**Python Tutorial**

[](https://training.javatpoint.com/python-development-training)

Python tutorial provides basic and advanced concepts of Python. Our Python tutorial is designed for beginners and professionals.

Python is a simple, general purpose, high level, and object-oriented programming language.

Python is an interpreted scripting language also. *Guido Van Rossum* is known as the founder of Python programming.

Our Python tutorial includes all topics of Python Programming such as installation, control statements, Strings, Lists, Tuples, Dictionary, Modules, Exceptions, Date and Time, File I/O, Programs, etc. There are also given Python interview questions to help you better understand Python Programming.

**What is Python**

**Python** is a general purpose, dynamic, high-level, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Python is *easy to learn* yet powerful and versatile scripting language, which makes it attractive for Application Development.

Python's syntax and *dynamic typing* with its interpreted nature make it an ideal language for scripting and rapid application development.

Python supports *multiple programming pattern*, including object-oriented, imperative, and functional or procedural programming styles.

Python is not intended to work in a particular area, such as web programming. That is why it is known as *multipurpose* programming language because it can be used with web, enterprise, 3D CAD, etc.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write a=10 to assign an integer value in an integer variable.

Python makes the development and debugging *fast* because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

**Python 2 vs. Python 3**

In most of the programming languages, whenever a new version releases, it supports the features and syntax of the existing version of the language, therefore, it is easier for the projects to switch in the newer version. However, in the case of Python, the two versions Python 2 and Python 3 are very much different from each other.

A list of differences between Python 2 and Python 3 are given below:

1. Python 2 uses **print** as a statement and used as print "something" to print some string on the console. On the other hand, Python 3 uses **print** as a function and used as print("something") to print something on the console.
2. Python 2 uses the function raw\_input() to accept the user's input. It returns the string representing the value, which is typed by the user. To convert it into the integer, we need to use the int() function in Python. On the other hand, Python 3 uses input() function which automatically interpreted the type of input entered by the user. However, we can cast this value to any type by using primitive functions (int(), str(), etc.).
3. In Python 2, the implicit string type is ASCII, whereas, in Python 3, the implicit string type is Unicode.
4. Python 3 doesn't contain the xrange() function of Python 2. The xrange() is the variant of range() function which returns a xrange object that works similar to Java iterator. The range() returns a list for example the function range(0,3) contains 0, 1, 2.
5. There is also a small change made in Exception handling in Python 3. It defines a keyword **as** which is necessary to be used. We will discuss it in Exception handling section of Python programming tutorial.

**Java vs Python Program**

Unlike the other programming languages, Python provides the facility to execute the code using few lines. **For example** - Suppose we want to print the **"Hello World"** program in Java; it will take three lines to print it.

**Java Program**

1. public class HelloWorld {
2. public static void main(String[] args){
3. // Prints "Hello, World" to the terminal window.
4. System.out.println("Hello World");
5. }
6. }

**Python Program**

On the other hand, we can do this using one statement in Python.

print("Hello World")

Both programs will print the same result, but it takes only one statement without using a semicolon or curly braces in Python.

**Python Basic Syntax**

There is no use of curly braces or semicolon in Python programming language. It is English-like language. But Python uses the indentation to define a block of code. Indentation is nothing but adding whitespace before the statement when it is needed. **For example -**

1. def func():
2. statement 1
3. statement 2
4. …………………
5. …………………
6. statement N

In the above example, the statements that are same level to right belong to the function. Generally, we can use four whitespaces to define indentation.

**Python History**

Python was invented by **Guido van Rossum** in 1991 at CWI in Netherland. The idea of Python programming language has taken from the ABC programming language or we can say that ABC is a predecessor of Python language.

There is also a fact behind the choosing name Python. Guido van Rossum was a fan of the popular BBC comedy show of that time, **"Monty Python's Flying Circus"**. So he decided to pick the name **Python** for his newly created programming language.

Python has the vast community across the world and releases its version within the short period.

**Why learn Python?**

Python provides many useful features to the programmer. These features make it most popular and widely used language. We have listed below few-essential feature of Python.

* Easy to use and Learn
* Expressive Language
* Interpreted Language
* Object-Oriented Language
* Open Source Language
* Extensible
* Learn Standard Library
* GUI Programming Support
* Integrated
* Embeddable
* Dynamic Memory Allocation
* Wide Range of Libraries and Frameworks

**Where is Python used?**

Python is a general-purpose, popular programming language and it is used in almost every technical field. The various areas of Python use are given below.

* Data Science
* Date Mining
* Desktop Applications
* Console-based Applications
* Mobile Applications
* Software Development
* Artificial Intelligence
* Web Applications
* Enterprise Applications
* 3D CAD Applications
* Machine Learning
* Computer Vision or Image Processing Applications.
* Speech Recognitions

**Python Popular Frameworks and Libraries**

Python has wide range of libraries and frameworks widely used in various fields such as machine learning, artificial intelligence, web applications, etc. We define some popular frameworks and libraries of Python as follows.

* **Web development (Server-side) -** Django Flask, Pyramid, CherryPy
* **GUIs based applications -** Tk, PyGTK, PyQt, PyJs, etc.
* **Machine Learning -** TensorFlow, PyTorch, **Scikit-learn**, Matplotlib, Scipy, etc.
* **Mathematics -** Numpy, Pandas, etc.

**Python print() Function**

The **print()** function displays the given object to the standard output device (screen) or to the text stream file.

Unlike the other programming languages, Python **print()** function is most unique and versatile function.

The syntax of **print()** function is given below.

1. print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Let's explain its parameters one by one.

* **objects -** An object is nothing but a statement that to be printed. The \* sign represents that there can be multiple statements.
* **sep -** The **sep** parameter separates the print values. Default values is ' '.
* **end -** The **end** is printed at last in the statement.
* **file -** It must be an object with a write(string) method.
* **flush -** The stream or file is forcibly flushed if it is true. By default, its value is false.

Let's understand the following example.

**Example - 1: Return a value**

1. print("Welcome to javaTpoint.")
3. a = 10
4. # Two objects are passed in print() function
5. print("a =", a)
7. b = a
8. # Three objects are passed in print function
9. print('a =', a, '= b')

**Output:**

Welcome to javaTpoint.

a = 10

a = 10 = b

As we can see in the above output, the multiple objects can be printed in the single **print()** statement. We just need to use comma (,) to separate with each other.

**Example - 2: Using sep and end argument**

1. a = 10
2. print("a =", a, sep='dddd', end='\n\n\n')
3. print("a =", a, sep='0', end='$$$$$')

**Output:**

a =dddd10

a =010$$$$$

In the first **print()** statement, we use the **sep** and **end** arguments. The given object is printed just after the **sep** values. The value of end parameter printed at the last of given object. As we can see that, the second **print()** function printed the result after the three black lines.

**Taking Input to the User**

Python provides the **input()** function which is used to take input from the user. Let's understand the following example.

**Example -**

1. name = input("Enter a name of student:")
2. print("The student name is: ", name)

**Output:**

Enter a name of student: Devansh

The student name is: Devansh

By default, the **input()** function takes the string input but what if we want to take other data types as an input.

If we want to take input as an integer number, we need to typecast the **input()** function into an integer.

**For example -**

**Example -**

1. a  = int(input("Enter first number: "))
2. b = int(input("Enter second number: "))
4. print(a+b)

**Output:**

Enter first number: 50

Enter second number: 100

150

We can take any type of values using **input()** function.

**Python Operators**

Operators are the symbols which perform various operations on Python objects. Python operators are the most essential to work with the Python data types. In addition, Python also provides identify membership and bitwise operators. We will learn all these operators with the suitable example in following tutorial.

* **Python Operators**

**Python Conditional Statements**

Conditional statements help us to execute a particular block for a particular condition. In this tutorial, we will learn how to use the conditional expression to execute a different block of statements. Python provides if and else keywords to set up logical conditions. The elif keyword is also used as conditional statement.

* **Python if..else statement**

**Python Loops**

Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times. For this purpose, the programming languages provide various types of loops capable of repeating some specific code several times. Consider the following tutorial to understand the statements in detail.

* **Python Loops**
* **Python For Loop**
* **Python While Loop**

**Python Data Structures**

Data structures are referred which can hold some data together or we say that they are used to store the data in organized way. Python provides built-in data structures such as **list, tuple, dictionary, and set**. We can perform complex tasks using data structures.

**Python List**

Python list holds the ordered collection of items. We can store a sequence of items in a list. Python list is mutable which means it can be modified after its creation. The items of lists are enclosed within the square bracket [] and separated by the comma. Let's see the example of list.

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]

If we try to print the type of L1, L2, and L3 using **type()** function then it will come out to be a list.

1. print(type(L1))
2. print(type(L2))

**Output:**

<class 'list'>

<class 'list'>

To learn more about list, visit the following tutorial.

* **Python List**
* **Python List Functions**

**Python Tuple**

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

Tuple can be defined as follows

**Example -**

1. tup = ("Apple", "Mango" , "Orange" , "Banana")
2. print(type(tup))
3. print(tup)

**Output:**

<class 'tuple'>

('Apple', 'Mango', 'Orange', 'Banana')

If we try to add new to the tuple, it will throw an error.

**Example -**

1. tup = ("Apple", "Mango" , "Orange" , "Banana")
3. tup[2] = "Papaya"
4. print(tup)

**Output:**

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/gamewithturtle.py", line 3, in

tup[2] = "Papaya"

TypeError: 'tuple' object does not support item assignment

The above program throws an error because tuples are immutable type. To learn more about tuple, visit the Python Tuples.

* **Python Tuple**

**Python String**

Python string is a sequence of characters. It is a collection of the characters surrounded by single quotes, double quotes, or triple quotes. It can also define as collection of the Unicode characters. We can create a string as follows.

**Example -**

1. # Creating string using double quotes
2. str1 = "Hi Python"
3. print(str1)
4. # Creating string using single quotes
5. str1 = 'Hi Python'
6. print(str1)
7. # Creating string using triple quotes
8. str1 = '''Hi Python'''
9. print(str1)

**Output:**

Hi Python

Hi Python

Hi Python

Python doesn't support the character data-type. A single character written as 'p' is treated as a string of length 1.

Stings are also immutable. We can't change after it is declared. To learn more about the string, visit the following tutorial.

* **Python Strings**
* **Python String Method**

**Dictionaries**

Python Dictionary is a most efficient data structure and used to store the large amount of data. It stores the data in the key-value pair format. Each value is stored corresponding to its key.

Keys must be a unique and value can be any type such as integer, list, tuple, etc.

It is a mutable type; we can reassign after its creation. Below is the example of creating dictionary in Python.

**Example -**

1. employee = {"Name": "John", "Age": 29, "salary":250000,"Company":"GOOGLE"}
2. print(type(employee))
3. print("printing Employee data .... ")
4. print(employee)

**Output:**

<class 'dict'>

Printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 250000, 'Company': 'GOOGLE'}

The empty curly braces {} are used to create empty dictionary. To learn more, visit the complete tutorial of the dictionary.

* **Python Dictionary**
* **Python Dictionary Methods**

**Python Sets**

A Python set is a collection of unordered elements. Each element in set must be unique and immutable. Sets are mutable which means we can modify anytime throughout the program. Let's understand the example of creating set in Python.

**Example -**

1. # Creating Set
2. Month = {"January", "February", "March", "April", "May", "June", "July"}
3. print(Month)
4. print(type(Month))

**Output:**

{'March', 'July', 'April', 'May', 'June', 'February', 'January'}

<class 'set'>

To get the more information about sets, visit the following resources.

* **Python Sets**
* **Python Set Methods**

**Python Functional Programming**

This section of Python tutorial defines some important tools related to functional programming such as **lambda and recursive functions**. These functions are very efficient in accomplishing the complex tasks. We define a few important functions, such as **reduce, map,** and **filter.** Python provides the **functools** module that includes various **functional programming tools**. Visit the following tutorial to learn more about functional programming.

* Python Function
* Python map() Function
* Python filter() Function
* Python reduce() Function
* Python functool Module
* Python Lambda Function

**Python File I/O**

Files are used to store data in a computer disk. In this tutorial, we explain the built-in file object of Python. We can open a file using Python script and perform various operations such as writing, reading, and appending. There are various ways of opening a file. We are explained with the relevant example. We will also learn to perform read/write operations on binary files.

* **Python File I/O**

**Python Modules**

Python modules are the program files that contain a Python code or functions. There are two types of module in the Python - User-define modules and built-in modules. A module that the user defines, or we can say that our Python code saved with **.py** extension, is treated as a user-define module.

Built-in modules are predefined modules of Python. To use the functionality of the modules, we need to import them into our current working program.

* **Python Modules**

**Python Exceptions**

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error.

* **Python Exceptions**

**Python CSV**

A **csv** stands for "comma separated values", which is defined as a simple file format that uses specific structuring to arrange tabular data. It stores tabular data such as spreadsheet or database in plain text and has a common format for data interchange. A **csv** file opens into the excel sheet, and the rows and columns data define the standard format. Visit the following tutorial to learn the CSV module in detail.

* **Python Read CSV File**
* **Python Write CSV File**

**Python Sending Mail**

We can send or read a mail using the Python script. Python's standard library modules are useful for handling various protocols such as PoP3 and IMAP. We will learn how to send a mail with the popular email service SMTP from a Python script.

* **Python Sending Emails**

**Python Magic Methods**

Python magic method is defined as the special method which adds "magic" to a class. It starts and ends with double underscores, for example, **\_init\_** or **\_str\_.**

The built-in classes define many magic methods. The **dir()** function can be used to see the number of magic methods inherited by a class. It has two prefixes and suffix underscores in the method name.

* **Python Magic Methods**

**Python Oops Concepts**

Everything in Python is treated as an object including integer values, floats, functions, classes, and none. Apart from that, Python supports all oriented concepts. Below is the brief introduction of oops concepts of Python.

* **Classes and Objects -** Python classes are the blueprint of the object. An object is a collection of data and method that act on the data.
* **Inheritance -** An inheritance is a technique where one class inherits the properties of other classes.
* **Constructor -** Python provides a special method **\_\_init\_\_()** which is known as a constructor. This method is automatically called when an object is instantiated.
* **Data Member -** A variable that holds data associated with a class and its objects.

To read the oops concept in detail, visit the following resources.

* **Python Oops Concepts**
* **Python Object and classes**
* **Python Constructor**
* **Python Inheritance**
* **Python Polymorphism**

**Python Advance Topics**

Python includes many advance and useful concepts that help the programmer to solve the complex tasks. These concepts are given below.

**Python Iterator**

An iterator is simply an object that can be iterated upon. It returns one object at a time. It can be implemented using the two special methods, **\_\_iter\_\_() and \_\_next\_\_()**.

To learn more about the iterators visit our **Python Iterators** tutorial.

**Python Generators**

The Generators are an easiest way of creating Iterators. To learn more about, visit our **Python Generators** tutorial.

**Python Decorators**

These are used to modify the behavior of the function. Decorators provide the flexibility to wrap another function to expand the working of wrapped function, without permanently modifying it.

To learn more about, visit the **Python Decorators** tutorial.

**Python Database Connections**

We can use various databases along with Python. You can learn the full tutorial to visit below resources. Python DBI-API acclaims standard sets of functionality to be included in the database connectivity modules for respective RDBMS products. We explain all important database connectivity using Python DBI-API.

**Python MySQL**

Environment Setup

Database Connection

Creating New Database

Creating Tables

Insert Operation

Read Operation

Update Operation

Join Operation

Performing Transactions

**Python MongoDB**

Python MongoDB

**Python SQLite**

Python SQLite

**Python CGI**

Python CGI stands for **"Common Gateway Interface",** which is used to define how to exchange information between the webserver and a custom Python scripts. The **Common Gateway Interface** is a standard for external gateway programs to interface with the server, such as HTTP Servers. To learn more about Python CGI, visit the following tutorial.

* **Python CGI**

# Python Features

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

### 1) Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

### 2) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.

### 3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

### 4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

### 5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

### 6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

### 7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

### 8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

### 9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

### 10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

### 11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

### 12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15.

# Python History and Versions

* Python laid its foundation in the late 1980s.
* The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
* In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
* In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
* Python 2.0 added new features such as list comprehensions, garbage collection systems.
* On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
* ABC programming language is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.
* The following programming languages influence Python:
  + ABC language.
  + Modula-3

## Why the Name Python?

There is a fact behind choosing the name Python. **Guido van Rossum** was reading the script of a popular BBC comedy series "**Monty Python's Flying Circus**". It was late on-air 1970s.

Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the **"Monty Python's Flying Circus"** for their newly created programming language.

The comedy series was creative and well random. It talks about everything. Thus it is slow and unpredictable, which made it very interesting.

Python is also versatile and widely used in every technical field, such as Machine Learning, Artificial Intelligence, Web Development, Mobile Application, Desktop Application, Scientific Calculation, etc.

## Python Version List

Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.

A list of Python versions with its released date is given below.

|  |  |
| --- | --- |
| **Python Version** | **Released Date** |
| Python 1.0 | January 1994 |
| Python 1.5 | December 31, 1997 |
| Python 1.6 | September 5, 2000 |
| Python 2.0 | October 16, 2000 |
| Python 2.1 | April 17, 2001 |
| Python 2.2 | December 21, 2001 |
| Python 2.3 | July 29, 2003 |
| Python 2.4 | November 30, 2004 |
| Python 2.5 | September 19, 2006 |
| Python 2.6 | October 1, 2008 |
| Python 2.7 | July 3, 2010 |
| Python 3.0 | December 3, 2008 |
| Python 3.1 | June 27, 2009 |
| Python 3.2 | February 20, 2011 |
| Python 3.3 | September 29, 2012 |
| Python 3.4 | March 16, 2014 |
| Python 3.5 | September 13, 2015 |
| Python 3.6 | December 23, 2016 |
| Python 3.7 | June 27, 2018 |
| Python 3.8 | October 14, 2019 |

## Tips to Keep in Mind While Learning Python

The most common question asked by the beginners - **"What is the best way to learn Python"?** It is the initial and relevant question because first step in learning any programming language is to know how to learn.

The proper way of learning will help us to learn fast and become a good Python developer.

In this section, we will discuss various tips that we should keep in mind while learning Python.

### 1. Make it Clear Why We Want to Learn

The goal should be clear before learning the Python. Python is an easy, a vast language as well. It includes numbers of libraries, modules, in-built functions and data structures. If the goal is unclear then it will be a boring and monotonous journey of learning Python. Without any clear goal, you perhaps won't make it done.

So, first figure out the motivation behind learning, which can anything be such as knowing something new, develop projects using Python, switch to Python, etc. Below are the general areas where Python is widely used. Pick any of them.

* Data Analysis and Processing
* Artificial Intelligence
* Games
* Hardware/Sensor/Robots
* Desktop Applications

Choose any one or two areas according to your interest and start the journey towards learning Python.

### 2. Learn the Basic Syntax

It is the most essential and basic step to learn the syntax of the Python programming language. We have to learn the basic syntax before dive deeper into learning it. As we have discussed in our earlier tutorial, Python is easy to learn and has a simple syntax. It doesn't use semicolon and brackets. Its syntax is like the English language.

So it will take minimum amount of time to learning its syntax. Once we get its syntax properly, further learning will be easier and quicker getting to work on projects.

#### Note - Learn Python 3, not Python 2.7, because the industry no longer uses it. Our Python tutorial is based on its latest version Python 3.

### 3. Write Code by Own

Writing the code is the most effective and robust way to learn Python. First, try to write code on paper and run in mind (Dry Run) then move to the system. Writing code on paper will help us get familiar quickly with the syntax and the concept store in the deep memory. While writing the code, try to use proper functions and suitable variables names.

There are many editors available for Python programming which highlights the syntax related issue automatically. So we don't need to pay lot of attention of these mistakes.

### 4. Keep Practicing

The next important step is to do the practice. It needs to implementing the Python concepts through the code. We should be consistence to our daily coding practice.

Consistency is the key of success in any aspect of life not only in programming. Writing code daily will help to develop muscle memory.

We can do the problem exercise of related concepts or solve at least 2 or 3 problems of Python. It may seem hard but muscle memory plays large part in programing. It will take us ahead from those who believe only the reading concept of Python is sufficient.

### 5. Make Notes as Needed

Creating notes by own is an excellent method to learn the concepts and syntax of Python. It will establish stability and focus that helps you become a Python developer. Make brief and concise notes with relevant information and include appropriate examples of the subject concerned.

Maintain own notes are also helped to learn fast. A study published in Psychological Science that -

The students who were taking longhand notes in the studies were forced to be more selective — because you can't write as fast as you can type.

### 6. Discuss Concepts with Other

Coding seems to be solitary activity, but we can enhance our skills by interacting with the others. We should discuss our doubts to the expert or friends who are learning Python. This habit will help to get additional information, tips and tricks, and solution of coding problems. One of the best advantages of Python, it has a great community. Therefore, we can also learn from passionate Python enthusiasts.

### 7. Do small Projects

After understanding Python's basic concept, a beginner should try to work on small projects. It will help to understand Python more deeply and become more component in it. Theoretical knowledge is not enough to get command over the Python language. These projects can be anything as long as they teach you something. You can start with the small projects such as calculator app, a tic-toc-toe game, an alarm clock app, a to-do list, student or customer management system, etc.

Once you get handy with a small project, you can easily shift toward your interesting domain (Machine Learning, Web Development, etc.).

### 8. Teach Others

There is a famous saying that **"If you want to learn something then you should teach other"**. It is also true in case of learning Python. Share your information to other students via creating blog posts, recording videos or taking classes in local training center. It will help us to enhance the understanding of Python and explore the unseen loopholes in your knowledge. If you don't want to do all these, join the online forum and post your answers on Python related questions.

### 9. Explore Libraries and Frameworks

Python consists of vast libraries and various frameworks. After getting familiar with Python's basic concepts, the next step is to explore the Python libraries. Libraries are essential to work with the domain specific projects. In the following section, we describe the brief introduction of the main libraries.

* **TensorFlow -** It is an artificial intelligence library which allows us to create large scale AI based projects.
* **Django -** It is an open source framework that allows us to develop web applications. It is easy, flexible, and simple to manage.
* **Flask -** It is also an open source web framework. It is used to develop lightweight web applications.
* **Pandas -** It is a Python library which is used to perform scientific computations.
* **Keras -** It is an open source library, which is used to work around the neural network.

There are many libraries in Python. Above, we have mentioned a few of them.

### 10. Contribute to Open Source

As we know, Python is an open source language that means it is freely available for everyone. We can also contribute to Python online community to enhance our knowledge. Contributing to open source projects is the best way to explore own knowledge. We also receive the feedback, comments or suggestions for work that we submitted. The feedback will enable the best practices for Python programming and help us to become a good Python developer.

## Usage of Python

Python is a general purpose, open source, high-level programming language and also provides number of libraries and frameworks. Python has gained popularity because of its simplicity, easy syntax and user-friendly environment. The usage of Python as follows.

