extend(new array which we want to add in the original array)
```

Example

This code helps us to understand how we can use .append() and .extend(), to add the elements in the array.

```
import array as arr
myArr = arr.array('i', [20, 35, 55, 65])

# use of append function to add the element
myArr.append(77)
print('After use of append(), updated array is:', myArr)

# use of extend function to add the list of element
myArr.extend([1, 2, 3, 4])
print('After use of extend(), updated array is:', myArr)
```

Accessing Elements from Array

We can access the elements of the array by using the index of the elements of the array.

Note: If we are trying to access that index of the array which is greater than the last index of the array then it will raise the out of bound error.

Example
This code access the element of the array at a particular index.

```
import array as arr

myArr = arr.array('i', [20, 35, 55, 65])

indOfElement = 2

accessedElement = myArr[indOfElement]

print('Element at 2nd index of array is: ', accessedElement)
```

Removing Elements from Array
We can use the .remove() and pop() functions to remove the elements from the array.

- The pop function deletes the element which is present at the last index of the array.
- The remove function takes an element as a parameter that should be removed from the array. If there are duplicates of the same element which is to be removed from the array, then it will remove the first occurrence of that element.