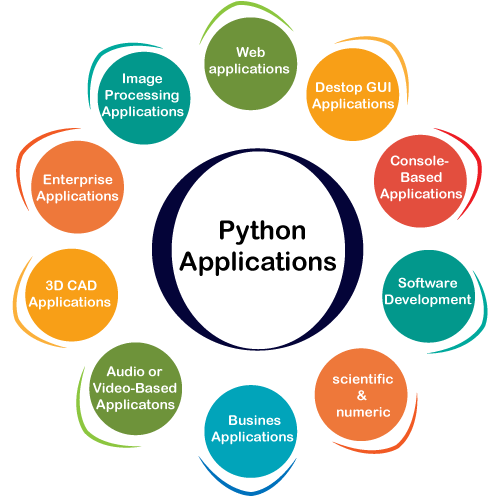
* Desktop Applications
* Web Applications
* Data Science
* Artificial Intelligence
* Machine Learning
* Scientific Computing
* Robotics
* Internet of Things (IoT)
* Gaming
* Mobile Apps
* Data Analysis and Preprocessing

In the next topic, we will discuss the Python Application, where we have defined Python's usage in detail

**Python Applications**

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied.



**1) Web Applications**

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on **Instagram**. Python provides many useful frameworks, and these are given below:

* Django and Pyramid framework(Use for heavy applications)
* Flask and Bottle (Micro-framework)
* Plone and Django CMS (Advance Content management)

**2) Desktop GUI Applications**

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface. Some popular GUI libraries are given below.

* Tkinter or Tk
* wxWidgetM
* Kivy (used for writing multitouch applications )
* PyQt or Pyside

**3) Console-based Application**

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively. It is famous for having REPL, which means **the Read-Eval-Print Loop** that makes it the most suitable language for the command-line applications.

Python provides many free library or module which helps to build the command-line apps. The necessary **IO** libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

**4) Software Development**

Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

* **SCons** is used to build control.
* **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.
* **Round** or **Trac** for bug tracking and project management.

**5) Scientific and Numeric**

This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Implementing machine learning algorithms require complex mathematical calculation. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc. If you have some basic knowledge of Python, you need to import libraries on the top of the code. Few popular frameworks of machine libraries are given below.

* SciPy
* Scikit-learn
* NumPy
* Pandas
* Matplotlib

**6) Business Applications**

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a **Tryton** platform which is used to develop the business application.

**7) Audio or Video-based Applications**

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer, cplay,** etc. The few multimedia libraries are given below.

* Gstreamer
* Pyglet
* QT Phonon

**8) 3D CAD Applications**

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

* Fandango (Popular )
* CAMVOX
* HeeksCNC
* AnyCAD
* RCAM

**9) Enterprise Applications**

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

**10) Image Processing Application**

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

* OpenCV
* Pillow
* SimpleITK

In this topic, we have described all types of applications where Python plays an essential role in the development of these applications. In the next tutorial, we will learn more concepts about Python.

**How to Install Python (Environment Set-up)**

In order to become Python developer, the first step is to learn how to install or update Python on a local machine or computer. In this tutorial, we will discuss the installation of Python on various operating systems.

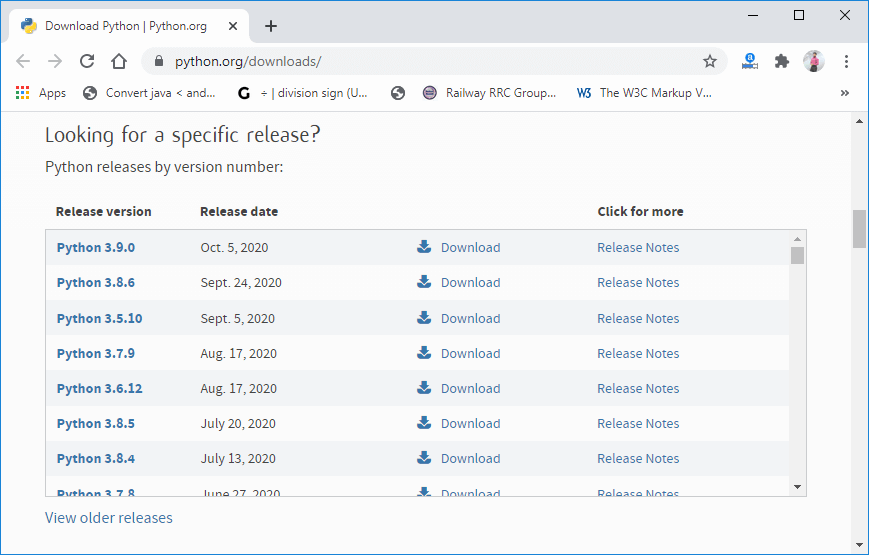
**Installation on Windows**

Visit the link *https://www.python.org/downloads/* to download the latest release of Python. In this process, we will install Python 3.8.6 on our Windows operating system. When we click on the above link, it will bring us the following page.

**Step - 1: Select the Python's version to download.**

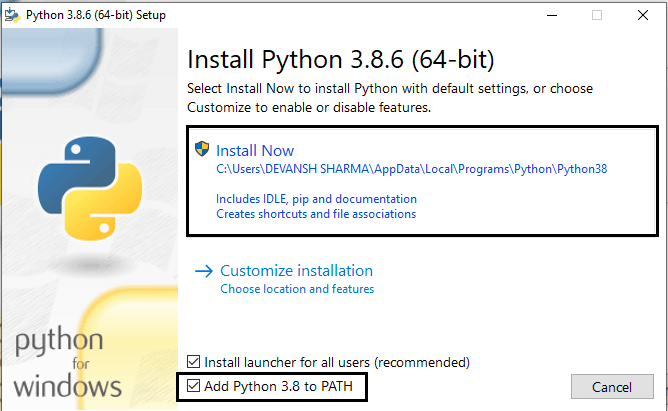
Click on the download button.

x



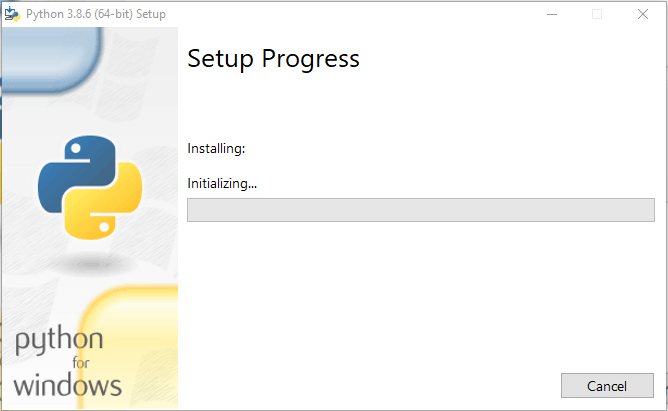
**Step - 2: Click on the Install Now**

Double-click the executable file, which is downloaded; the following window will open. Select Customize installation and proceed. Click on the Add Path check box, it will set the Python path automatically.

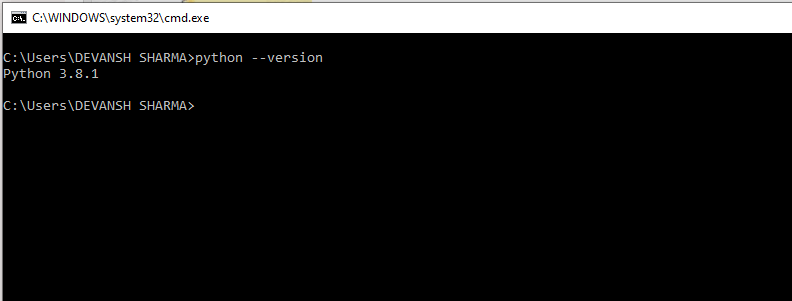


We can also click on the customize installation to choose desired location and features. Other important thing is install launcher for the all user must be checked.

**Step - 3 Installation in Process**



Now, try to run python on the command prompt. Type the command python -version in case of python3.



We are ready to work with the Python.

**How to install Python on Ubuntu 16.04 LTS?**

**Introduction**

Python is a high level, dynamic and general purpose programming language. It was designed and developed by Python Software Foundation.

Python is a cross-platform language and was first released on 20, February 1991.

In this tutorial, we are working on the installation process of Python that includes the following steps.

Although Python 2.7 is installed by default in Ubuntu, but we can install other Python versions like python3.

To install python3, just type the following command in the terminal and it will be installed automatically.

**Python Installation**

1) **Update the APT Repository**

1. $ apt-get update

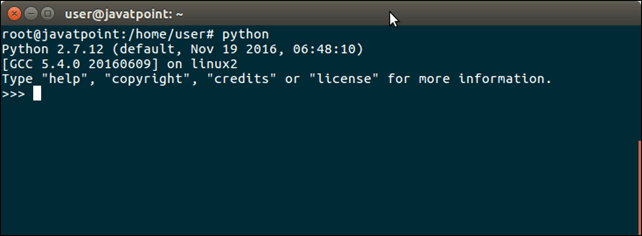
2) **Install Python**

1. $ apt-get install python3.6

2) **Verify Python**

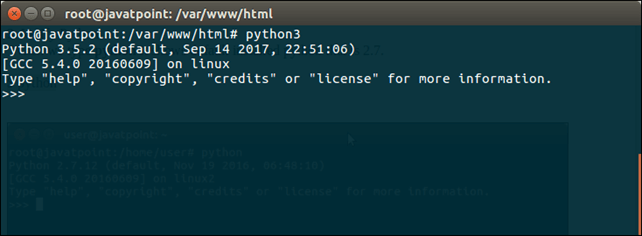
When we type **python** it shows default installed python that is 2.7.

1. $ python



For Python3 type the following command, then it will show the other version as well.

1. $ python3



Well, on the basis of these commands, we can test application for both Python versions.

**First Python Program**

In this Section, we will discuss the basic syntax of Python, we will run a simple program to print **Hello World** on the console.

Python provides us the two ways to run a program:

* Using Interactive interpreter prompt
* Using a script file

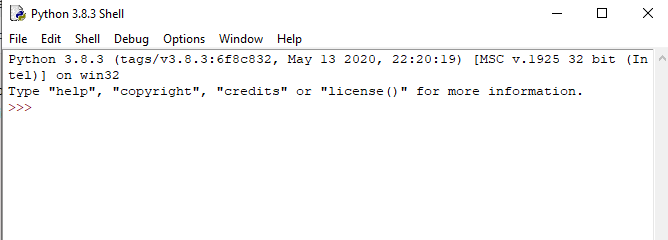
Let's discuss each one of them in detail.

**Interactive interpreter prompt**

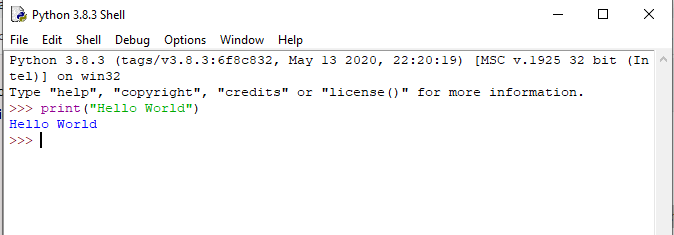
Python provides us the feature to execute the Python statement one by one at the interactive prompt. It is preferable in the case where we are concerned about the output of each line of our Python program.

To open the interactive mode, open the terminal (or command prompt) and type python (python3 in case if you have Python2 and Python3 both installed on your system).

It will open the following prompt where we can execute the Python statement and check their impact on the console.



After writing the print statement, press the **Enter** key.



Here, we get the message **"Hello World !"** printed on the console.

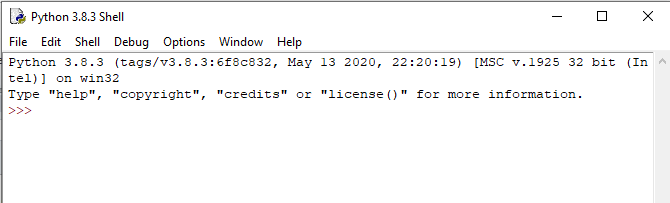
**Using a script file (Script Mode Programming)**

The interpreter prompt is best to run the single-line statements of the code. However, we cannot write the code every-time on the terminal. It is not suitable to write multiple lines of code.

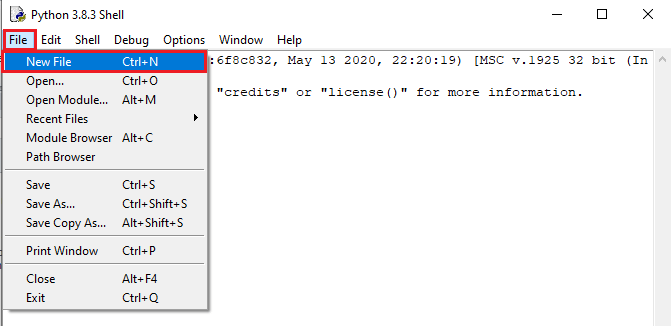
Using the script mode, we can write multiple lines code into a file which can be executed later. For this purpose, we need to open an editor like notepad, create a file named and save it with **.py** extension, which stands for **"Python".** Now, we will implement the above example using the script mode.

1. print ("hello world"); #here, we have used print() function to print the message on the console.

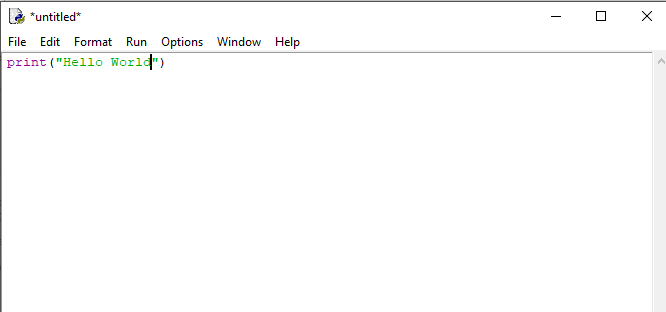
To run this file named as first.py, we need to run the following command on the terminal.



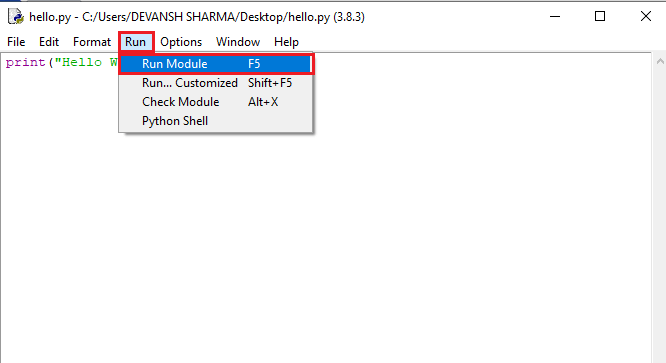
**Step - 1:** Open the Python interactive shell, and click **"File"** then choose **"New",** it will open a new blank script in which we can write our code.



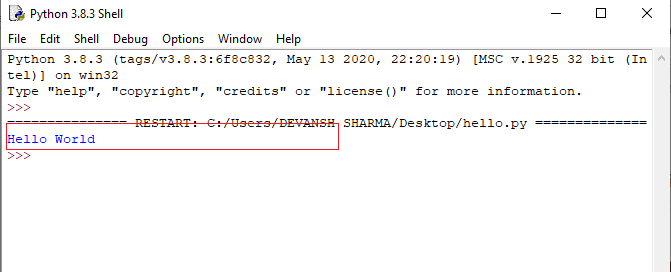
**Step -2:** Now, write the code and press **"Ctrl+S"** to save the file.



**Step - 3:** After saving the code, we can run it by clicking "Run" or "Run Module". It will display the output to the shell.

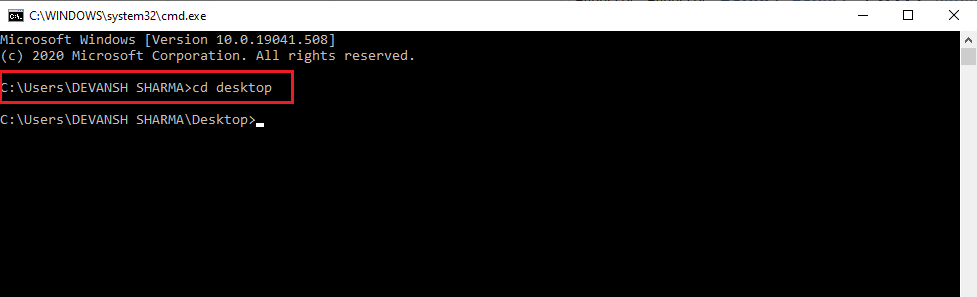


The output will be shown as follows.

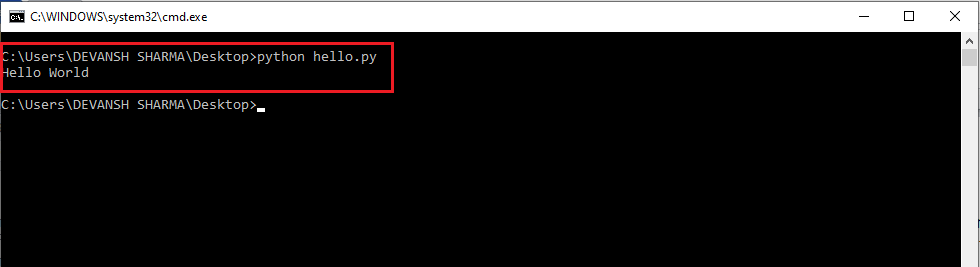


**Step - 4:** Apart from that, we can also run the file using the operating system terminal. But, we should be aware of the path of the directory where we have saved our file.

* Open the command line prompt and navigate to the directory.



* We need to type the **python** keyword, followed by the file name and hit enter to run the Python file.



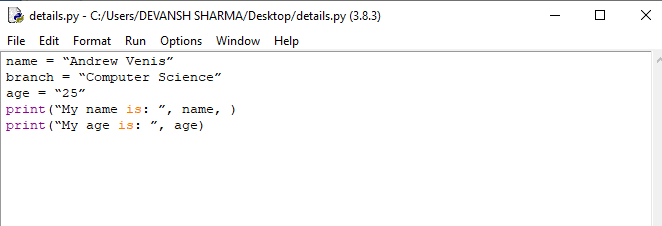
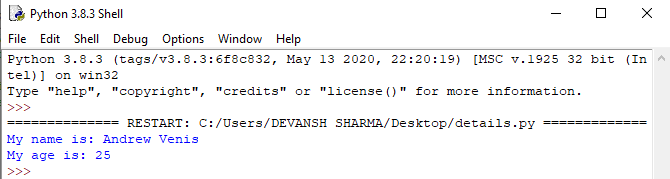
**Multi-line Statements**

Multi-line statements are written into the notepad like an editor and saved it with **.py** extension. In the following example, we have defined the execution of the multiple code lines using the Python script.

**Code:**

1. name = "Andrew Venis"
2. branch = "Computer Science"
3. age = "25"
4. print("My name is: ", name, )
5. print("My age is: ", age)

**Script File:**

**Pros and Cons of Script Mode**

The script mode has few advantages and disadvantages as well. Let's understand the following advantages of running code in script mode.

* We can run multiple lines of code.
* Debugging is easy in script mode.
* It is appropriate for beginners and also for experts.

Let's see the disadvantages of the script mode.

* We have to save the code every time if we make any change in the code.
* It can be tedious when we run a single or a few lines of code.

**Get Started with PyCharm**

In our first program, we have used gedit on our CentOS as an editor. On Windows, we have an alternative like notepad or notepad++ to edit the code. However, these editors are not used as IDE for python since they are unable to show the syntax related suggestions.

JetBrains provides the most popular and a widely used cross-platform IDE **PyCharm** to run the python programs.

**PyCharm installation**

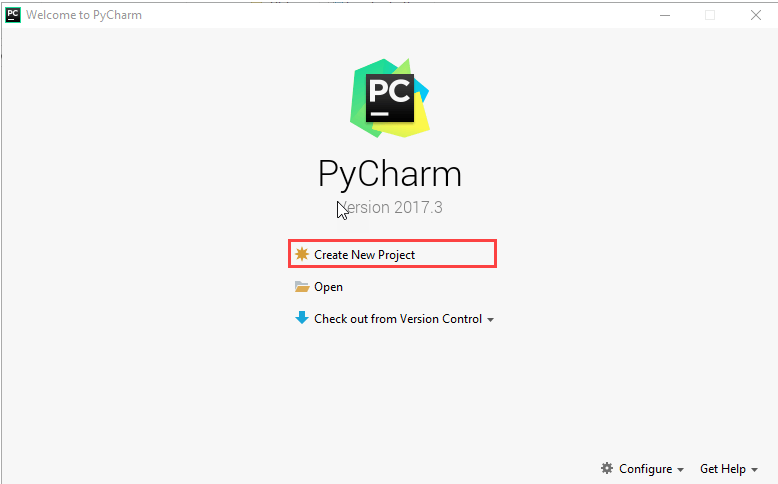
As we have already stated, PyCharm is a cross-platform IDE, and hence it can be installed on a variety of the operating systems. In this section of the tutorial, we will cover the installation process of PyCharm on Windows, MacOS, CentOS, and Ubuntu.

**Windows**

Installing PyCharm on Windows is very simple. To install PyCharm on Windows operating system, visit the link https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows to download the executable installer. **Double click** the installer (.exe) file and install PyCharm by clicking next at each step.

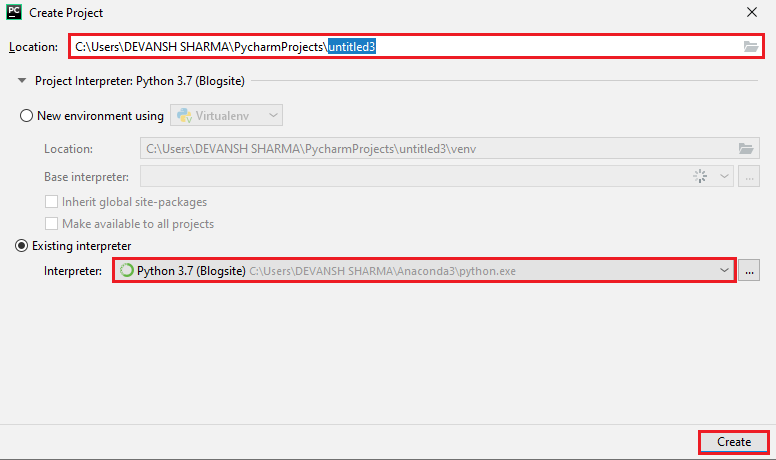
To create a first program to Pycharm follows the following step.

**Step - 1.** Open Pycharm editor. Click on "Create New Project" option to create new project.

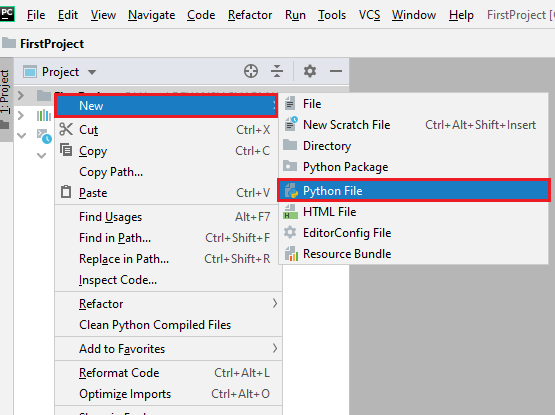


**Step - 2.** Select a location to save the project.

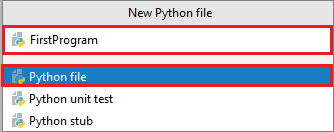
1. We can save the newly created project at desired memory location or can keep file location as it is but atleast change the project default name **untitled** to **"FirstProject"** or something meaningful.
2. Pycharm automatically found the installed Python interpreter.
3. After change the name click on the "Create" Button.



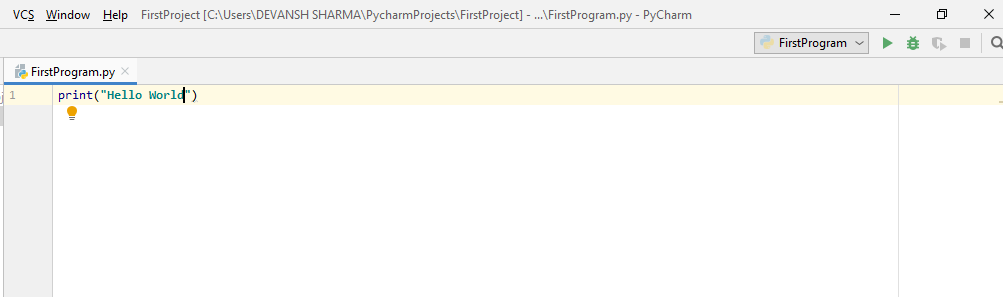
**Step - 3.** Click on "**File"** menu and select **"New"**. By clicking "New" option it will show various file formats. Select the "Python File".



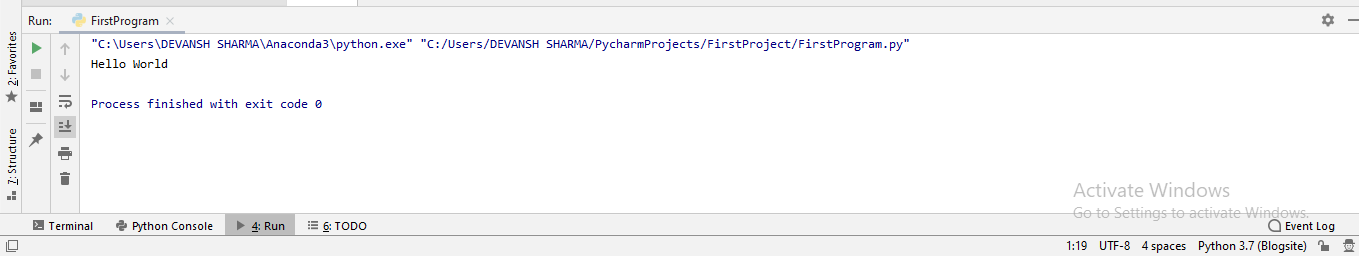
**Step - 4.** Now type the name of the Python file and click on "OK". We have written the "FirstProgram".



**Step - 5.** Now type the first program - print("Hello World") then click on the "Run" menu to run program.



**Step - 6.** The output will appear at the bottom of the screen.

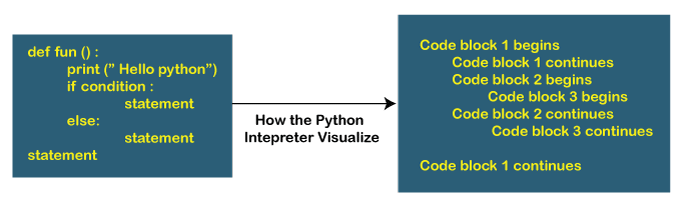


**Basic Syntax of Python**

**Indentation and Comment in Python**

Indentation is the most significant concept of the Python programming language. Improper use of indentation will end up **"IndentationError"** in our code.

Indentation is nothing but adding whitespaces before the statement when it is needed. Without indentation Python doesn't know which statement to be executed to next. Indentation also defines which statements belong to which block. If there is no indentation or improper indentation, it will display "**IndentationError"** and interrupt our code.



Python indentation defines the particular group of statements belongs to the particular block. The programming languages such as C, C++, java use the curly braces {} to define code blocks.

In Python, statements that are the same level to the right belong to the same block. We can use four whitespaces to define indentation. Let's see the following lines of code.

**Example -**

1. list1 = [1, 2, 3, 4, 5]
2. for i in list1:
3. print(i)
4. if i==4:
5. break
6. print("End of for loop")

**Output:**

1

2

3

4

End of for loop

**Explanation:**

In the above code, for loop has a code blocks and if the statement has its code block inside for loop. Both indented with four whitespaces. The last **print()** statement is not indented; that's means it doesn't belong to for loop.

**Comments in Python**

Comments are essential for defining the code and help us and other to understand the code. By looking the comment, we can easily understand the intention of every line that we have written in code. We can also find the error very easily, fix them, and use in other applications.

In Python, we can apply comments using the # hash character. The Python interpreter entirely ignores the lines followed by a hash character. A good programmer always uses the comments to make code under stable. Let's see the following example of a comment.

1. name  = "Thomas"   # Assigning string value to the name variable

We can add comment in each line of the Python code.

1. Fees = 10000      # defining course fees is 10000
2. Fees = 20000      # defining course fees is 20000

It is good idea to add code in any line of the code section of code whose purpose is not obvious. This is a best practice to learn while doing the coding.

**Types of Comment**

Python provides the facility to write comments in two ways- single line comment and multi-line comment.

**Single-Line Comment -** Single-Line comment starts with the hash # character followed by text for further explanation.

1. # defining the marks of a student
2. Marks = 90

We can also write a comment next to a code statement. Consider the following example.

1. Name = "James"   # the name of a student is James
2. Marks = 90            # defining student's marks
3. Branch = "Computer Science"   # defining student branch

**Multi-Line Comments -** Python doesn't have explicit support for multi-line comments but we can use hash # character to the multiple lines. **For example -**

1. # we are defining for loop
2. # To iterate the given list.
3. # run this code.

We can also use another way.

1. " " "
2. This is an example
3. Of multi-line comment
4. Using triple-quotes
5. " " "

This is the basic introduction of the comments. Visit our **Python Comment** tutorial to learn it in detail.

**Python Identifiers**

Python identifiers refer to a name used to identify a variable, function, module, class, module or other objects. There are few rules to follow while naming the Python Variable.

* A variable name must start with either an English letter or underscore (\_).
* A variable name cannot start with the number.
* Special characters are not allowed in the variable name.
* The variable's name is case sensitive.

Let's understand the following example.

**Example -**

1. number = 10
2. print(num)
4. \_a = 100
5. print(\_a)
7. x\_y = 1000
8. print(x\_y)

**Output:**

10

100

1000

We have defined the basic syntax of the Python programming language. We must be familiar with the core concept of any programming languages. Once we memorize the concepts as mentioned above. The journey of learning Python will become easier.

**How to Install PyCharm on Ubuntu 16.04 LTS?**

**Introduction**

PyCharm is an IDE (Integrated Development Environment) which is used to write code and develop applications. Specifically, It is designed for Python programming and supports web development using Django.

It was designed and developed by JetBrains Inc and initially released on July, 2010.

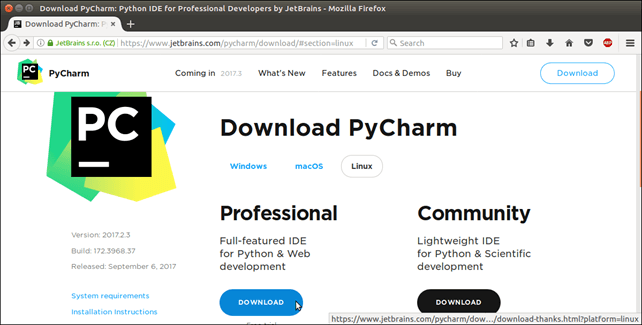
**Prerequisites**

* Ubuntu 64 bit
* KDE, GNOME or Unity DE desktop
* Python 2.4 or higher, Jython, PyPy or IronPython

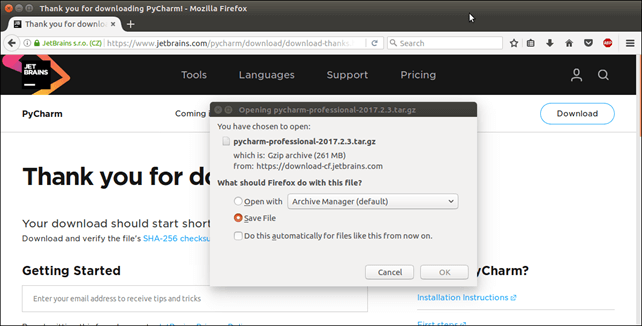
**PyCharm Installation**

1) **Download Pycharm Archive**

Visit official site <https://www.jetbrains.com> and download archive file.

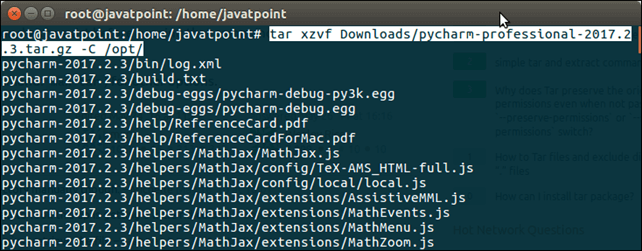


**Save archive**



2) **Move and Extract**

1. $ tar xzvf Downloads/pycharm-professional-2017.2.3.tar.gz -C /opt/

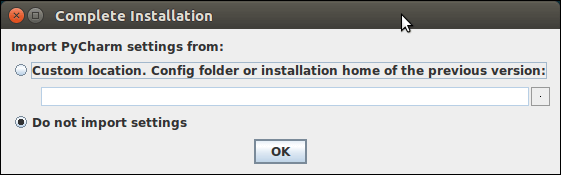


3) **Install PyCharm**

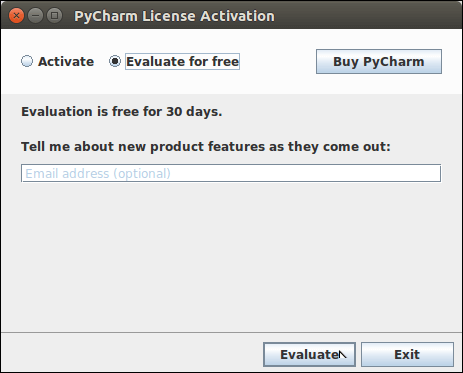
1. $ sudo /opt/pycharm-2017.2.3/bin/pycharm.sh

This command will open installation pop ups that are showing below.

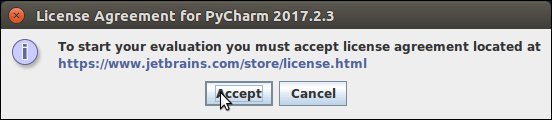
**Import settings if any**



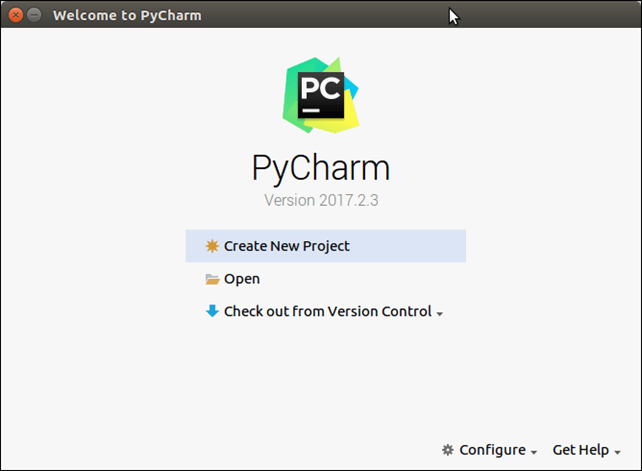
**Select trial version**



**Accept license agreement**



After that PyCharm welcome page displays that look like below.



Well, PyCharm has installed successfully. Now, we can create new python projects and develop applications.

Happy Coding!

**Python Variables**

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

**Identifier Naming**

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

* The first character of the variable must be an alphabet or underscore ( \_ ).
* All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
* Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Identifier name must not be similar to any keyword defined in the language.
* Identifier names are case sensitive; for example, my name, and MyName is not the same.
* Examples of valid identifiers: a123, \_n, n\_9, etc.
* Examples of invalid identifiers: 1a, n%4, n 9, etc.

**Declaring Variable and Assigning Values**

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

**Object References**

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

1. print("John")

**Output:**

John

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type()** function.

1. type("John")

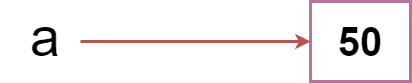
**Output:**

<class 'str'>

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

1. a = 50



In the above image, the variable **a** refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

a = 50

b = a

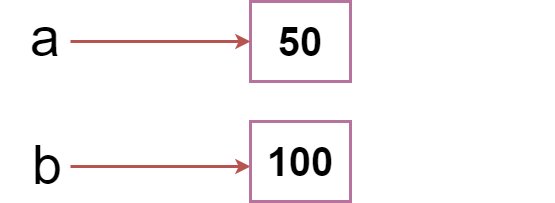


The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

a = 50

b =100



Python manages memory efficiently if we assign the same variable to two different values.

**Object Identity**

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id()** function, is used to identify the object identifier. Consider the following example.

1. a = 50
2. b = a
3. print(id(a))
4. print(id(b))
5. # Reassigned variable a
6. a = 500
7. print(id(a))

**Output:**

140734982691168

140734982691168

2822056960944

We assigned the **b = a, a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to 500; then it referred to the new object identifier.

**Variable Names**

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(\_). Consider the following example of valid variables names.

1. name = "Devansh"
2. age = 20
3. marks = 80.50
5. print(name)
6. print(age)
7. print(marks)

**Output:**

Devansh

20

80.5

Consider the following valid variables name.

1. name = "A"
2. Name = "B"
3. naMe = "C"
4. NAME = "D"
5. n\_a\_m\_e = "E"
6. \_name = "F"
7. name\_ = "G"
8. \_name\_ = "H"
9. na56me = "I"
11. print(name,Name,naMe,NAME,n\_a\_m\_e, NAME, n\_a\_m\_e, \_name, name\_,\_name, na56me)

**Output:**

A B C D E D E F G F I

In the above example, we have declared a few valid variable names such as name, \_name\_ , etc. But it is not recommended because when we try to read code, it may create confusion. The variable name should be descriptive to make code more readable.

The multi-word keywords can be created by the following method.

* **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
* **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
* **Snake Case -** In the snake case, Words are separated by the underscore. For example - name\_of\_student, etc.

**Multiple Assignment**

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

**1. Assigning single value to multiple variables**

**Eg:**

1. x=y=z=50
2. print(x)
3. print(y)
4. print(z)

**Output:**

50

50

50

**2. Assigning multiple values to multiple variables:**

**Eg:**

1. a,b,c=5,10,15
2. print a
3. print b
4. print c

**Output:**

5

10

15

The values will be assigned in the order in which variables appear.

**Python Variable Types**

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

**Local Variable**

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

**Example -**

1. # Declaring a function
2. def add():
3. # Defining local variables. They has scope only within a function
4. a = 20
5. b = 30
6. c = a + b
7. print("The sum is:", c)
9. # Calling a function
10. add()

**Output:**

The sum is: 50

**Explanation:**

In the above code, we declared a function named **add()** and assigned a few variables within the function. These variables will be referred to as the **local variables** which have scope only inside the function. If we try to use them outside the function, we get a following error.

1. add()
2. # Accessing local variable outside the function
3. print(a)

**Output:**

The sum is: 50

print(a)

NameError: name 'a' is not defined

We tried to use local variable outside their scope; it threw the **NameError.**

**Global Variables**

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function. If we don't use the **global** keyword, the function treats it as a local variable. Let's understand the following example.

**Example -**

1. # Declare a variable and initialize it
2. x = 101
4. # Global variable in function
5. def mainFunction():
6. # printing a global variable
7. global x
8. print(x)
9. # modifying a global variable
10. x = 'Welcome To Javatpoint'
11. print(x)
13. mainFunction()
14. print(x)

**Output:**

101

Welcome To Javatpoint

Welcome To Javatpoint

**Explanation:**

In the above code, we declare a global variable **x** and assign a value to it. Next, we defined a function and accessed the declared variable using the **global** keyword inside the function. Now we can modify its value. Then, we assigned a new string value to the variable x.

Now, we called the function and proceeded to print **x**. It printed the as newly assigned value of x.

**Delete a variable**

We can delete the variable using the **del** keyword. The syntax is given below.

**Syntax -**

1. del <variable\_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

**Example -**

1. # Assigning a value to x
2. x = 6
3. print(x)
4. # deleting a variable.
5. del x
6. print(x)

**Output:**

6

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 389, in

print(x)

NameError: name 'x' is not defined

**Maximum Possible Value of an Integer in Python**

Unlike the other programming languages, Python doesn't have long int or float data types. It treats all integer values as an **int** data type. Here, the question arises. What is the maximum possible value can hold by the variable in Python? Consider the following example.

**Example -**

1. # A Python program to display that we can store
2. # large numbers in Python
4. a = 10000000000000000000000000000000000000000000
5. a = a + 1
6. print(type(a))
7. print (a)

**Output:**

<class 'int'>

10000000000000000000000000000000000000000001

As we can see in the above example, we assigned a large integer value to variable **x** and checked its type. It printed **class <int>** not long int. Hence, there is no limitation number by bits and we can expand to the limit of our memory.

Python doesn't have any special data type to store larger numbers.

**Print Single and Multiple Variables in Python**

We can print multiple variables within the single print statement. Below are the example of single and multiple printing values.

**Example - 1 (Printing Single Variable)**

1. # printing single value
2. a = 5
3. print(a)
4. print((a))

**Output:**

5

5

**Example - 2 (Printing Multiple Variables)**

1. a = 5
2. b = 6
3. # printing multiple variables
4. print(a,b)
5. # separate the variables by the comma
6. Print(1, 2, 3, 4, 5, 6, 7, 8)

**Output:**

5 6

1 2 3 4 5 6 7 8

**Basic Fundamentals:**

This section contains the fundamentals of Python, such as:

**i)Tokens and their types.**

**ii) Comments**

**a)Tokens:**

* The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
* The token is the smallest unit inside the given program.

There are following tokens in Python:

* Keywords.
* Identifiers.
* Literals.
* Operators.

We will discuss above the tokens in detail next tutorials.

**Python Data Types**

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

1. a = 5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

1. a=10
2. b="Hi Python"
3. c = 10.5
4. print(type(a))
5. print(type(b))
6. print(type(c))

**Output:**

<type 'int'>

<type 'str'>

<type 'float'>

**Standard data types**

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. [Numbers](https://www.javatpoint.com/python-data-types#numbers)
2. [Sequence Type](https://www.javatpoint.com/python-data-types#SequenceType)
3. [Boolean](https://www.javatpoint.com/python-data-types#Boolean)
4. [Set](https://www.javatpoint.com/python-data-types#Set)
5. [Dictionary](https://www.javatpoint.com/python-data-types#dictionary)



In this section of the tutorial, we will give a brief introduction of the above data-types. We will discuss each one of them in detail later in this tutorial.

**Numbers**

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable. For example;

1. a = 5
2. print("The type of a", type(a))
4. b = 40.5
5. print("The type of b", type(b))
7. c = 1+3j
8. print("The type of c", type(c))
9. print(" c is a complex number", isinstance(1+3j,complex))

**Output:**

The type of a <class 'int'>

The type of b <class 'float'>

The type of c <class 'complex'>

c is complex number: True

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

**Sequence Type**

**String**

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator \* is known as a repetition operator as the operation "Python" \*2 returns 'Python Python'.

The following example illustrates the string in Python.

**Example - 1**

1. str = "string using double quotes"
2. print(str)
3. s = '''''A multiline
4. string'''
5. print(s)

**Output:**

string using double quotes

A multiline

string

Consider the following example of string handling.

**Example - 2**

1. str1 = 'hello javatpoint' #string str1
2. str2 = ' how are you' #string str2
3. print (str1[0:2]) #printing first two character using slice operator
4. print (str1[4]) #printing 4th character of the string
5. print (str1\*2) #printing the string twice
6. print (str1 + str2) #printing the concatenation of str1 and str2

**Output:**

he

o

hello javatpointhello javatpoint

hello javatpoint how are you

**List**

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

Consider the following example.

1. list1  = [1, "hi", "Python", 2]
2. #Checking type of given list
3. print(type(list1))
5. #Printing the list1
6. print (list1)
8. # List slicing
9. print (list1[3:])
11. # List slicing
12. print (list1[0:2])
14. # List Concatenation using + operator
15. print (list1 + list1)
17. # List repetation using \* operator
18. print (list1 \* 3)

**Output:**

[1, 'hi', 'Python', 2]

[2]

[1, 'hi']

[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]

[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]

**Tuple**

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

1. tup  = ("hi", "Python", 2)
2. # Checking type of tup
3. print (type(tup))
5. #Printing the tuple
6. print (tup)
8. # Tuple slicing
9. print (tup[1:])
10. print (tup[0:1])
12. # Tuple concatenation using + operator
13. print (tup + tup)
15. # Tuple repatation using \* operator
16. print (tup \* 3)
18. # Adding value to tup. It will throw an error.
19. t[2] = "hi"

**Output:**

<class 'tuple'>

('hi', 'Python', 2)

('Python', 2)

('hi',)

('hi', 'Python', 2, 'hi', 'Python', 2)

('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):

File "main.py", line 14, in <module>

t[2] = "hi";

TypeError: 'tuple' object does not support item assignment

**Dictionary**

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

1. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
3. # Printing dictionary
4. print (d)
6. # Accesing value using keys
7. print("1st name is "+d[1])
8. print("2nd name is "+ d[4])
10. print (d.keys())
11. print (d.values())

**Output:**

1st name is Jimmy

2nd name is mike

{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}

dict\_keys([1, 2, 3, 4])

dict\_values(['Jimmy', 'Alex', 'john', 'mike'])

**Boolean**

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

1. # Python program to check the boolean type
2. print(type(True))
3. print(type(False))
4. print(false)

**Output:**

<class 'bool'>

<class 'bool'>

NameError: name 'false' is not defined

**Set**

Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values. Consider the following example.

1. # Creating Empty set
2. set1 = set()
4. set2 = {'James', 2, 3,'Python'}
6. #Printing Set value
7. print(set2)
9. # Adding element to the set
11. set2.add(10)
12. print(set2)
14. #Removing element from the set
15. set2.remove(2)
16. print(set2)

**Output:**

{3, 'Python', 'James', 2}

{'Python', 'James', 3, 2, 10}

{'Python', 'James', 3, 10}

**Python Keywords**

Every scripting language has designated words or keywords, with particular definitions and usage guidelines. Python is no exception. The fundamental constituent elements of any Python program are Python keywords.

This tutorial will give you a basic overview of all Python keywords and a detailed discussion of some important keywords that are frequently used.

**Introducing Python Keywords**

Python keywords are unique words reserved with defined meanings and functions that we can only apply for those functions. You'll never need to import any keyword into your program because they're permanently present.

Python's built-in methods and classes are not the same as the keywords. Built-in methods and classes are constantly present; however, they are not as limited in their application as keywords.

Assigning a particular meaning to Python keywords means you can't use them for other purposes in our code. You'll get a message of SyntaxError if you attempt to do the same. If you attempt to assign anything to a built-in method or type, you will not receive a SyntaxError message; however, it is still not a smart idea.

Python contains thirty-five keywords in the most recent version, i.e., Python 3.8. Here we have shown a complete list of Python keywords for the reader's reference.

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

In distinct versions of Python, the preceding keywords might be changed. Some extras may be introduced, while others may be deleted. By writing the following statement into the coding window, you can anytime retrieve the collection of keywords in the version you are working on.

**Code**

1. # Python program to demonstrate the application of iskeyword()
2. # importing keyword library which has lists
3. import keyword
5. # displaying the complete list using "kwlist()."
6. print("The set of keywords in this version is: ")
7. print( keyword.kwlist )

**Output:**

The set of keywords in this version is :

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

By calling help(), you can retrieve a list of currently offered keywords:

**Code**

1. help("keywords")

**How to Identify Python Keywords**

Python's keyword collection has evolved as new versions were introduced. The await and async keywords, for instance, were not introduced till Python 3.7. Also, in Python 2.7, the words print and exec constituted keywords; however, in Python 3+, they were changed into built-in methods and are no longer part of the set of keywords. In the paragraphs below, you'll discover numerous methods for determining whether a particular word in Python is a keyword or not.

**Write Code on a Syntax Highlighting IDE**

There are plenty of excellent Python IDEs available. They'll all highlight keywords to set them apart from the rest of the terms in the code. This facility will assist you in immediately identifying Python keywords during coding so that you do not misuse them.

**Verify Keywords with Script in a REPL**

There are several ways to detect acceptable Python keywords plus know further regarding them in the Python REPL.

**Look for a SyntaxError**

Lastly, if you receive a SyntaxError when attempting to allocate to it, name a method with it, or do anything else with that, and it isn't permitted, it's probably a keyword. This one is somewhat more difficult to see, but it is still a technique for Python to tell you if you're misusing a keyword.

**Python Keywords and Their Usage**

The following sections categorize Python keywords under the headings based on their frequency of use. The first category, for instance, includes all keywords utilized as values, whereas the next group includes keywords employed as operators. These classifications will aid in understanding how keywords are employed and will assist you in arranging the huge collection of Python keywords.

* A few terms mentioned in the segment following may be unfamiliar to you. They're explained here, and you must understand what they mean before moving on:
* The Boolean assessment of a variable is referred to as truthfulness. A value's truthfulness reveals if the value of the variable is true or false.

In the Boolean paradigm, truth refers to any variable that evaluates to true. Pass an item as an input to bool() to see if it is true. If True is returned, the value of the item is true. Strings and lists which are not empty, non-zero numbers, and many other objects are illustrations of true values.

False refers to any item in a Boolean expression that returns false. Pass an item as an input to bool() to see if it is false. If False is returned, the value of the item is false. Examples of false values are " ", 0, { }, and [ ].

**Value Keywords: True, False, None**

Three Python keywords are employed as values in this example. These are singular values, which we can reuse indefinitely and every time correspond to the same entity. These values will most probably be seen and used frequently.

**The Keywords True and False**

These keywords are typed in lowercase in conventional computer languages (true and false); however, they are typed in uppercase in Python every time. In Python script, the True Python keyword represents the Boolean true state. False is a keyword equivalent to True, except it has the negative Boolean state of false.

True and False are those keywords that can be allocated to variables or parameters and are compared directly.

**Code**

1. print( 4 == 4 )
2. print( 6 > 9 )
3. print( True or False )
4. print( 9 <= 28 )
5. print( 6 > 9 )
6. print( True and False )

**Output:**

True

False

True

True

False

False

Because the first, third, and fourth statements are true, the interpreter gives True for those and False for other statements. True and False are the equivalent in Python as 1 & 0. We can use the accompanying illustration to support this claim:

**Code**

1. print( True == 3 )
2. print( False == 0 )
3. print( True + True + True)

**Output:**

False

True

3

**The None Keyword**

None is a Python keyword that means "nothing." None is known as nil, null, or undefined in different computer languages.

If a function does not have a return clause, it will give None as the default output:

**Code**

1. print( None == 0 )
2. print( None == " " )
3. print( None == False )
4. A = None
5. B = None
6. print( A == B )

**Output:**

False

False

False

True

If a no\_return\_function returns nothing, it will simply return a None value. None is delivered by functions that do not meet a return expression in the program flow. Consider the following scenario:

**Code**

1. def no\_return\_function():
2. num1 = 10
3. num2 = 20
4. addition = num1 + num2
6. number = no\_return\_function()
7. print( number )

**Output:**

None

This program has a function with\_return that performs multiple operations and contains a return expression. As a result, if we display a number, we get None, which is given by default when there is no return statement. Here's an example showing this:

**Code**

1. def with\_return( num ):
2. if num % 4 == 0:
3. return False
5. number = with\_return( 67 )
6. print( number )

**Output:**

None

**Operator Keywords: and, or, not, in, is**

Several Python keywords are employed as operators to perform mathematical operations. In many other computer languages, these operators are represented by characters such as &, |, and!. All of these are keyword operations in Python:

|  |  |  |
| --- | --- | --- |
| **Mathematical Operations** | **Operations in Other Languages** | **Python Keyword** |
| **AND, ∧** | && | and |
| **OR, ∨** | || | or |
| **NOT, ¬** | ! | not |
| **CONTAINS, ∈** |  | in |
| **IDENTITY** | === | is |

Writers created Python programming with clarity in mind. As a result, many operators in other computer languages that employ characters in Python are English words called keywords.

**The and Keyword**

The Python keyword and determines whether both the left-hand side and right-hand side operands and are true or false. The outcome will be True if both components are true. If one is false, the outcome will also be False:

|  |  |  |
| --- | --- | --- |
| **Truth table for and** | | |
| **X** | **Y** | **X and Y** |
| True | True | True |
| False | True | False |
| True | False | False |
| False | False | False |

1. <component1> and <component2>

It's worth noting that the outcomes of an and statement aren't always True or False. Due to and's peculiar behavior, this is the case. Instead of processing the inputs to corresponding Boolean values, it just gives <component1> if it is false or <component2> if it is true. The outputs of a and expression could be utilized with a conditional if clause or provided to bool() to acquire an obvious True or False answer.

**The or Keyword**

The or keyword in Python is utilized to check if, at minimum, 1 of the inputs is true. If the first argument is true, the or operation yields it; otherwise, the second argument is returned:

1. <component1> or <component2>

Similarly to the and keyword, the or keyword does not change its inputs to corresponding Boolean values. Instead, the outcomes are determined based on whether they are true or false.

|  |  |  |
| --- | --- | --- |
| **Truth table for or** | | |
| **X** | **Y** | **X or Y** |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

**The not Keyword**

The not keyword in Python is utilized to acquire a variable's contrary Boolean value:

The not keyword is employed to switch the Boolean interpretation or outcome in conditional sentences or other Boolean equations. Not, unlike and, and or, determines the specific Boolean state, True or False, afterward returns the inverse.

|  |  |
| --- | --- |
| **Truth Table for not** | |
| **X** | **not X** |
| True | False |
| False | True |

**Code**

1. False and True
2. False or True
3. not True

**Output:**

False

True

False

**The in Keyword**

The in keyword of Python is a robust confinement checker, also known as a membership operator. If you provide it an element to seek and a container or series to seek into, it will give True or False, depending on if that given element was located in the given container:

1. <an\_element> in <a\_container>

Testing for a certain character in a string is a nice illustration of how to use the in keyword:

**Code**

1. container = "Javatpoint"
2. print( "p" in container )
3. print( "P" in container )

**Output:**

True

False

Lists, dictionaries, tuples, strings, or any data type with the method \_\_contains\_\_(), or we can iterate over it will work with the in keyword.

**The is Keyword**

In Python, it's used to check the identification of objects. The == operation is used to determine whether two arguments are identical. It also determines whether two arguments relate to the unique object.

When the objects are the same, it gives True; otherwise, it gives False.

**Code**

1. print( True is True )
2. print( False is True )
3. print( None is not None )
4. print( (9 + 5) is (7 \* 2) )

**Output:**

True

False

False

True

True, False, and None are all the same in Python since there is just one version.

**Code**

1. print( [] == [] )
2. print( [] is [] )
3. print( {} == {} )
4. print( {} is {} )

**Output:**

True

False

True

False

A blank dictionary or list is the same as another blank one. However, they aren't identical entities because they are stored independently in memory. This is because both the list and the dictionary are changeable.

**Code**

1. print( '' == '' )
2. print( '' is '' )

**Output:**

True

True

Strings and tuples, unlike lists and dictionaries, are unchangeable. As a result, two equal strings or tuples are also identical. They're both referring to the unique memory region.

**The nonlocal Keyword**

Nonlocal keyword usage is fairly analogous to global keyword usage. The keyword nonlocal is designed to indicate that a variable within a function that is inside a function, i.e., a nested function is just not local to it, implying that it is located in the outer function. We must define a non-local parameter with nonlocal if we ever need to change its value under a nested function. Otherwise, the nested function creates a local variable using that title. The example below will assist us in clarifying this.

**Code**

1. def the\_outer\_function():
2. var = 10
3. def the\_inner\_function():
4. nonlocal var
5. var = 14
6. print("The value inside the inner function: ", var)
7. the\_inner\_function()
8. print("The value inside the outer function: ", var)
10. the\_outer\_function()

**Output:**

The value inside the inner function: 14

The value inside the outer function: 14

the\_inner\_function() is placed inside the\_outer\_function in this case.

The the\_outer\_function has a variable named var. Var is not a global variable, as you may have noticed. As a result, if we wish to change it inside the the\_inner\_function(), we should declare it using nonlocal.

As a result, the variable was effectively updated within the nested the\_inner\_function, as evidenced by the results. The following is what happens if you don't use the nonlocal keyword:

**Code**

1. def the\_outer\_function():
2. var = 10
3. def the\_inner\_function():
4. var = 14
5. print("Value inside the inner function: ", var)
6. the\_inner\_function()
7. print("Value inside the outer function: ", var)
9. the\_outer\_function()

**Output:**

Value inside the inner function: 14

Value inside the outer function: 10

**Iteration Keywords: for, while, break, continue**

The iterative process and looping are essential programming fundamentals. To generate and operate with loops, Python has multiple keywords. These would be utilized and observed in almost every Python program. Knowing how to use them correctly can assist you in becoming a better Python developer.

**The for Keyword**

The for loop is by far the most popular loop in Python. It's built by blending two Python keywords. They are for and in, as previously explained.

**The while Keyword**

Python's while loop employs the term while and functions similarly to other computer languages' while loops. The block after the while phrase will be repeated repeatedly until the condition following the while keyword is false.

**The break Keyword**

If you want to quickly break out of a loop, employ the break keyword. We can use this keyword in both for and while loops.

**The continue Keyword**

You can use the continue Python keyword if you wish to jump to the subsequent loop iteration. The continue keyword, as in many other computer languages, enables you to quit performing the present loop iteration and go on to the subsequent one.

**Code**

1. # Program to show the use of keywords for, while, break, continue
2. for i in range(15):
4. print( i + 4, end = " ")
6. # breaking the loop when i = 9
7. if i == 9:
8. break
9. print()
11. # looping from 1 to 15
12. i = 0 # initial condition
13. while i < 15:
15. # When i has value 9, loop will jump to next iteration using continue. It will not print
16. if i == 9:
17. i += 3
18. continue
19. else:
20. # when i is not equal to 9, adding 2 and printing the value
21. print( i + 2, end = " ")
23. i += 1

**Output:**

4 5 6 7 8 9 10 11 12 13

2 3 4 5 6 7 8 9 10 14 15 16

**Exception Handling Keywords - try, except, raise, finally, and assert**

**try:** This keyword is designed to handle exceptions and is used in conjunction with the keyword except to handle problems in the program. When there is some kind of error, the program inside the "try" block is verified, but the code in that block is not executed.

**except:** As previously stated, this operates in conjunction with "try" to handle exceptions.

**finally:** Whatever the outcome of the "try" section, the "finally" box is implemented every time.

**raise:** The raise keyword could be used to specifically raise an exception.

**assert:** This method is used to help in troubleshooting. Often used to ensure that code is correct. Nothing occurs if an expression is interpreted as true; however, if it is false, "AssertionError" is raised. An output with the error, followed by a comma, can also be printed.

**Code**

1. # initializing the numbers
2. var1 = 4
3. var2 = 0
5. # Exception raised in the try section
6. try:
7. d = var1 // var2 # this will raise a "divide by zero" exception.
8. print( d )
9. # this section will handle exception raised in try block
10. except ZeroDivisionError:
11. print("We cannot divide by zero")
12. finally:
13. # If exception is raised or not, this block will be executed every time
14. print("This is inside finally block")
15. # by using assert keyword we will check if var2 is 0
16. print ("The value of var1 / var2 is : ")
17. assert var2 != 0, "Divide by 0 error"
18. print (var1 / var2)

**Output:**

We cannot divide by zero

This is inside finally block

The value of var1 / var2 is :

---------------------------------------------------------------------------

AssertionError Traceback (most recent call last)

Input In [44], in ()

15 # by using assert keyword we will check if var2 is 0

16 print ("The value of var1 / var2 is : ")

---> 17 assert var2 != 0, "Divide by 0 error"

18 print (var1 / var2)

AssertionError: Divide by 0 error

**The pass Keyword**

In Python, a null sentence is called a pass. It serves as a stand-in for something else. When it is run, nothing occurs.

Let's say we possess a function that has not been coded yet however we wish to do so in the long term. If we write just this in the middle of code,

**Code**

1. def function\_pass( arguments ):

**Output:**

def function\_pass( arguments ):

^

IndentationError: expected an indented block after function definition on line 1

as shown, IndentationError will be thrown. Rather, we use the pass command to create a blank container.

**Code**

1. def function\_pass( arguments ):
2. pass

We can use the pass keyword to create an empty class too.

**Code**

1. class passed\_class:
2. pass

**The return Keyword**

The return expression is used to leave a function and generate a result.

The None keyword is returned by default if we don't specifically return a value. The accompanying example demonstrates this.

**Code**

1. def func\_with\_return():
2. var = 13
3. return var
5. def func\_with\_no\_return():
6. var = 10
8. print( func\_with\_return() )
9. print( func\_with\_no\_return() )

**Output:**

13

None

**The del Keyword**

The del keyword is used to remove any reference to an object. In Python, every entity is an object. We can use the del command to remove a variable reference.

**Code**

1. var1 = var2 = 5
2. del var1
3. print( var2 )
4. print( var1 )

**Output:**

5

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

Input In [42], in ()

2 del var1

3 print( var2 )

----> 4 print( var1 )

NameError: name 'var1' is not defined

We can notice that the variable var1's reference has been removed. As a result, it's no longer recognized. However, var2 still exists.

Deleting entries from a collection like a list or a dictionary is also possible with del:

**Code**

1. list\_ = ['A','B','C']
2. del list\_[2]
3. print(list\_)

**Output:**

['A', 'B']

**Python Literals**

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

**1. String literals:**

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

**Example:**

1. "Aman" , '12345'

**Types of Strings:**

There are two types of Strings supported in Python:

**a) Single-line String**- Strings that are terminated within a single-line are known as Single line Strings.

**Example:**

1. text1='hello'

**b) Multi-line String -** A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

**1) Adding black slash at the end of each line.**

**Example:**

1. text1='hello\
2. user'
3. print(text1)

'hellouser'

**2) Using triple quotation marks:-**

**Example:**

1. str2='''''welcome
2. to
3. SSSIT'''
4. print str2

**Output:**

welcome

to

SSSIT

**II. Numeric literals:**

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Int(signed integers)** | **Long(long integers)** | **float(floating point)** | **Complex(complex)** |
| Numbers( can be both positive and negative) with no fractional part.eg: 100 | Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L | Real numbers with both integer and fractional part eg: -26.2 | In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j |

**Example - Numeric Literals**

1. x = 0b10100 #Binary Literals
2. y = 100 #Decimal Literal
3. z = 0o215 #Octal Literal
4. u = 0x12d #Hexadecimal Literal
6. #Float Literal
7. float\_1 = 100.5
8. float\_2 = 1.5e2
10. #Complex Literal
11. a = 5+3.14j
13. print(x, y, z, u)
14. print(float\_1, float\_2)
15. print(a, a.imag, a.real)

**Output:**

20 100 141 301

100.5 150.0

(5+3.14j) 3.14 5.0

**III. Boolean literals:**

A Boolean literal can have any of the two values: True or False.

**Example - Boolean Literals**

1. x = (1 == True)
2. y = (2 == False)
3. z = (3 == True)
4. a = True + 10
5. b = False + 10
7. print("x is", x)
8. print("y is", y)
9. print("z is", z)
10. print("a:", a)
11. print("b:", b)

**Output:**

x is True

y is False

z is False

a: 11

b: 10

**IV. Special literals.**

Python contains one special literal i.e., **None.**

None is used to specify to that field that is not created. It is also used for the end of lists in Python.

**Example - Special Literals**

1. val1=10
2. val2=None
3. print(val1)
4. print(val2)

**Output:**

10

None

**V. Literal Collections.**

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

**List:**

* List contains items of different data types. Lists are mutable i.e., modifiable.
* The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

**Example - List literals**

1. list=['John',678,20.4,'Peter']
2. list1=[456,'Andrew']
3. print(list)
4. print(list + list1)

**Output:**

['John', 678, 20.4, 'Peter']

['John', 678, 20.4, 'Peter', 456, 'Andrew']

**Dictionary:**

* Python dictionary stores the data in the key-value pair.
* It is enclosed by curly-braces {} and each pair is separated by the commas(,).

**Example**

1. dict = {'name': 'Pater', 'Age':18,'Roll\_nu':101}
2. print(dict)

**Output:**

{'name': 'Pater', 'Age': 18, 'Roll\_nu': 101}

**Tuple:**

* Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
* It is enclosed by the parentheses () and each element is separated by the comma(,).

**Example**

1. tup = (10,20,"Dev",[2,3,4])
2. print(tup)

**Output:**

(10, 20, 'Dev', [2, 3, 4])

**Set:**

* Python set is the collection of the unordered dataset.
* It is enclosed by the {} and each element is separated by the comma(,).

**Example: - Set Literals**

1. set = {'apple','grapes','guava','papaya'}
2. print(set)

**Output:**

{'guava', 'apple', 'papaya', 'grapes'}

**Python Operators**

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

* Arithmetic operators
* Comparison operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

**Arithmetic Operators**

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*) operators.

Consider the following table for a detailed explanation of arithmetic operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+ (Addition)** | It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30 |
| **- (Subtraction)** | It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10 |
| **/ (divide)** | It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0 |
| **\* (Multiplication)** | It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a \* b = 200 |
| **% (reminder)** | It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0 |
| **\*\* (Exponent)** | It is an exponent operator represented as it calculates the first operand power to the second operand. |
| **// (Floor division)** | It gives the floor value of the quotient produced by dividing the two operands. |

**Comparison operator**

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | If the value of two operands is equal, then the condition becomes true. |
| != | If the value of two operands is not equal, then the condition becomes true. |
| <= | If the first operand is less than or equal to the second operand, then the condition becomes true. |
| >= | If the first operand is greater than or equal to the second operand, then the condition becomes true. |
| > | If the first operand is greater than the second operand, then the condition becomes true. |
| **<** | If the first operand is less than the second operand, then the condition becomes true. |

**Assignment Operators**

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | It assigns the value of the right expression to the left operand. |
| += | It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30. |
| -= | It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10. |
| \*= | It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0. |
| \*\*= | a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will assign 4\*\*2 = 16 to a. |
| //= | A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a. |

**Bitwise Operators**

The bitwise operators perform bit by bit operation on the values of the two operands. Consider the following example.

**For example,**

1. if a = 7
2. b = 6
3. then, binary (a) = 0111
4. binary (b) = 0110
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & (binary and) | If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied. |
| | (binary or) | The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1. |
| ^ (binary xor) | The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0. |
| ~ (negation) | It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa. |
| << (left shift) | The left operand value is moved left by the number of bits present in the right operand. |
| >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |

**Logical Operators**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| and | If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true. |
| or | If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true. |
| not | If an expression **a** is true, then not (a) will be false and vice versa. |

**Membership Operators**

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| in | It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary). |
| not in | It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary). |

**Identity Operators**

The identity operators are used to decide whether an element certain class or type.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| is | It is evaluated to be true if the reference present at both sides point to the same object. |
| is not | It is evaluated to be true if the reference present at both sides do not point to the same object. |

**Operator Precedence**

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | The exponent operator is given priority over all the others used in the expression. |
| ~ + - | The negation, unary plus, and minus. |
| \* / % // | The multiplication, divide, modules, reminder, and floor division. |
| + - | Binary plus, and minus |
| >> << | Left shift. and right shift |
| & | Binary and. |
| ^ | | Binary xor, and or |
| <= < > >= | Comparison operators (less than, less than equal to, greater than, greater then equal to). |
| <> == != | Equality operators. |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

**Python Comments**

We'll study how to write comments in our program in this article. We'll also learn about single-line comments, multi-line comments, documentation strings, and other Python comments.

**Introduction to Python Comments**

We may wish to describe the code we develop. We might wish to take notes of why a section of script functions, for instance. We leverage the remarks to accomplish this. Formulas, procedures, and sophisticated business logic are typically explained with comments. The Python interpreter overlooks the remarks and solely interprets the script when running a program. Single-line comments, multi-line comments, and documentation strings are the 3 types of comments in Python.

**Advantages of Using Comments**

Our code is more comprehensible when we use comments in it. It assists us in recalling why specific sections of code were created by making the program more understandable.

Aside from that, we can leverage comments to overlook specific code while evaluating other code sections. This simple technique stops some lines from running or creates a fast pseudo-code for the program.

Below are some of the most common uses for comments:

* Readability of the Code
* Restrict code execution
* Provide an overview of the program or project metadata
* To add resources to the code

**Types of Comments in Python**

In Python, there are 3 types of comments. They are described below:

**Single-Line Comments**

Single-line remarks in Python have shown to be effective for providing quick descriptions for parameters, function definitions, and expressions. A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation. Consider the accompanying code snippet, which shows how to use a single line comment:

**Code**

1. # This code is to show an example of a single-line comment
2. print( 'This statement does not have a hashtag before it' )

**Output:**

This statement does not have a hashtag before it

The following is the comment:

1. # This code is to show an example of a single-line comment

The Python compiler ignores this line.

Everything following the # is omitted. As a result, we may put the program mentioned above in one line as follows:

**Code**

1. print( 'This is not a comment' ) # this code is to show an example of a single-line comment

**Output:**

This is not a comment

This program's output will be identical to the example above. The computer overlooks all content following #.

**Multi-Line Comments**

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

**With Multiple Hashtags (#)**

In Python, we may use hashtags (#) multiple times to construct multiple lines of comments. Every line with a (#) before it will be regarded as a single-line comment.

**Code**

1. # it is a
2. # comment
3. # extending to multiple lines

In this case, each line is considered a comment, and they are all omitted.

**Using String Literals**

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

**Code**

1. 'it is a comment extending to multiple lines'

We can observe that on running this code, there will be no output; thus, we utilize the strings inside triple quotes(""") as multi-line comments.

**Python Docstring**

The strings enclosed in triple quotes that come immediately after the defined function are called Python docstring. It's designed to link documentation developed for Python modules, methods, classes, and functions together. It's placed just beneath the function, module, or class to explain what they perform. The docstring is then readily accessible in Python using the \_\_doc\_\_ attribute.

**Code**

1. # Code to show how we use docstrings in Python
3. def add(x, y):
4. """This function adds the values of x and y"""
5. return x + y
7. # Displaying the docstring of the add function
8. print( add.\_\_doc\_\_ )

**Output:**

This function adds the values of x and y

**Python If-else statements**

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

|  |  |
| --- | --- |
| **Statement** | **Description** |
|  |  |
| If Statement | The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed. |
| If - else Statement | The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed. |
| Nested if Statement | Nested if statements enable us to use if ? else statement inside an outer if statement. |

**Indentation in Python**

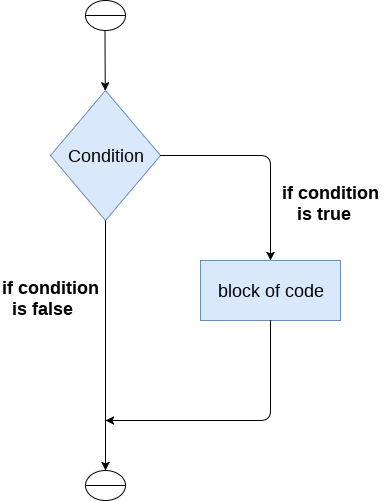
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

**The if statement**

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

1. if expression:
2. statement

**Example 1**

1. num = int(input("enter the number?"))
2. if num%2 == 0:
3. print("Number is even")

**Output:**

enter the number?10

Number is even

**Example 2 : Program to print the largest of the three numbers.**

1. a = int(input("Enter a? "));
2. b = int(input("Enter b? "));
3. c = int(input("Enter c? "));
4. if a>b and a>c:
5. print("a is largest");
6. if b>a and b>c:
7. print("b is largest");
8. if c>a and c>b:
9. print("c is largest");

**Output:**

Enter a? 100

Enter b? 120

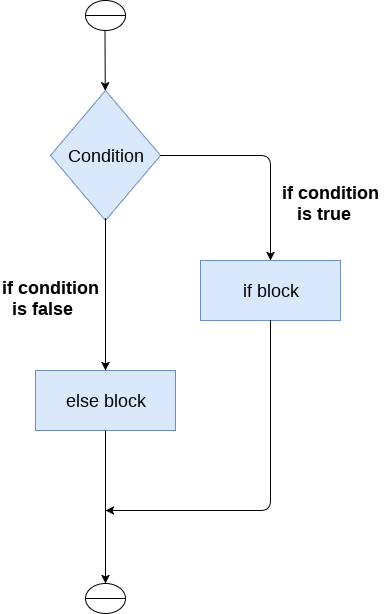
Enter c? 130

c is largest

**The if-else statement**

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

1. if condition:
2. #block of statements
3. else:
4. #another block of statements (else-block)

**Example 1 : Program to check whether a person is eligible to vote or not.**

1. age = int (input("Enter your age? "))
2. if age>=18:
3. print("You are eligible to vote !!");
4. else:
5. print("Sorry! you have to wait !!");

**Output:**

Enter your age? 90

You are eligible to vote !!

**Example 2: Program to check whether a number is even or not.**

1. num = int(input("enter the number?"))
2. if num%2 == 0:
3. print("Number is even...")
4. else:
5. print("Number is odd...")

**Output:**

enter the number?10

Number is even

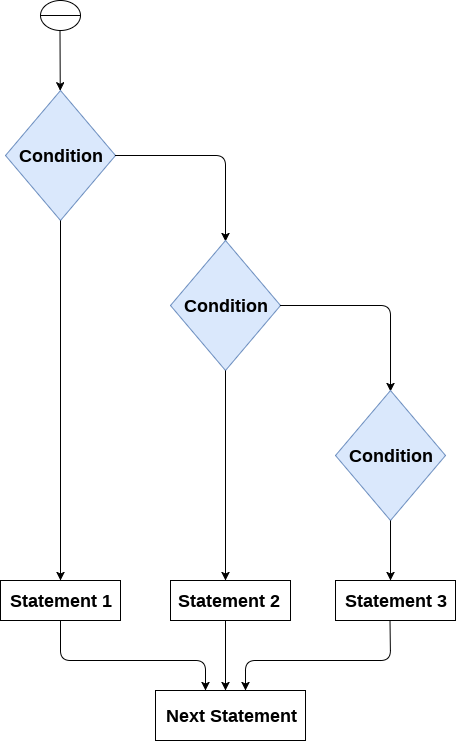
**The elif statement**

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.

1. if expression 1:
2. # block of statements
4. elif expression 2:
5. # block of statements
7. elif expression 3:
8. # block of statements
10. else:
11. # block of statements



**Example 1**

1. number = int(input("Enter the number?"))
2. if number==10:
3. print("number is equals to 10")
4. elif number==50:
5. print("number is equal to 50");
6. elif number==100:
7. print("number is equal to 100");
8. else:
9. print("number is not equal to 10, 50 or 100");

**Output:**

Enter the number?15

number is not equal to 10, 50 or 100

**Example 2**

1. marks = int(input("Enter the marks? "))
2. f marks > 85 and marks <= 100:
3. print("Congrats ! you scored grade A ...")
4. lif marks > 60 and marks <= 85:
5. print("You scored grade B + ...")
6. lif marks > 40 and marks <= 60:
7. print("You scored grade B ...")
8. lif (marks > 30 and marks <= 40):
9. print("You scored grade C ...")
10. lse:
11. print("Sorry you are fail ?")

**Python Loops**

The following loops are available in Python to fulfil the looping needs. Python offers 3 choices for running the loops. The basic functionality of all the techniques is the same, although the syntax and the amount of time required for checking the condition differ.

We can run a single statement or set of statements repeatedly using a loop command.

The following sorts of loops are available in the Python programming language.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the loop** | **Loop Type & Description** |
| 1 | **While loop** | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | **For loop** | This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable. |
| 3 | **Nested loops** | We can iterate a loop inside another loop. |

**Loop Control Statements**

Statements used to control loops and change the course of iteration are called control statements. All the objects produced within the local scope of the loop are deleted when execution is completed.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Python provides the following control statements. We will discuss them later in detail.

Let us quickly go over the definitions of these loop control statements.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the control statement** | **Description** |
| 1 | **Break statement** | This command terminates the loop's execution and transfers the program's control to the statement next to the loop. |
| 2 | **Continue statement** | This command skips the current iteration of the loop. The statements following the continue statement are not executed once the Python interpreter reaches the continue statement. |
| 3 | **Pass statement** | The pass statement is used when a statement is syntactically necessary, but no code is to be executed. |

**The for Loop**

Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.

**Syntax of the for Loop**

1. for value in sequence:
2. { code block }

In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.

Loop iterates until the final item of the sequence are reached.

**Code**

1. # Python program to show how the for loop works
3. # Creating a sequence which is a tuple of numbers
4. numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]
6. # variable to store the square of the number
7. square = 0
9. # Creating an empty list
10. squares = []
12. # Creating a for loop
13. for value in numbers:
14. square = value \*\* 2
15. squares.append(square)
16. print("The list of squares is", squares)

**Output:**

The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1, 81, 4]

**Using else Statement with for Loop**

As already said, a for loop executes the code block until the sequence element is reached. The statement is written right after the for loop is executed after the execution of the for loop is complete.

Only if the execution is complete does the else statement comes into play. It won't be executed if we exit the loop or if an error is thrown.

Here is a code to better understand if-else statements.

**Code**

1. # Python program to show how if-else statements work
3. string = "Python Loop"
5. # Initiating a loop
6. for s in a string:
7. # giving a condition in if block
8. if s == "o":
9. print("If block")
10. # if condition is not satisfied then else block will be executed
11. else:
12. print(s)

**Output:**

P

y

t

h

If block

n

L

If block

If block

p

Now similarly, using else with for loop.

**Syntax:**

1. for value in sequence:
2. # executes the statements until sequences are exhausted
3. else:
4. # executes these statements when for loop is completed

**Code**

1. # Python program to show how to use else statement with for loop
3. # Creating a sequence
4. tuple\_ = (3, 4, 6, 8, 9, 2, 3, 8, 9, 7)
6. # Initiating the loop
7. for value in tuple\_:
8. if value % 2 != 0:
9. print(value)
10. # giving an else statement
11. else:
12. print("These are the odd numbers present in the tuple")

**Output:**

3

9

3

9

7

These are the odd numbers present in the tuple

**The range() Function**

With the help of the range() function, we may produce a series of numbers. range(10) will produce values between 0 and 9. (10 numbers).

We can give specific start, stop, and step size values in the manner range(start, stop, step size). If the step size is not specified, it defaults to 1.

Since it doesn't create every value it "contains" after we construct it, the range object can be characterized as being "slow." It does provide in, len, and \_\_getitem\_\_ actions, but it is not an iterator.

The example that follows will make this clear.

**Code**

1. # Python program to show the working of range() function
3. print(range(15))
5. print(list(range(15)))
7. print(list(range(4, 9)))
9. print(list(range(5, 25, 4)))

**Output:**

range(0, 15)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

[4, 5, 6, 7, 8]

[5, 9, 13, 17, 21]

To iterate through a sequence of items, we can apply the range() method in for loops. We can use indexing to iterate through the given sequence by combining it with an iterable's len() function. Here's an illustration.

**Code**

1. # Python program to iterate over a sequence with the help of indexing
3. tuple\_ = ("Python", "Loops", "Sequence", "Condition", "Range")
5. # iterating over tuple\_ using range() function
6. for iterator in range(len(tuple\_)):
7. print(tuple\_[iterator].upper())

**Output:**

PYTHON

LOOPS

SEQUENCE

CONDITION

RANGE

**While Loop**

While loops are used in Python to iterate until a specified condition is met. However, the statement in the program that follows the while loop is executed once the condition changes to false.

**Syntax of the while loop is:**

1. while <condition>:
2. { code block }

All the coding statements that follow a structural command define a code block. These statements are intended with the same number of spaces. Python groups statements together with indentation.   
**Code**

1. # Python program to show how to use a while loop
2. counter = 0
3. # Initiating the loop
4. while counter < 10: # giving the condition
5. counter = counter + 3
6. print("Python Loops")

**Output:**

Python Loops

Python Loops

Python Loops

Python Loops

**Using else Statement with while Loops**

As discussed earlier in the for loop section, we can use the else statement with the while loop also. It has the same syntax.

**Code**

1. #Python program to show how to use else statement with the while loop
2. counter = 0
4. # Iterating through the while loop
5. while (counter < 10):
6. counter = counter + 3
7. print("Python Loops") # Executed untile condition is met
8. # Once the condition of while loop gives False this statement will be executed
9. else:
10. print("Code block inside the else statement")

**Output:**

Python Loops

Python Loops

Python Loops

Python Loops

Code block inside the else statement

**Single statement while Block**

The loop can be declared in a single statement, as seen below. This is similar to the if-else block, where we can write the code block in a single line.

**Code**

1. # Python program to show how to write a single statement while loop
2. counter = 0
3. while (count < 3): print("Python Loops")

**Loop Control Statements**

Now we will discuss the loop control statements in detail. We will see an example of each control statement.

**Continue Statement**

It returns the control to the beginning of the loop.

**Code**

1. # Python program to show how the continue statement works
3. # Initiating the loop
4. for string in "Python Loops":
5. if string == "o" or string == "p" or string == "t":
6. continue
7. print('Current Letter:', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: h

Current Letter: n

Current Letter:

Current Letter: L

Current Letter: s

**Break Statement**

It stops the execution of the loop when the break statement is reached.

**Code**

1. # Python program to show how the break statement works
3. # Initiating the loop
4. for string in "Python Loops":
5. if string == 'L':
6. break
7. print('Current Letter: ', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: t

Current Letter: h

Current Letter: o

Current Letter: n

Current Letter:

**Pass Statement**

Pass statements are used to create empty loops. Pass statement is also employed for classes, functions, and empty control statements.

**Code**

1. # Python program to show how the pass statement works
2. for a string in "Python Loops":
3. pass
4. print( 'Last Letter:', string)

**Output:**

Last Letter: s

**Python for loop**

Python is a powerful, general-purpose scripting language intended to be simple to understand and implement. It is free to access because it is open-source. This tutorial will teach us how to use Python for loops, one of the most basic looping instructions in Python programming.

**Introduction to for Loop in Python**

In Python, the for loop is often used to iterate over iterable objects such as lists, tuples, or strings. Traversal is the process of iterating across a series. If we have a section of code that we would like to repeat a certain number of times, we employ for loops. The for-loop is usually used on an iterable object such as a list or the in-built range function. The for statement in Python traverses through the elements of a series, running the block of code each time. The for statement is in opposition to the "while" loop, which is employed whenever a condition requires to be verified each repetition or when a piece of code is to be repeated indefinitely.

**Syntax of for Loop**

1. for value in sequence:
2. {loop body}

On each iteration, the value is the parameter that gets the element's value within the iterable sequence. If an expression statement is present in a sequence, it is processed first. The iterating variable iterating\_variable is then allocated to the first element in the sequence. After that, the intended block is run. The statement block is performed until the whole sequence is completed, and each element in the sequence is allocated to iterating\_variable. The for loop's material is distinguished from the rest of the program using indentation.

**Example of Python for Loop**

**Code**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. # Code to find the sum of squares of each element of the list using for loop
3. # creating the list of numbers
4. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
6. # initializing a variable that will store the sum
7. sum\_ = 0
9. # using for loop to iterate over the list
10. for num in numbers:
12. sum\_ = sum\_ + num \*\* 2
14. print("The sum of squares is: ", sum\_)

**Output:**

The sum of squares is: 774

**The range() Function**

Because the "range" function appears so frequently in for loops, we might mistakenly believe the range is a component of the syntax of for loop. It isn't: it's a Python built-in method that provides a series that follows a specified pattern (usually serial integers), fulfilling the criteria of giving a series for the for expression to run over. There is no necessity to count because for can act straight on sequences most of the time. If they're coming from some other language with distinctive loop syntax, this is a frequent novice construct:

**Code**

1. my\_list = [3, 5, 6, 8, 4]
2. for iter\_var in range( len( my\_list ) ):
3. my\_list.append(my\_list[iter\_var] + 2)
4. print( my\_list )

**Output:**

[3, 5, 6, 8, 4, 5, 7, 8, 10, 6]

**Iterating by Using Index of Sequence**

Another method of iterating through every item is to use an index offset within the sequence. Here's a simple illustration:

**Code**

1. # Code to find the sum of squares of each element of the list using for loop
3. # creating the list of numbers
4. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
6. # initializing a variable that will store the sum
7. sum\_ = 0
9. # using for loop to iterate over list
10. for num in range( len(numbers) ):
12. sum\_ = sum\_ + numbers[num] \*\* 2
14. print("The sum of squares is: ", sum\_)

**Output:**

The sum of squares is: 774

The len() built-in method that returns the total number of items in the list or tuple and the built-in function range(), which returns the exact sequence to iterate over, came in handy here.

**Using else Statement with for Loop**

Python allows you to connect an else expression with a loop expression.

When the else clause is combined with a for loop, it is performed after the circuit has finished iterating over the list.

The following instance shows how to use an otherwise statement in conjunction with a for expression to find students' marks from the record.

**Code**

1. # code to print marks of a student from the record
2. student\_name\_1 = 'Itika'
3. student\_name\_2 = 'Parker'

6. # Creating a dictionary of records of the students
7. records = {'Itika': 90, 'Arshia': 92, 'Peter': 46}
8. def marks( student\_name ):
9. for a\_student in record: # for loop will iterate over the keys of the dictionary
10. if a\_student == student\_name:
11. return records[ a\_student ]
12. break
13. else:
14. return f'There is no student of name {student\_name} in the records'
16. # giving the function marks() name of two students
17. print( f"Marks of {student\_name\_1} are: ", marks( student\_name\_1 ) )
18. print( f"Marks of {student\_name\_2} are: ", marks( student\_name\_2 ) )

**Output:**

Marks of Itika are: 90

Marks of Parker are: There is no student of name Parker in the records

**Nested Loops**

If we have a piece of script that we want to run a number of times and then another piece of script inside that script that we want to run B number of times, we employ a "nested loop." When working with an iterable in the lists, these are widely utilized in Python.

**Code**

1. import random
2. numbers = [ ]
3. for val in range(0, 11):
4. numbers.append( random.randint( 0, 11 ) )
5. for num in range( 0, 11 ):
6. for i in numbers:
7. if num == i:
8. print( num, end = " " )

**Output:**

0 2 4 5 6 7 8 8 9 10

**Python While Loops**

In coding, loops are designed to execute a specified code block repeatedly. We'll learn how to construct a while loop in Python, the syntax of a while loop, loop controls like break and continue, and other exercises in this tutorial.

**Introduction of Python While Loop**

The Python while loop iteration of a code block is executed as long as the given condition, i.e., conditional\_expression, is true.

If we don't know how many times we'll execute the iteration ahead of time, we can write an indefinite loop.

**Syntax of Python While Loop**

1. while conditional\_expression:
2. Code block of while

The given condition, i.e., conditional\_expression, is evaluated initially in the Python while loop. Then, if the conditional expression gives a boolean value True, the while loop statements are executed. The conditional expression is verified again when the complete code block is executed. This procedure repeatedly occurs until the conditional expression returns the boolean value False.

* The statements of the Python while loop are dictated by indentation.
* The code block begins when a statement is indented & ends with the very first unindented statement.
* Any non-zero number in Python is interpreted as boolean True. False is interpreted as None and 0.

**Python While Loop Example**

Here we will sum of squares of the first 15 natural numbers using a while loop.

**Code**

1. # Python program example to show the use of while loop
3. num = 15
5. # initializing summation and a counter for iteration
6. summation = 0
7. c = 1
9. while c <= num: # specifying the condition of the loop
10. # begining the code block
11. summation = c\*\*2 + summation
12. c = c + 1    # incrementing the counter
14. # print the final sum
15. print("The sum of squares is", summation)

**Output:**

The sum of squares is 1240

Provided that our counter parameter i gives boolean true for the condition, i less than or equal to num, the loop repeatedly executes the code block i number of times.

Next is a crucial point (which is mostly forgotten). We have to increment the counter parameter's value in the loop's statements. If we don't, our while loop will execute itself indefinitely (a never-ending loop).

Finally, we print the result using the print statement.

**Exercises of Python While Loop**

**Prime Numbers and Python While Loop**

Using a while loop, we will construct a Python program to verify if the given integer is a prime number or not.

**Code**

1. num = [34, 12, 54, 23, 75, 34, 11]
3. def prime\_number(number):
4. condition = 0
5. iteration = 2
6. while iteration <= number / 2:
7. if number % iteration == 0:
8. condition = 1
9. break
10. iteration = iteration + 1
12. if condition == 0:
13. print(f"{number} is a PRIME number")
14. else:
15. print(f"{number} is not a PRIME number")
16. for i in num:
17. prime\_number(i)

**Output:**

34 is not a PRIME number

12 is not a PRIME number

54 is not a PRIME number

23 is a PRIME number

75 is not a PRIME number

34 is not a PRIME number

11 is a PRIME number

**Multiplication Table using While Loop**

In this example, we will use the while loop for printing the multiplication table of a given number.

**Code**

1. num = 21
2. counter = 1
3. # we will use a while loop for iterating 10 times for the multiplication table
4. print("The Multiplication Table of: ", num)
5. while counter <= 10: # specifying the condition
6. ans = num \* counter
7. print (num, 'x', counter, '=', ans)
8. counter += 1 # expression to increment the counter

**Output:**

The Multiplication Table of: 21

21 x 1 = 21

21 x 2 = 42

21 x 3 = 63

21 x 4 = 84

21 x 5 = 105

21 x 6 = 126

21 x 7 = 147

21 x 8 = 168

21 x 9 = 189

21 x 10 = 210

**Python While Loop with List**

We will use a Python while loop to square every number of a list

**Code**

1. # Python program to square every number of a list
2. # initializing a list
3. list\_ = [3, 5, 1, 4, 6]
4. squares = []
5. # programing a while loop
6. while list\_: # until list is not empty this expression will give boolean True after that False
7. squares.append( (list\_.pop())\*\*2)
8. # print the squares
9. print( squares )

[36, 16, 1, 25, 9]

In the preceding example, we execute a while loop over a given list of integers that will repeatedly run as long as an element in the list is found.

**Python While Loop Multiple Conditions**

We'll need to recruit logical operators to combine two or more expressions specifying conditions into a single while loop. This instructs Python on collectively analyzing all of the given expressions of conditions.

We can construct a while loop with multiple conditions in this example. We have given two conditions and a and keyword, meaning until both conditions give boolean True, the loop will execute the statements.

**Code**

1. num1 = 17
2. num2 = -12
4. while num1 > 5 and num2 < -5 : # multiple conditions in a single while loop
5. num1 -= 2
6. num2 += 3
7. print( (num1, num2) )

**Output:**

(15, -9)

(13, -6)

(11, -3)

Let's look at another example of multiple conditions with an OR operator.

**Code**

1. num1 = 17
2. num2 = -12
4. while num1 > 5 or num2 < -5 :
5. num1 -= 2
6. num2 += 3
7. print( (num1, num2) )

**Output:**

(15, -9)

(13, -6)

(11, -3)

(9, 0)

(7, 3)

(5, 6)

We can also group multiple logical expressions in the while loop, as shown in this example.

**Code**

1. num1 = 9
2. num = 14
3. maximum\_value = 4
4. counter = 0
5. while (counter < num1 or counter < num2) and not counter >= maximum\_value: # grouping multiple conditions
6. print(f"Number of iterations: {counter}")
7. counter += 1

**Output:**

Number of iterations: 0

Number of iterations: 1

Number of iterations: 2

Number of iterations: 3

**Single Statement While Loop**

Similar to the if statement syntax, if our while clause consists of one statement, it may be written on the same line as the while keyword.

Here is the syntax and example of a one-line while clause -

1. # Python program to show how to create a single statement while loop
2. counter = 1
3. while counter: print('Python While Loops')

**Loop Control Statements**

Now we will discuss the loop control statements in detail. We will see an example of each control statement.

**Continue Statement**

It returns the control of the Python interpreter to the beginning of the loop.

**Code**

1. # Python program to show how to use continue loop control
3. # Initiating the loop
4. for string in "While Loops":
5. if string == "o" or string == "i" or string == "e":
6. continue
7. print('Current Letter:', string)

**Output:**

Current Letter: W

Current Letter: h

Current Letter: l

Current Letter:

Current Letter: L

Current Letter: p

Current Letter: s

**Break Statement**

It stops the execution of the loop when the break statement is reached.

**Code**

1. # Python program to show how to use the break statement
3. # Initiating the loop
4. for string in "Python Loops":
5. if string == 'n':
6. break
7. print('Current Letter: ', string)

**Output:**

Current Letter: P

Current Letter: y

Current Letter: t

Current Letter: h

Current Letter: o

**Pass Statement**

Pass statements are used to create empty loops. Pass statement is also employed for classes, functions, and empty control statements.

**Code**

1. # Python program to show how to use the pass statement
2. for a string in "Python Loops":
3. pass
4. print( 'Last Letter:', string)

**Output:**

Last Letter: s

**Python break statement**

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

1. #loop statements
2. break;

**Example 1**

1. list =[1,2,3,4]
2. count = 1;
3. for i in list:
4. if i == 4:
5. print("item matched")
6. count = count + 1;
7. break
8. print("found at",count,"location");

**Output:**

item matched

found at 2 location

**Example 2**

1. str = "python"
2. for i in str:
3. if i == 'o':
4. break
5. print(i);

**Output:**

p

y

t

h

**Example 3: break statement with while loop**

1. i = 0;
2. while 1:
3. print(i," ",end=""),
4. i=i+1;
5. if i == 10:
6. break;
7. print("came out of while loop");

**Output:**

0 1 2 3 4 5 6 7 8 9 came out of while loop

**Example 3**

1. n=2
2. while 1:
3. i=1;
4. while i<=10:
5. print("%d X %d = %d\n"%(n,i,n\*i));
6. i = i+1;
7. choice = int(input("Do you want to continue printing the table, press 0 for no?"))
8. if choice == 0:
9. break;
10. n=n+1

**Output:**

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0

**Python continue Statement**

In this tutorial, we'll look at how to use Python continue keyword to skip the remaining statements of the current loop and go to the next iteration. Also, the difference between continue and pass keywords.

**Application of the Continue Statement**

In Python, loops repeat processes on their own in an efficient way. However, there might be occasions when we wish to leave the current loop entirely, skip iteration, or dismiss the condition controlling the loop. We use Loop control statements in such cases. The continue keyword is a loop control statement that allows us to change the loop's control.

**The continue Keyword**

In Python, the continue keyword return control of the iteration to the beginning of the Python for loop or Python while loop. All remaining lines in the prevailing iteration of the loop are skipped by the continue keyword, which returns execution to the beginning of the next iteration of the loop.

Both Python while and Python for loops can leverage the continue statements.

**Example of Python Continue Statements in For Loop**

Assume the following scenario: we want to develop a program that returns numbers from 10 to 20 but not 15. It is mentioned that we must perform this with a ***'for'*** loop. Here's when the continue keyword comes into play. We will execute a loop from 10 to 20 and test the condition that the iterator is equal to 15. If it equals 15, we'll employ the continue statement to skip to the following iteration displaying any output; otherwise, the loop will print the result.

The following code is an example of the above scenario:

**Code**

1. # Python code to show example of continue statement
3. # looping from 10 to 20
4. for iterator in range(10, 21):
6. # If iterator is equals to 15, loop will continue to the next iteration
7. if iterator == 15:
8. continue
9. # otherwise printing the value of iterator
10. print( iterator )

**Output:**

10

11

12

13

14

16

17

18

19

20

Now will repeat the above code, but this time with a string. We will take a string "Javatpoint" and print each letter of the string except "a". This time we will use Python while loop to do so. Until the value of the iterator is less than the string's length, the while loop will keep executing.

**Code**

1. # Creating a string
2. string = "JavaTpoint"
3. # initializing an iterator
4. iterator = 0
6. # starting a while loop
7. while iterator < len(string):
8. # if loop is at letter a it will skip the remaining code and go to next iteration
9. if string[iterator] == 'a':
10. continue
11. # otherwise it will print the letter
12. print(string[ iterator ])
13. iterator += 1

**Output:**

J

v

T

p

o

i

n

t

**Python Continue vs. Pass**

Usually, there is some confusion in the pass and continue keywords. So here are the differences between these two.

|  |  |  |
| --- | --- | --- |
| **Headings** | **continue** | **pass** |
| **Definition** | The continue statement is utilized to skip the current loop's remaining statements, go to the following iteration, and return control to the beginning. | The pass keyword is used when a phrase is necessary syntactically to be placed but not to be executed. |
| **Action** | It takes the control back to the start of the loop. | Nothing happens if the Python interpreter encounters the pass statement. |
| **Application** | It works with both the Python while and Python for loops. | It performs nothing; hence it is a null operation. |
| **Syntax** | It has the following syntax: -: continue | Its syntax is as follows:- pass |
| **Interpretation** | It's mostly utilized within a loop's condition. | During the byte-compile stage, the pass keyword is removed. |

**Python Pass Statement**

We will discover more about pass statements in this tutorial. It is interpreted as a placeholder for the future execution of functions, classes, loops, etc.

**What is Pass Statement in Python?**

The null statement is another name for the pass statement. A Comment is not ignored by the Python interpreter, whereas a pass statement is not. Hence, they two are different Python keywords.

We can use the pass statement as a placeholder when unsure what code to provide. So, we only have to place the pass on that line. Pass may be used when we don't wish any code to be executed. We can simply insert a pass in places where empty code is prohibited, such as loops, functions, class definitions, or if-else statements.

**Syntax of the Pass Keyword**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. Keyword:
2. pass

Typically, we utilise it as a reference for the future.

Let's say we have a loop or an if-else statement that isn't to be filled now but that we wish to in the future. The pass keyword cannot have an empty body as it will be syntactically wrong. An error would be displayed by the Python interpreter suggesting to fill the space. Therefore, we create a code block that performs nothing using the pass statement.

**Example of the Pass Statement**

**Code**

1. # Python program to show how to use a pass statement in a for loop
2. '''''pass acts as a placeholder. We can fill this place later on'''
3. sequence = {"Python", "Pass", "Statement", "Placeholder"}
4. for value in sequence:
5. if value == "Pass":
6. pass # leaving an empty if block using the pass keyword
7. else:
8. print("Not reached pass keyword: ", value)

**Output:**

Not reached pass keyword: Python

Not reached pass keyword: Placeholder

Not reached pass keyword: Statement

The same thing is also possible to create an empty function or a class.

**Code**

1. # Python program to show how to create an empty function and an empty class
3. # Empty function:
4. def empty():
5. pass
7. # Empty class
8. class Empty:
9. pass

**Python String**

Till now, we have discussed numbers as the standard data-types in Python. In this section of the tutorial, we will discuss the most popular data type in Python, i.e., string.

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Consider the following example in Python to create a string.

**Syntax:**

1. str = "Hi Python !"

Here, if we check the type of the variable **str** using a Python script

1. print(type(str)), then it will print a string (str).

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

**Creating String in Python**

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or **docstrings**.

1. #Using single quotes
2. str1 = 'Hello Python'
3. print(str1)
4. #Using double quotes
5. str2 = "Hello Python"
6. print(str2)
8. #Using triple quotes
9. str3 = '''''Triple quotes are generally used for
10. represent the multiline or
11. docstring'''
12. print(str3)

**Output:**

Hello Python

Hello Python

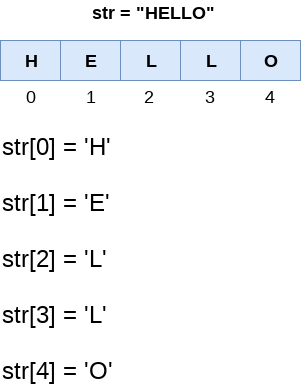
Triple quotes are generally used for

represent the multiline or

docstring

**Strings indexing and splitting**

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



Consider the following example:

1. str = "HELLO"
2. print(str[0])
3. print(str[1])
4. print(str[2])
5. print(str[3])
6. print(str[4])
7. # It returns the IndexError because 6th index doesn't exist
8. print(str[6])

**Output:**

H

E

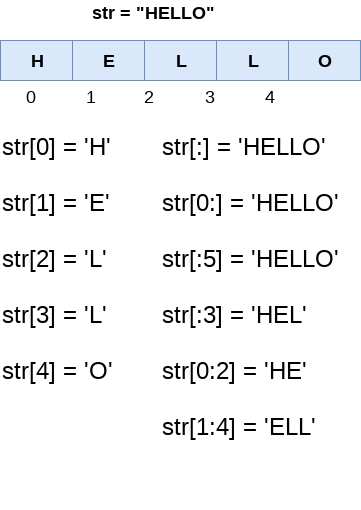
L

L

O

IndexError: string index out of range

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in Python to access the substring from the given string. Consider the following example.



Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

Consider the following example:

1. # Given String
2. str = "JAVATPOINT"
3. # Start Oth index to end
4. print(str[0:])
5. # Starts 1th index to 4th index
6. print(str[1:5])
7. # Starts 2nd index to 3rd index
8. print(str[2:4])
9. # Starts 0th to 2nd index
10. print(str[:3])
11. #Starts 4th to 6th index
12. print(str[4:7])

**Output:**

JAVATPOINT

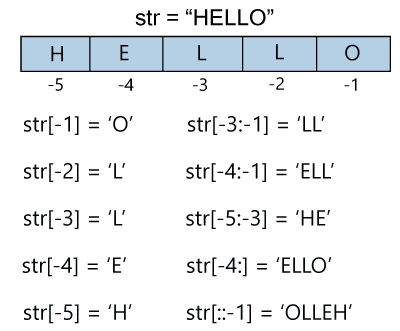
AVAT

VA

JAV

TPO

We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.



Consider the following example

1. str = 'JAVATPOINT'
2. print(str[-1])
3. print(str[-3])
4. print(str[-2:])
5. print(str[-4:-1])
6. print(str[-7:-2])
7. # Reversing the given string
8. print(str[::-1])
9. print(str[-12])

**Output:**

T

I

NT

OIN

ATPOI

TNIOPTAVAJ

IndexError: string index out of range

**Reassigning Strings**

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.

Consider the following example.

**Example 1**

1. str = "HELLO"
2. str[0] = "h"
3. print(str)

**Output:**

Traceback (most recent call last):

File "12.py", line 2, in <module>

str[0] = "h";

TypeError: 'str' object does not support item assignment

However, in example 1, the string **str** can be assigned completely to a new content as specified in the following example.

**Example 2**

1. str = "HELLO"
2. print(str)
3. str = "hello"
4. print(str)

**Output:**

HELLO

hello

**Deleting the String**

As we know that strings are immutable. We cannot delete or remove the characters from the string.  But we can delete the entire string using the **del** keyword.

1. str = "JAVATPOINT"
2. del str[1]

**Output:**

TypeError: 'str' object doesn't support item deletion

Now we are deleting entire string.

1. str1 = "JAVATPOINT"
2. del str1
3. print(str1)

**Output:**

NameError: name 'str1' is not defined

**String Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | It is known as concatenation operator used to join the strings given either side of the operator. |
| \* | It is known as repetition operator. It concatenates the multiple copies of the same string. |
| [] | It is known as slice operator. It is used to access the sub-strings of a particular string. |
| [:] | It is known as range slice operator. It is used to access the characters from the specified range. |
| in | It is known as membership operator. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |
| r/R | It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string. |
| % | It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python. |

**Example**

Consider the following example to understand the real use of Python operators.

1. str = "Hello"
2. str1 = " world"
3. print(str\*3) # prints HelloHelloHello
4. print(str+str1)# prints Hello world
5. print(str[4]) # prints o
6. print(str[2:4]); # prints ll
7. print('w' in str) # prints false as w is not present in str
8. print('wo' not in str1) # prints false as wo is present in str1.
9. print(r'C://python37') # prints C://python37 as it is written
10. print("The string str : %s"%(str)) # prints The string str : Hello

**Output:**

HelloHelloHello

Hello world

o

ll

False

False

C://python37

The string str : Hello

**Python String Formatting**

**Escape Sequence**

Let's suppose we need to write the text as - They said, "Hello what's going on?"- the given statement can be written in single quotes or double quotes but it will raise the **SyntaxError** as it contains both single and double-quotes.

**Example**

Consider the following example to understand the real use of Python operators.

1. str = "They said, "Hello what's going on?""
2. print(str)

**Output:**

SyntaxError: invalid syntax

We can use the triple quotes to accomplish this problem but Python provides the escape sequence.

The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

**Example -**

1. # using triple quotes
2. print('''''They said, "What's there?"''')
4. # escaping single quotes
5. print('They said, "What\'s going on?"')
7. # escaping double quotes
8. print("They said, \"What's going on?\"")

**Output:**

They said, "What's there?"

They said, "What's going on?"

They said, "What's going on?"

The list of an escape sequence is given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Escape Sequence** | **Description** | **Example** |
| 1. | \newline | It ignores the new line. | print("Python1 \  Python2 \  Python3")  **Output:**  Python1 Python2 Python3 |
| 2. | \\ | Backslash | print("\\")  **Output:**  \ |
| 3. | \' | Single Quotes | print('\'')  **Output:**  ' |
| 4. | \\'' | Double Quotes | print("\"")  **Output:**  " |
| 5. | \a | ASCII Bell | print("\a") |
| 6. | \b | ASCII Backspace(BS) | print("Hello \b World")  **Output:**  Hello World |
| 7. | \f | ASCII Formfeed | print("Hello \f World!")  Hello World! |
| 8. | \n | ASCII Linefeed | print("Hello \n World!")  **Output:**  Hello  World! |
| 9. | \r | ASCII Carriege Return(CR) | print("Hello \r World!")  **Output:**  World! |
| 10. | \t | ASCII Horizontal Tab | print("Hello \t World!")  **Output:**  Hello World! |
| 11. | \v | ASCII Vertical Tab | print("Hello \v World!")  **Output:**  Hello  World! |
| 12. | \ooo | Character with octal value | print("\110\145\154\154\157")  **Output:**  Hello |
| 13 | \xHH | Character with hex value. | print("\x48\x65\x6c\x6c\x6f")  **Output:**  Hello |

Here is the simple example of escape sequence.

1. print("C:\\Users\\DEVANSH SHARMA\\Python32\\Lib")
2. print("This is the \n multiline quotes")
3. print("This is \x48\x45\x58 representation")

**Output:**

C:\Users\DEVANSH SHARMA\Python32\Lib

This is the

multiline quotes

This is HEX representation

We can ignore the escape sequence from the given string by using the raw string. We can do this by writing **r** or **R** in front of the string. Consider the following example.

1. print(r"C:\\Users\\DEVANSH SHARMA\\Python32")

**Output:**

C:\\Users\\DEVANSH SHARMA\\Python32

**The format() method**

The **format()** method is the most flexible and useful method in formatting strings. The curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument. Let's have a look at the given an example:

1. # Using Curly braces
2. print("{} and {} both are the best friend".format("Devansh","Abhishek"))
4. #Positional Argument
5. print("{1} and {0} best players ".format("Virat","Rohit"))
7. #Keyword Argument
8. print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))

**Output:**

Devansh and Abhishek both are the best friend

Rohit and Virat best players

James,Peter,Ricky

**Python String Formatting Using % Operator**

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Consider the following example.

1. Integer = 10;
2. Float = 1.290
3. String = "Devansh"
4. print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String))

**Output:**

Hi I am Integer ... My value is 10

Hi I am float ... My value is 1.290000

Hi I am string ... My value is Devansh

**Python String functions**

Python provides various in-built functions that are used for string handling. Many String fun

|  |  |
| --- | --- |
| **Method** | **Description** |
| [capitalize()](https://www.javatpoint.com/python-string-capitalize-method) | It capitalizes the first character of the String. This function is deprecated in python3 |
| [casefold()](https://www.javatpoint.com/python-string-casefold-method) | It returns a version of s suitable for case-less comparisons. |
| [center(width ,fillchar)](https://www.javatpoint.com/python-string-center-method) | It returns a space padded string with the original string centred with equal number of left and right spaces. |
| [count(string,begin,end)](https://www.javatpoint.com/python-string-count-method) | It counts the number of occurrences of a substring in a String between begin and end index. |
| decode(encoding = 'UTF8', errors = 'strict') | Decodes the string using codec registered for encoding. |
| [encode()](https://www.javatpoint.com/python-string-encode-method) | Encode S using the codec registered for encoding. Default encoding is 'utf-8'. |
| [endswith(suffix ,begin=0,end=len(string))](https://www.javatpoint.com/python-string-endswith-method) | It returns a Boolean value if the string terminates with given suffix between begin and end. |
| [expandtabs(tabsize = 8)](https://www.javatpoint.com/python-string-expandtabs-method) | It defines tabs in string to multiple spaces. The default space value is 8. |
| [find(substring ,beginIndex, endIndex)](https://www.javatpoint.com/python-string-find-method) | It returns the index value of the string where substring is found between begin index and end index. |
| [format(value)](https://www.javatpoint.com/python-string-format-method) | It returns a formatted version of S, using the passed value. |
| [index(subsring, beginIndex, endIndex)](https://www.javatpoint.com/python-string-index-method) | It throws an exception if string is not found. It works same as find() method. |
| [isalnum()](https://www.javatpoint.com/python-string-isalnum-method) | It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false. |
| [isalpha()](https://www.javatpoint.com/python-string-isalpha-method) | It returns true if all the characters are alphabets and there is at least one character, otherwise False. |
| [isdecimal()](https://www.javatpoint.com/python-string-isdecimal-method) | It returns true if all the characters of the string are decimals. |
| [isdigit()](https://www.javatpoint.com/python-string-isdigit-method) | It returns true if all the characters are digits and there is at least one character, otherwise False. |
| [isidentifier()](https://www.javatpoint.com/python-string-isidentifier-method) | It returns true if the string is the valid identifier. |
| [islower()](https://www.javatpoint.com/python-string-islower-method) | It returns true if the characters of a string are in lower case, otherwise false. |
| [isnumeric()](https://www.javatpoint.com/python-string-isnumeric-method) | It returns true if the string contains only numeric characters. |
| [isprintable()](https://www.javatpoint.com/python-string-isprintable-method) | It returns true if all the characters of s are printable or s is empty, false otherwise. |
| [isupper()](https://www.javatpoint.com/python-string-isupper-method) | It returns false if characters of a string are in Upper case, otherwise False. |
| [isspace()](https://www.javatpoint.com/python-string-isspace-method) | It returns true if the characters of a string are white-space, otherwise false. |
| [istitle()](https://www.javatpoint.com/python-string-istitle-method) | It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case. |
| [isupper()](https://www.javatpoint.com/python-string-isupper-method) | It returns true if all the characters of the string(if exists) is true otherwise it returns false. |
| [join(seq)](https://www.javatpoint.com/python-string-join-method) | It merges the strings representation of the given sequence. |
| len(string) | It returns the length of a string. |
| [ljust(width[,fillchar])](https://www.javatpoint.com/python-string-ljust-method) | It returns the space padded strings with the original string left justified to the given width. |
| [lower()](https://www.javatpoint.com/python-string-lower-method) | It converts all the characters of a string to Lower case. |
| [lstrip()](https://www.javatpoint.com/python-string-lstrip-method) | It removes all leading whitespaces of a string and can also be used to remove particular character from leading. |
| [partition()](https://www.javatpoint.com/python-string-partition-method) | It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings. |
| maketrans() | It returns a translation table to be used in translate function. |
| [replace(old,new[,count])](https://www.javatpoint.com/python-string-replace-method) | It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given. |
| [rfind(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-rfind-method) | It is similar to find but it traverses the string in backward direction. |
| [rindex(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-rindex-method) | It is same as index but it traverses the string in backward direction. |
| [rjust(width,[,fillchar])](https://www.javatpoint.com/python-string-rjust-method) | Returns a space padded string having original string right justified to the number of characters specified. |
| [rstrip()](https://www.javatpoint.com/python-string-rstrip-method) | It removes all trailing whitespace of a string and can also be used to remove particular character from trailing. |
| [rsplit(sep=None, maxsplit = -1)](https://www.javatpoint.com/python-string-rsplit-method) | It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space. |
| [split(str,num=string.count(str))](https://www.javatpoint.com/python-string-split-method) | Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter. |
| [splitlines(num=string.count('\n'))](https://www.javatpoint.com/python-string-splitlines-method) | It returns the list of strings at each line with newline removed. |
| [startswith(str,beg=0,end=len(str))](https://www.javatpoint.com/python-string-startswith-method) | It returns a Boolean value if the string starts with given str between begin and end. |
| strip([chars]) | It is used to perform lstrip() and rstrip() on the string. |
| [swapcase()](https://www.javatpoint.com/python-string-swapcase-method) | It inverts case of all characters in a string. |
| title() | It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut. |
| [translate(table,deletechars = '')](https://www.javatpoint.com/python-string-translate-method) | It translates the string according to the translation table passed in the function . |
| [upper()](https://www.javatpoint.com/python-string-upper-method) | It converts all the characters of a string to Upper Case. |
| [zfill(width)](https://www.javatpoint.com/python-string-zfill-method) | Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| [rpartition()](https://www.javatpoint.com/python-string-rpartition-method) |  |

**Python List**

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]

IIf we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

1. print(type(L1))
2. print(type(L2))

**Output:**

<class 'list'>

<class 'list'>

**Characteristics of Lists**

The list has the following characteristics:

* The lists are ordered.
* The element of the list can access by index.
* The lists are the mutable type.
* The lists are mutable types.
* A list can store the number of various elements.

Let's check the first statement that lists are the ordered.

1. a = [1,2,"Peter",4.50,"Ricky",5,6]
2. b = [1,2,5,"Peter",4.50,"Ricky",6]
3. a ==b

**Output:**

False

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

1. a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
2. b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
3. a == b

**Output:**

True

Let's have a look at the list example in detail.

1. emp = ["John", 102, "USA"]
2. Dep1 = ["CS",10]
3. Dep2 = ["IT",11]
4. HOD\_CS = [10,"Mr. Holding"]
5. HOD\_IT = [11, "Mr. Bewon"]
6. print("printing employee data...")
7. print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
8. print("printing departments...")
9. print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))
10. print("HOD Details ....")
11. print("CS HOD Name: %s, Id: %d"%(HOD\_CS[1],HOD\_CS[0]))
12. print("IT HOD Name: %s, Id: %d"%(HOD\_IT[1],HOD\_IT[0]))
13. print(type(emp),type(Dep1),type(Dep2),type(HOD\_CS),type(HOD\_IT))

**Output:**

printing employee data...

Name : John, ID: 102, Country: USA

printing departments...

Department 1:

Name: CS, ID: 11

Department 2:

Name: IT, ID: 11

HOD Details ....

CS HOD Name: Mr. Holding, Id: 10

IT HOD Name: Mr. Bewon, Id: 11

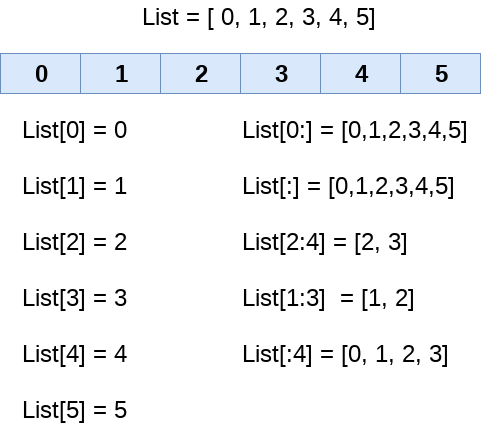
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

In the above example, we have created the lists which consist of the employee and department details and printed the corresponding details. Observe the above code to understand the concept of the list better.

**List indexing and splitting**

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



We can get the sub-list of the list using the following syntax.

1. list\_varible(start:stop:step)

* The **start** denotes the starting index position of the list.
* The **stop** denotes the last index position of the list.
* The **step** is used to skip the nth element within a **start:stop**

Consider the following example:

1. list = [1,2,3,4,5,6,7]
2. print(list[0])
3. print(list[1])
4. print(list[2])
5. print(list[3])
6. # Slicing the elements
7. print(list[0:6])
8. # By default the index value is 0 so its starts from the 0th element and go for index -1.
9. print(list[:])
10. print(list[2:5])
11. print(list[1:6:2])

**Output:**

1

2

3

4

[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7]

[3, 4, 5]

[2, 4, 6]

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.

Python Lists

Let's have a look at the following example where we will use negative indexing to access the elements of the list.

1. list = [1,2,3,4,5]
2. print(list[-1])
3. print(list[-3:])
4. print(list[:-1])
5. print(list[-3:-1])

**Output:**

5

[3, 4, 5]

[1, 2, 3, 4]

[3, 4]

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

**Updating List values**

Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator.

Python also provides append() and insert() methods, which can be used to add values to the list.

Consider the following example to update the values inside the list.

1. list = [1, 2, 3, 4, 5, 6]
2. print(list)
3. # It will assign value to the value to the second index
4. list[2] = 10
5. print(list)
6. # Adding multiple-element
7. list[1:3] = [89, 78]
8. print(list)
9. # It will add value at the end of the list
10. list[-1] = 25
11. print(list)

**Output:**

[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

1. list = [1, 2, 3, 4, 5, 6]
2. print(list)
3. # It will assign value to the value to second index
4. list[2] = 10
5. print(list)
6. # Adding multiple element
7. list[1:3] = [89, 78]
8. print(list)
9. # It will add value at the end of the list
10. list[-1] = 25
11. print(list)

**Output:**

[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]

**Python List Operations**

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings.

Let's see how the list responds to various operators.

1. Consider a Lists l1 = [1, 2, 3, 4], and l2 = [5, 6, 7, 8] to perform operation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the list elements to be repeated multiple times. | L1\*2 = [1, 2, 3, 4, 1, 2, 3, 4] |
| Concatenation | It concatenates the list mentioned on either side of the operator. | l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8] |
| Membership | It returns true if a particular item exists in a particular list otherwise false. | print(2 in l1) prints True. |
| Iteration | The for loop is used to iterate over the list elements. | for i in l1:  print(i)  **Output**  1  2  3  4 |
| Length | It is used to get the length of the list | len(l1) = 4 |

**Iterating a List**

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

1. list = ["John", "David", "James", "Jonathan"]
2. for i in list:
3. # The i variable will iterate over the elements of the List and contains each element in each iteration.
4. print(i)

**Output:**

John

David

James

Jonathan

**Adding elements to the list**

Python provides append() function which is used to add an element to the list. However, the append() function can only add value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

1. #Declaring the empty list
2. l =[]
3. #Number of elements will be entered by the user
4. n = int(input("Enter the number of elements in the list:"))
5. # for loop to take the input
6. for i in range(0,n):
7. # The input is taken from the user and added to the list as the item
8. l.append(input("Enter the item:"))
9. print("printing the list items..")
10. # traversal loop to print the list items
11. for i in l:
12. print(i, end = "  ")

**Output:**

Enter the number of elements in the list:5

Enter the item:25

Enter the item:46

Enter the item:12

Enter the item:75

Enter the item:42

printing the list items

25 46 12 75 42

**Removing elements from the list**

Python provides the **remove()** function which is used to remove the element from the list. Consider the following example to understand this concept.

**Example -**

1. list = [0,1,2,3,4]
2. print("printing original list: ");
3. for i in list:
4. print(i,end=" ")
5. list.remove(2)
6. print("\nprinting the list after the removal of first element...")
7. for i in list:
8. print(i,end=" ")

**Output:**

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

0 1 3 4

**Python List Built-in functions**

Python provides the following built-in functions, which can be used with the lists.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Function** | **Description** | **Example** |
| 1 | cmp(list1, list2) | It compares the elements of both the lists. | This method is not used in the Python 3 and the above versions. |
| 2 | len(list) | It is used to calculate the length of the list. | L1 = [1,2,3,4,5,6,7,8]  print(len(L1))  8 |
| 3 | max(list) | It returns the maximum element of the list. | L1 = [12,34,26,48,72]  print(max(L1))  72 |
| 4 | min(list) | It returns the minimum element of the list. | L1 = [12,34,26,48,72]  print(min(L1))  12 |
| 5 | list(seq) | It converts any sequence to the list. | str = "Johnson"  s = list(str)  print(type(s))  <class list> |

Let's have a look at the few list examples.

**Example: 1-** Write the program to remove the duplicate element of the list.

1. list1 = [1,2,2,3,55,98,65,65,13,29]
2. # Declare an empty list that will store unique values
3. list2 = []
4. for i in list1:
5. if i not in list2:
6. list2.append(i)
7. print(list2)

**Output:**

[1, 2, 3, 55, 98, 65, 13, 29]

**Example:2-** Write a program to find the sum of the element in the list.

1. list1 = [3,4,5,9,10,12,24]
2. sum = 0
3. for i in list1:
4. sum = sum+i
5. print("The sum is:",sum)

**Output:**

The sum is: 67

**Example: 3-** Write the program to find the lists consist of at least one common element.

1. list1 = [1,2,3,4,5,6]
2. list2 = [7,8,9,2,10]
3. for x in list1:
4. for y in list2:
5. if x == y:
6. print("The common element is:",x)

**Output:**

The common element is: 2

**Python Tuples**

A collection of ordered and immutable objects is known as a tuple. Tuples and lists are similar as they both are sequences. Though, tuples and lists are different because we cannot modify tuples, although we can modify lists after creating them, and also because we use parentheses to create tuples while we use square brackets to create lists.

Placing different values separated by commas and enclosed in parentheses forms a tuple. For instance,

**Example**

1. tuple\_1 = ("Python", "tuples", "immutable", "object")
2. tuple\_2 = (23, 42, 12, 53, 64)
3. tuple\_3 = "Python", "Tuples", "Ordered", "Collection"

We represent an empty tuple by two parentheses enclosing nothing.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. Empty\_tuple = ()

We need to add a comma after the element to create a tuple of a single element.

1. Tuple\_1 = (50,)

Tuple indices begin at 0, and similar to strings, we can slice them, concatenate them, and perform other operations.

**Creating a Tuple**

All the objects (elements) must be enclosed in parenthesis (), each separated by a comma, to form a tuple. Although using parenthesis is not required, it is recommended to do so.

Whatever the number of objects, even of various data types, can be included in a tuple (dictionary, string, float, list, etc.).

**Code**

1. # Python program to show how to create a tuple
3. # Creating an empty tuple
4. empty\_tuple = ()
5. print("Empty tuple: ", empty\_tuple)
7. # Creating tuple having integers
8. int\_tuple = (4, 6, 8, 10, 12, 14)
9. print("Tuple with integers: ", int\_tuple)
11. # Creating a tuple having objects of different data types
12. mixed\_tuple = (4, "Python", 9.3)
13. print("Tuple with different data types: ", mixed\_tuple)
15. # Creating a nested tuple
16. nested\_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
17. print("A nested tuple: ", nested\_tuple)

**Output:**

Empty tuple: ()

Tuple with integers: (4, 6, 8, 10, 12, 14)

Tuple with different data types: (4, 'Python', 9.3)

A nested tuple: ('Python', {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))

Parentheses are not mandated to build tuples. Tuple packing is the term for this.

**Code**

1. # Python program to create a tuple without using parentheses
3. # Creating a tuple
4. tuple\_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
6. # displaying the tuple created
7. print(tuple\_)
9. # Checking the data type of object tuple\_
10. print( type(tuple\_) )
12. # trying to modify tuple\_
13. try:
14. tuple\_[1] = 4.2
15. except:
16. print( TypeError )

**Output:**

(4, 5.7, 'Tuples', ['Python', 'Tuples'])

<class 'tuple'>

<class 'TypeError'>

It can be challenging to build a tuple with just one element.

Placing just the element in parentheses is not sufficient. It will require a comma after the element to be recognized as a tuple.

**Code**

1. # Python program to show how to create a tuple having a single element
3. single\_tuple = ("Tuple")
4. print( type(single\_tuple) )
6. # Creating a tuple that has only one element
7. single\_tuple = ("Tuple",)
8. print( type(single\_tuple) )
10. # Creating tuple without parentheses
11. single\_tuple = "Tuple",
12. print( type(single\_tuple) )

**Output:**

<class 'str'>

<class 'tuple'>

<class 'tuple'>

**Accessing Tuple Elements**

We can access the objects of a tuple in a variety of ways.

**Indexing**

To access an object of a tuple, we can use the index operator [], where indexing in the tuple starts from 0.

A tuple with 5 items will have indices ranging from 0 to 4. An IndexError will be raised if we try to access an index from the tuple that is outside the range of the tuple index. In this case, an index above 4 will be out of range.

We cannot give an index of a floating data type or other kinds because the index in Python must be an integer. TypeError will appear as a result if we give a floating index.

The example below illustrates how indexing is performed in nested tuples to access elements.

**Code**

1. # Python program to show how to access tuple elements
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Collection")
6. print(tuple\_[0])
7. print(tuple\_[1])
8. # trying to access element index more than the length of a tuple
9. try:
10. print(tuple\_[5])
11. except Exception as e:
12. print(e)
13. # trying to access elements through the index of floating data type
14. try:
15. print(tuple\_[1.0])
16. except Exception as e:
17. print(e)
19. # Creating a nested tuple
20. nested\_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))
22. # Accessing the index of a nested tuple
23. print(nested\_tuple[0][3])
24. print(nested\_tuple[1][1])

**Output:**

Python

Tuple

tuple index out of range

tuple indices must be integers or slices, not float

l

6

**Negative Indexing**

Python's sequence objects support negative indexing.

The last item of the collection is represented by -1, the second last item by -2, and so on.

**Code**

1. # Python program to show how negative indexing works in Python tuples
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Collection")
6. # Printing elements using negative indices
7. print("Element at -1 index: ", tuple\_[-1])
9. print("Elements between -4 and -1 are: ", tuple\_[-4:-1])

**Output:**

Element at -1 index: Collection

Elements between -4 and -1 are: ('Python', 'Tuple', 'Ordered')

**Slicing**

We can use a slicing operator, a colon (:), to access a range of tuple elements.

**Code**

1. # Python program to show how slicing works in Python tuples
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
6. # Using slicing to access elements of the tuple
7. print("Elements between indices 1 and 3: ", tuple\_[1:3])
9. # Using negative indexing in slicing
10. print("Elements between indices 0 and -4: ", tuple\_[:-4])
12. # Printing the entire tuple by using the default start and end values.
13. print("Entire tuple: ", tuple\_[:])

**Output:**

Elements between indices 1 and 3: ('Tuple', 'Ordered')

Elements between indices 0 and -4: ('Python', 'Tuple')

Entire tuple: ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection', 'Objects')

**Deleting a Tuple**

The elements of a tuple cannot be changed, as was already said. Therefore, we are unable to eliminate or remove elements of a tuple.

However, the keyword del makes it feasible to delete a tuple completely.

**Code**

1. # Python program to show how to delete elements of a Python tuple
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
6. # Deleting a particular element of the tuple
7. try:
8. del tuple\_[3]
9. print(tuple\_)
10. except Exception as e:
11. print(e)
13. # Deleting the variable from the global space of the program
14. del tuple\_
16. # Trying accessing the tuple after deleting it
17. try:
18. print(tuple\_)
19. except Exception as e:
20. print(e)

**Output:**

'tuple' object doesn't support item deletion

name 'tuple\_' is not defined

**Repetition Tuples in Python**

**Code**

1. # Python program to show repetition in tuples
3. tuple\_ = ('Python',"Tuples")
4. print("Original tuple is: ", tuple\_)
6. # Repeting the tuple elements
7. tuple\_ = tuple\_ \* 3
8. print("New tuple is: ", tuple\_)

**Output:**

Original tuple is: ('Python', 'Tuples')

New tuple is: ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')

**Tuple Methods**

Tuple does not provide methods to add or delete elements, and there are only the following two choices.

Examples of these methods are given below.

**Code**

1. # Python program to show how to tuple methods (.index() and .count()) work
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")
6. # Counting the occurrence of an element of the tuple using the count() method
7. print(tuple\_.count('Ordered'))
9. # Getting the index of an element using the index() method
10. print(tuple\_.index('Ordered')) # This method returns index of the first occurrence of the element

**Output:**

2

2

**Tuple Membership Test**

Using the in keyword, we can determine whether an item is present in the given tuple or not.

**Code**

1. # Python program to show how to perform membership test for tuples
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")
6. # In operator
7. print('Tuple' in tuple\_)
8. print('Items' in tuple\_)
10. # Not in operator
11. print('Immutable' not in tuple\_)
12. print('Items' not in tuple\_)

**Output:**

True

False

False

True

**Iterating Through a Tuple**

We can use a for loop to iterate through each element of a tuple.

**Code**

1. # Python program to show how to iterate over tuple elements
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable")
6. # Iterating over tuple elements using a for loop
7. for item in tuple\_:
8. print(item)

**Output:**

Python

Tuple

Ordered

Immutable

**Changing a Tuple**

Tuples, as opposed to lists, are immutable objects.

This implies that after a tuple's elements have been specified, we cannot modify them. However, we can modify the nested elements of an element if the element itself is a mutable data type like a list.

A tuple can be assigned to many values (reassignment).

**Code**

1. # Python program to show that Python tuples are immutable objects
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", [1,2,3,4])
6. # Trying to change the element at index 2
7. try:
8. tuple\_[2] = "Items"
9. print(tuple\_)
10. except Exception as e:
11. print( e )
13. # But inside a tuple, we can change elements of a mutable object
14. tuple\_[-1][2] = 10
15. print(tuple\_)
17. # Changing the whole tuple
18. tuple\_ = ("Python", "Items")
19. print(tuple\_)

**Output:**

'tuple' object does not support item assignment

('Python', 'Tuple', 'Ordered', 'Immutable', [1, 2, 10, 4])

('Python', 'Items')

To merge multiple tuples, we can use the + operator. Concatenation is the term for this.

Using the \* operator, we may also repeat a tuple's elements for a specified number of times. This is already shown above.

The results of the operations + and \* are new tuples.

**Code**

1. # Python program to show how to concatenate tuples
3. # Creating a tuple
4. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable")
6. # Adding a tuple to the tuple\_
7. print(tuple\_ + (4, 5, 6))

**Output:**

('Python', 'Tuple', 'Ordered', 'Immutable', 4, 5, 6)

**Advantages of Tuple over List**

Tuples and lists are employed in similar contexts because of how similar they are. A tuple implementation has several benefits over a list, though. The following are a few of the primary benefits:

* We generally employ lists for homogeneous data types and tuples for heterogeneous data types.
* Tuple iteration is quicker than list iteration because tuples are immutable. There is such a modest performance improvement.
* Tuples with immutable components can function as the key for a Python dictionary object. This feature is not feasible with lists.
* Collecting data in a tuple will ensure that it stays write-protected if it never changes.

**Python Set**

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

**Creating a set**

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

**Example 1: Using curly braces**

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. print(Days)
3. print(type(Days))
4. print("looping through the set elements ... ")
5. for i in Days:
6. print(i)

**Output:**

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

**Example 2: Using set() method**

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. print(Days)
3. print(type(Days))
4. print("looping through the set elements ... ")
5. for i in Days:
6. print(i)

**Output:**

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

It can contain any type of element such as integer, float, tuple etc. But mutable elements (list, dictionary, set) can't be a member of set. Consider the following example.

1. # Creating a set which have immutable elements
2. set1 = {1,2,3, "JavaTpoint", 20.5, 14}
3. print(type(set1))
4. #Creating a set which have mutable element
5. set2 = {1,2,3,["Javatpoint",4]}
6. print(type(set2))

**Output:**

<class 'set'>

Traceback (most recent call last)

<ipython-input-5-9605bb6fbc68> in <module>

4

5 #Creating a set which holds mutable elements

----> 6 set2 = {1,2,3,["Javatpoint",4]}

7 print(type(set2))

TypeError: unhashable type: 'list'

In the above code, we have created two sets, the set **set1** have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.

1. # Empty curly braces will create dictionary
2. set3 = {}
3. print(type(set3))
5. # Empty set using set() function
6. set4 = set()
7. print(type(set4))

**Output:**

<class 'dict'>

<class 'set'>

Let's see what happened if we provide the duplicate element to the set.

1. set5 = {1,2,4,4,5,8,9,9,10}
2. print("Return set with unique elements:",set5)

**Output:**

Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}

In the above code, we can see that **set5** consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

**Adding items to the set**

Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

**Example: 1 - Using add() method**

1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nAdding other months to the set...");
5. Months.add("July");
6. Months.add ("August");
7. print("\nPrinting the modified set...");
8. print(Months)
9. print("\nlooping through the set elements ... ")
10. for i in Months:
11. print(i)

**Output:**

printing the original set ...

{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...

{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...

February

July

May

April

March

August

June

January

To add more than one item in the set, Python provides the **update()** method. It accepts iterable as an argument.

Consider the following example.

**Example - 2 Using update() function**

1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nupdating the original set ... ")
5. Months.update(["July","August","September","October"]);
6. print("\nprinting the modified set ... ")
7. print(Months);

**Output:**

printing the original set ...

{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

**Removing items from the set**

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these function, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Consider the following example.

**Example-1 Using discard() method**

1. months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nRemoving some months from the set...");
5. months.discard("January");
6. months.discard("May");
7. print("\nPrinting the modified set...");
8. print(months)
9. print("\nlooping through the set elements ... ")
10. for i in months:
11. print(i)

**Output:**

printing the original set ...

{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...

{'February', 'March', 'April', 'June'}

looping through the set elements ...

February

March

April

June

Python provides also the **remove()** method to remove the item from the set. Consider the following example to remove the items using **remove()** method.

**Example-2 Using remove() function**

1. months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nRemoving some months from the set...");
5. months.remove("January");
6. months.remove("May");
7. print("\nPrinting the modified set...");
8. print(months)

**Output:**

printing the original set ...

{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.

Consider the following example to remove the item from the set using pop() method.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nRemoving some months from the set...");
5. Months.pop();
6. Months.pop();
7. print("\nPrinting the modified set...");
8. print(Months)

**Output:**

printing the original set ...

{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...

{'May', 'April', 'February', 'March'}

In the above code, the last element of the **Month** set is **March** but the pop() method removed the **June and January** because the set is unordered and the pop() method could not determine the last element of the set.

Python provides the clear() method to remove all the items from the set.

Consider the following example.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nRemoving all the items from the set...");
5. Months.clear()
6. print("\nPrinting the modified set...")
7. print(Months)

**Output:**

printing the original set ...

{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...

set()

**Difference between discard() and remove()**

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

Consider the following example.

**Example-**

1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nRemoving items through discard() method...");
5. Months.discard("Feb"); #will not give an error although the key feb is not available in the set
6. print("\nprinting the modified set...")
7. print(Months)
8. print("\nRemoving items through remove() method...");
9. Months.remove("Jan") #will give an error as the key jan is not available in the set.
10. print("\nPrinting the modified set...")
11. print(Months)

**Output:**

printing the original set ...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...

Traceback (most recent call last):

File "set.py", line 9, in

Months.remove("Jan")

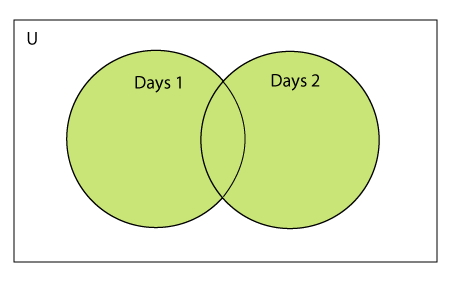
KeyError: 'Jan'

**Python Set Operations**

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

**Union of two Sets**

The union of two sets is calculated by using the pipe (|) operator. The union of the two sets contains all the items that are present in both the sets.



Consider the following example to calculate the union of two sets.

**Example 1: using union | operator**

1. Days1 = {"Monday","Tuesday","Wednesday","Thursday", "Sunday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. print(Days1|Days2) #printing the union of the sets

**Output:**

{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

**Example 2: using union() method**

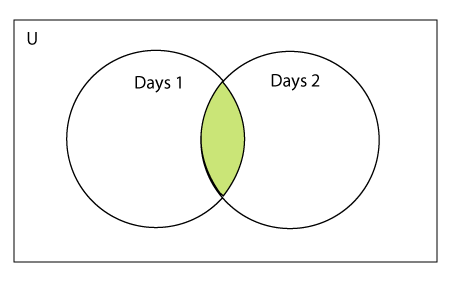
1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. print(Days1.union(Days2)) #printing the union of the sets

**Output:**

{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}

**Intersection of two sets**

The intersection of two sets can be performed by the **and &** operator or the **intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.



Consider the following example.

**Example 1: Using & operator**

1. Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday","Tuesday","Sunday", "Friday"}
3. print(Days1&Days2) #prints the intersection of the two sets

**Output:**

{'Monday', 'Tuesday'}

**Example 2: Using intersection() method**

1. set1 = {"Devansh","John", "David", "Martin"}
2. set2 = {"Steve", "Milan", "David", "Martin"}
3. print(set1.intersection(set2)) #prints the intersection of the two sets

**Output:**

{'Martin', 'David'}

**Example 3:**

1. set1 = {1,2,3,4,5,6,7}
2. set2 = {1,2,20,32,5,9}
3. set3 = set1.intersection(set2)
4. print(set3)

**Output:**

{1,2,5}

**The intersection\_update() method**

The **intersection\_update()** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The **intersection\_update()** method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

Consider the following example.

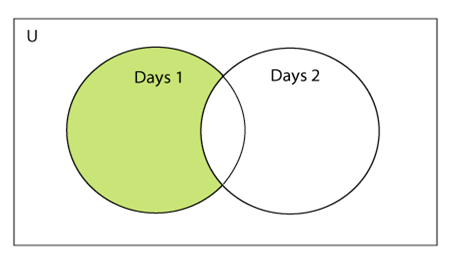
1. a = {"Devansh", "bob", "castle"}
2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
5. a.intersection\_update(b, c)
7. print(a)

**Output:**

{'castle'}

**Difference between the two sets**

The difference of two sets can be calculated by using the subtraction (-) operator or **intersection()** method. Suppose there are two sets A and B, and the difference is A-B that denotes the resulting set will be obtained that element of A, which is not present in the set B.



Consider the following example.

**Example 1 : Using subtraction ( - ) operator**

1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. print(Days1-Days2) #{"Wednesday", "Thursday" will be printed}

**Output:**

{'Thursday', 'Wednesday'}

**Example 2 : Using difference() method**

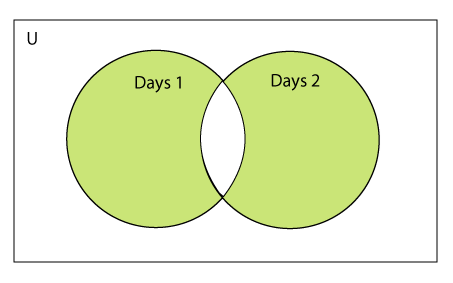
1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. print(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2

**Output:**

{'Thursday', 'Wednesday'}

**Symmetric Difference of two sets**

The symmetric difference of two sets is calculated by ^ operator or **symmetric\_difference()** method. Symmetric difference of sets, it removes that element which is present in both sets. Consider the following example:



**Example - 1: Using ^ operator**

1. a = {1,2,3,4,5,6}
2. b = {1,2,9,8,10}
3. c = a^b
4. print(c)

**Output:**

{3, 4, 5, 6, 8, 9, 10}

**Example - 2: Using symmetric\_difference() method**

1. a = {1,2,3,4,5,6}
2. b = {1,2,9,8,10}
3. c = a.symmetric\_difference(b)
4. print(c)

**Output:**

{3, 4, 5, 6, 8, 9, 10}

**Set comparisons**

Python allows us to use the comparison operators i.e., <, >, <=, >= , == with the sets by using which we can check whether a set is a subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

Consider the following example.

1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday"}
3. Days3 = {"Monday", "Tuesday", "Friday"}
5. #Days1 is the superset of Days2 hence it will print true.
6. print (Days1>Days2)
8. #prints false since Days1 is not the subset of Days2
9. print (Days1<Days2)
11. #prints false since Days2 and Days3 are not equivalent
12. print (Days2 == Days3)

**Output:**

True

False

False

**FrozenSets**

The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set cannot be changed and therefore it can be used as a key in the dictionary.

The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

1. Frozenset = frozenset([1,2,3,4,5])
2. print(type(Frozenset))
3. print("\nprinting the content of frozen set...")
4. for i in Frozenset:
5. print(i);
6. Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation

**Output:**

<class 'frozenset'>

printing the content of frozen set...

1

2

3

4

5

Traceback (most recent call last):

File "set.py", line 6, in <module>

Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation

AttributeError: 'frozenset' object has no attribute 'add'

**Frozenset for the dictionary**

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Consider the following example.

1. Dictionary = {"Name":"John", "Country":"USA", "ID":101}
2. print(type(Dictionary))
3. Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
4. print(type(Frozenset))
5. for i in Frozenset:
6. print(i)

**Output:**

<class 'dict'>

<class 'frozenset'>

Name

Country

ID

**Set Programming Example**

**Example - 1:** Write a program to remove the given number from the set.

1. my\_set = {1,2,3,4,5,6,12,24}
2. n = int(input("Enter the number you want to remove"))
3. my\_set.discard(n)
4. print("After Removing:",my\_set)

**Output:**

Enter the number you want to remove:12

After Removing: {1, 2, 3, 4, 5, 6, 24}

**Example - 2:** Write a program to add multiple elements to the set.

1. set1 = set([1,2,4,"John","CS"])
2. set1.update(["Apple","Mango","Grapes"])
3. print(set1)

**Output:**

{1, 2, 4, 'Apple', 'John', 'CS', 'Mango', 'Grapes'}

**Example - 3:** Write a program to find the union between two set.

1. set1 = set(["Peter","Joseph", 65,59,96])
2. set2  = set(["Peter",1,2,"Joseph"])
3. set3 = set1.union(set2)
4. print(set3)

**Output:**

{96, 65, 2, 'Joseph', 1, 'Peter', 59}

**Example- 4:** Write a program to find the intersection between two sets.

1. set1 = {23,44,56,67,90,45,"Javatpoint"}
2. set2 = {13,23,56,76,"Sachin"}
3. set3 = set1.intersection(set2)
4. print(set3)

**Output:**

{56, 23}

**Example - 5:** Write the program to add element to the frozenset.

1. set1 = {23,44,56,67,90,45,"Javatpoint"}
2. set2 = {13,23,56,76,"Sachin"}
3. set3 = set1.intersection(set2)
4. print(set3)

**Output:**

TypeError: 'frozenset' object does not support item assignment

Above code raised an error because frozensets are immutable and can't be changed after creation.

**Example - 6:** Write the program to find the issuperset, issubset and superset.

1. set1 = set(["Peter","James","Camroon","Ricky","Donald"])
2. set2 = set(["Camroon","Washington","Peter"])
3. set3 = set(["Peter"])
5. issubset = set1 >= set2
6. print(issubset)
7. issuperset = set1 <= set2
8. print(issuperset)
9. issubset = set3 <= set2
10. print(issubset)
11. issuperset = set2 >= set3
12. print(issuperset)

**Output:**

False

False

True

True

**Python Built-in set methods**

Python contains the following methods to be used with the sets.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [add(item)](https://www.javatpoint.com/python-set-add-method) | It adds an item to the set. It has no effect if the item is already present in the set. |
| 2 | clear() | It deletes all the items from the set. |
| 3 | copy() | It returns a shallow copy of the set. |
| 4 | difference\_update(....) | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | [discard(item)](https://www.javatpoint.com/python-set-discard-method) | It removes the specified item from the set. |
| 6 | intersection() | It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified). |
| 7 | intersection\_update(....) | It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified). |
| 8 | Isdisjoint(....) | Return True if two sets have a null intersection. |
| 9 | Issubset(....) | Report whether another set contains this set. |
| 10 | Issuperset(....) | Report whether this set contains another set. |
| 11 | [pop()](https://www.javatpoint.com/python-set-pop-method) | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | [remove(item)](https://www.javatpoint.com/python-set-remove-method) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | symmetric\_difference(....) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | symmetric\_difference\_update(....) | Update a set with the symmetric difference of itself and another. |
| 15 | union(....) | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | update() | Update a set with the union of itself and others. |

**Python Dictionary**

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

* Keys must be a single element
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

**Creating the dictionary**

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

**Syntax:**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. Dict = {"Name": "Tom", "Age": 22}

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

Let's see an example to create a dictionary and print its content.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)

**Output**

<class 'dict'>

Printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

1. # Creating an empty Dictionary
2. Dict = {}
3. print("Empty Dictionary: ")
4. print(Dict)
6. # Creating a Dictionary
7. # with dict() method
8. Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
9. print("\nCreate Dictionary by using  dict(): ")
10. print(Dict)
12. # Creating a Dictionary
13. # with each item as a Pair
14. Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
15. print("\nDictionary with each item as a pair: ")
16. print(Dict)

**Output:**

Empty Dictionary:

{}

Create Dictionary by using dict():

{1: 'Java', 2: 'T', 3: 'Point'}

Dictionary with each item as a pair:

{1: 'Devansh', 2: 'Sharma'}

**Accessing the dictionary values**

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print("Name : %s" %Employee["Name"])
5. print("Age : %d" %Employee["Age"])
6. print("Salary : %d" %Employee["salary"])
7. print("Company : %s" %Employee["Company"])

**Output:**

<class 'dict'>

printing Employee data ....

Name : John

Age : 29

Salary : 25000

Company : GOOGLE

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

**Adding dictionary values**

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

Let's see an example to update the dictionary values.

**Example - 1:**

1. # Creating an empty Dictionary
2. Dict = {}
3. print("Empty Dictionary: ")
4. print(Dict)
6. # Adding elements to dictionary one at a time
7. Dict[0] = 'Peter'
8. Dict[2] = 'Joseph'
9. Dict[3] = 'Ricky'
10. print("\nDictionary after adding 3 elements: ")
11. print(Dict)
13. # Adding set of values
14. # with a single Key
15. # The Emp\_ages doesn't exist to dictionary
16. Dict['Emp\_ages'] = 20, 33, 24
17. print("\nDictionary after adding 3 elements: ")
18. print(Dict)
20. # Updating existing Key's Value
21. Dict[3] = 'JavaTpoint'
22. print("\nUpdated key value: ")
23. print(Dict)

**Output:**

Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

**Example - 2:**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. print("printing the new data");
11. print(Employee)

**Output:**

Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

**Deleting elements using del keyword**

The items of the dictionary can be deleted by using the **del** keyword as given below.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Deleting some of the employee data")
6. del Employee["Name"]
7. del Employee["Company"]
8. print("printing the modified information ")
9. print(Employee)
10. print("Deleting the dictionary: Employee");
11. del Employee
12. print("Lets try to print it again ");
13. print(Employee)

**Output:**

<class 'dict'>

printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Deleting some of the employee data

printing the modified information

{'Age': 29, 'salary': 25000}

Deleting the dictionary: Employee

Lets try to print it again

NameError: name 'Employee' is not defined

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

* **Using pop() method**

The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

1. # Creating a Dictionary
2. Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
3. # Deleting a key
4. # using pop() method
5. pop\_ele = Dict.pop(3)
6. print(Dict)

**Output:**

{1: 'JavaTpoint', 2: 'Peter'}

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

**Iterating Dictionary**

A dictionary can be iterated using for loop as given below.

**Example 1**

**# for loop to print all the keys of a dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. for x in Employee:
3. print(x)

**Output:**

Name

Age

salary

Company

**Example 2**

**#for loop to print all the values of the dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. for x in Employee:
3. print(Employee[x])

**Output:**

John

29

25000

GOOGLE

**Example - 3**

**#for loop to print the values of the dictionary by using values() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. for x in Employee.values():
3. print(x)

**Output:**

John

29

25000

GOOGLE

**Example 4**

**#for loop to print the items of the dictionary by using items() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. for x in Employee.items():
3. print(x)

**Output:**

('Name', 'John')

('Age', 29)

('salary', 25000)

('Company', 'GOOGLE')

**Properties of Dictionary keys**

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

1. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
2. for x,y in Employee.items():
3. print(x,y)

**Output:**

Name John

Age 29

Salary 25000

Company GOOGLE

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
2. for x,y in Employee.items():
3. print(x,y)

**Output:**

Traceback (most recent call last):

File "dictionary.py", line 1, in

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

TypeError: unhashable type: 'list'

**Built-in Dictionary functions**

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(dict1, dict2) | It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false. |
| 2 | len(dict) | It is used to calculate the length of the dictionary. |
| 3 | str(dict) | It converts the dictionary into the printable string representation. |
| 4 | type(variable) | It is used to print the type of the passed variable. |

**Built-in Dictionary methods**

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [dic.clear()](https://www.javatpoint.com/python-dictionary-clear-method) | It is used to delete all the items of the dictionary. |
| 2 | [dict.copy()](https://www.javatpoint.com/python-dictionary-copy-method) | It returns a shallow copy of the dictionary. |
| 3 | [dict.fromkeys(iterable, value = None, /)](https://www.javatpoint.com/python-dictionary-fromkeys-method) | Create a new dictionary from the iterable with the values equal to value. |
| 4 | [dict.get(key, default = "None")](https://www.javatpoint.com/python-dictionary-get-method) | It is used to get the value specified for the passed key. |
| 5 | dict.has\_key(key) | It returns true if the dictionary contains the specified key. |
| 6 | [dict.items()](https://www.javatpoint.com/python-dictionary-items-method) | It returns all the key-value pairs as a tuple. |
| 7 | [dict.keys()](https://www.javatpoint.com/python-dictionary-keys-method) | It returns all the keys of the dictionary. |
| 8 | [dict.setdefault(key,default= "None")](https://www.javatpoint.com/python-dictionary-setdefault-method) | It is used to set the key to the default value if the key is not specified in the dictionary |
| 9 | [dict.update(dict2)](https://www.javatpoint.com/python-dictionary-update-method) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 10 | [dict.values()](https://www.javatpoint.com/python-dictionary-values-method) | It returns all the values of the dictionary. |
| 11 | [len()](https://www.javatpoint.com/python-dictionary-len-method) |  |
| 12 | [popItem()](https://www.javatpoint.com/python-dictionary-popitem-method) |  |
| 13 | [pop()](https://www.javatpoint.com/python-dictionary-pop-method) |  |
| 14 | [count()](https://www.javatpoint.com/python-dictionary-count-method) |  |
| 15 | [index()](https://www.javatpoint.com/python-dictionary-index-method) |  |

**Python Functions**

This tutorial will learn about the basics of Python functions, including what they are, their syntax, their main components, return keywords, and major types. We will also see examples of how to define a Python function.

**What are Python Functions?**

A function is a collection of related assertions that performs a mathematical, analytical, or evaluative operation. Python functions are simple to define and essential to intermediate-level programming. The exact criteria hold to function names as they do to variable names. The goal is to group up certain often performed actions and define a function. Rather than rewriting the same code block over and over for varied input variables, we may call the function and repurpose the code included within it with different variables.

The functions are broad of two types, user-defined and built-in functions. It aids in keeping the software succinct, non-repetitive, and well-organized.

**Advantages of Functions in Python**

Python functions have the following benefits.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

* By including functions, we can prevent repeating the same code block repeatedly in a program.
* Python functions, once defined, can be called many times and from anywhere in a program.
* If our Python program is large, it can be separated into numerous functions which is simple to track.
* The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

However, calling functions has always been overhead in a Python program.

**Syntax of Python Function**

**Code**

1. def name\_of\_function( parameters ):
2. """This is a docstring"""
3. # code block

The following elements make up define a function, as seen above.

* The beginning of a function header is indicated by a keyword called def.
* name\_of\_function is the function's name that we can use to separate it from others. We will use this name to call the function later in the program. The same criteria apply to naming functions as to naming variables in Python.
* We pass arguments to the defined function using parameters. They are optional, though.
* The function header is terminated by a colon (:).
* We can use a documentation string called docstring in the short form to explain the purpose of the function.
* The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
* We can use a return expression to return a value from a defined function.

**Example of a User-Defined Function**

We will define a function that when called will return the square of the number passed to it as an argument.

**Code**

1. def square( num ):
2. """
3. This function computes the square of the number.
4. """
5. return num\*\*2
6. object\_ = square(9)
7. print( "The square of the number is: ", object\_ )

**Output:**

The square of the number is: 81

**Calling a Function**

A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.

After a function's fundamental framework is complete, we can call it from anywhere in the program. The following is an example of how to use the a\_function function.

**Code**

1. # Defining a function
2. def a\_function( string ):
3. "This prints the value of length of string"
4. return len(string)
6. # Calling the function we defined
7. print( "Length of the string Functions is: ", a\_function( "Functions" ) )
8. print( "Length of the string Python is: ", a\_function( "Python" ) )

**Output:**

Length of the string Functions is: 9

Length of the string Python is: 6

**Pass by Reference vs. Value**

In the Python programming language, all arguments are supplied by reference. It implies that if we modify the value of an argument within a function, the change is also reflected in the calling function. For instance,

**Code**

1. # defining the function
2. def square( my\_list ):
3. '''''This function will find the square of items in list'''
4. squares = []
5. for l in my\_list:
6. squares.append( l\*\*2 )
7. return squares
9. # calling the defined function
10. list\_ = [45, 52, 13];
11. result = square( list\_ )
12. print( "Squares of the list is: ", result )

**Output:**

Squares of the list is: [2025, 2704, 169]

**Function Arguments**

The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

**Default Arguments**

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. def function( num1, num2 = 40 ):
4. print("num1 is: ", num1)
5. print("num2 is: ", num2)

8. # Calling the function and passing only one argument
9. print( "Passing one argument" )
10. function(10)
12. # Now giving two arguments to the function
13. print( "Passing two arguments" )
14. function(10,30)

**Output:**

Passing one argument

num1 is: 10

num2 is: 40

Passing two arguments

num1 is: 10

num2 is: 30

**Keyword Arguments**

The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.

Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order. The function() method can also be called with keywords in the following manner:

**Code**

1. # Python code to demonstrate the use of keyword arguments
3. # Defining a function
4. def function( num1, num2 ):
5. print("num1 is: ", num1)
6. print("num2 is: ", num2)
8. # Calling function and passing arguments without using keyword
9. print( "Without using keyword" )
10. function( 50, 30)
12. # Calling function and passing arguments using keyword
13. print( "With using keyword" )
14. function( num2 = 50, num1 = 30)

**Output:**

Without using keyword

num1 is: 50

num2 is: 30

With using keyword

num1 is: 30

num2 is: 50

**Required Arguments**

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

**Code**

1. # Python code to demonstrate the use of default arguments
3. # Defining a function
4. def function( num1, num2 ):
5. print("num1 is: ", num1)
6. print("num2 is: ", num2)
8. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
9. print( "Passing out of order arguments" )
10. function( 30, 20 )
12. # Calling function and passing only one argument
13. print( "Passing only one argument" )
14. try:
15. function( 30 )
16. except:
17. print( "Function needs two positional arguments" )

**Output:**

Passing out of order arguments

num1 is: 30

num2 is: 20

Passing only one argument

Function needs two positional arguments

**Variable-Length Arguments**

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

1. **\*args -**These are Non-Keyword Arguments
2. **\*\*kwargs -** These are Keyword Arguments.

Here is an example to clarify Variable length arguments

**Code**

1. # Python code to demonstrate the use of variable-length arguments
3. # Defining a function
4. def function( \*args\_list ):
5. ans = []
6. for l in args\_list:
7. ans.append( l.upper() )
8. return ans
9. # Passing args arguments
10. object = function('Python', 'Functions', 'tutorial')
11. print( object )
13. # defining a function
14. def function( \*\*kargs\_list ):
15. ans = []
16. for key, value in kargs\_list.items():
17. ans.append([key, value])
18. return ans
19. # Paasing kwargs arguments
20. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
21. print(object)

**Output:**

['PYTHON', 'FUNCTIONS', 'TUTORIAL']

[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]

**return Statement**

We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

**Syntax:**

1. return < expression to be returned as output >

An argument, a statement, or a value can be used in the return statement, which is given as output when a specific task or function is completed. If we do not write a return statement, then None object is returned by a defined function.

Here is an example of a return statement in Python functions.

**Code**

1. # Python code to demonstrate the use of return statements
3. # Defining a function with return statement
4. def square( num ):
5. return num\*\*2
7. # Calling function and passing arguments.
8. print( "With return statement" )
9. print( square( 39 ) )
11. # Defining a function without return statement
12. def square( num ):
13. num\*\*2
15. # Calling function and passing arguments.
16. print( "Without return statement" )
17. print( square( 39 ) )

**Output:**

With return statement

1521

Without return statement

None

**The Anonymous Functions**

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.

Lambda functions contain their unique local domain, meaning they can only reference variables in their argument list and the global domain name.

Although lambda expressions seem to be a one-line representation of a function, they are not like inline expressions in C and C++, which pass function stack allocations at execution for efficiency concerns.

**Syntax**

Lambda functions have exactly one line in their syntax:

1. lambda [argument1 [,argument2... .argumentn]] : expression

Below is an illustration of how to use the lambda function:

**Code**

1. # Defining a function
2. lambda\_ = lambda argument1, argument2: argument1 + argument2;
4. # Calling the function and passing values
5. print( "Value of the function is : ", lambda\_( 20, 30 ) )
6. print( "Value of the function is : ", lambda\_( 40, 50 ) )

**Output:**

Value of the function is : 50

Value of the function is : 90

**Scope and Lifetime of Variables**

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.

The period of a variable's existence in RAM is referred to as its lifetime. Variables within a function have the same lifespan as the function itself.

When we get out of the function, they are removed. As a result, a function does not retain a variable's value from earlier executions.

Here's a simple example of a variable's scope within a function.

**Code**

1. #defining a function to print a number.
2. def number( ):
3. num = 30
4. print( "Value of num inside the function: ", num)
6. num = 20
7. number()
8. print( "Value of num outside the function:", num)

**Output:**

Value of num inside the function: 30

Value of num outside the function: 20

Here, we can observe that the initial value of num is 20. Even if the function number() modified the value of num to 30, the value of num outside the function remained unchanged.

This is because the variable num within the function is distinct from the variable outside the function (local to the function). Despite their identical variable name, they are 2 distinct variables having distinct scopes.

Variables beyond the function, on the contrary, are accessible within the function. These variables have a global reach.

We can retrieve their values inside the function but cannot alter (change) them. If we declare a variable global using the keyword global, we can also change the variable's value outside the function.

**Python Function within Another Function**

Functions are considered first-class objects in Python. In a programming language, first-class objects are treated the same wherever they are used. They can be used in conditional expressions, as arguments, and saved in built-in data structures. If a programming language handles functions as first-class entities, it is said to implement first-class functions. Python supports the notion of First Class functions.

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope. Inner functions are constructed to cover them from the changes that happen outside the function. Many developers regard this process as encapsulation.

**Code**

1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. def function1():
4. string = 'Python functions tutorial'
6. def function2():
7. print( string )
9. function2()
10. function1()

**Output:**

Python functions tutorial

**Python Built-in Functions**

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

**Python abs() Function**

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

**Python abs() Function Example**

1. #  integer number
2. integer = -20
3. print('Absolute value of -40 is:', abs(integer))
5. #  floating number
6. floating = -20.83
7. print('Absolute value of -40.83 is:', abs(floating))

**Output:**

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Absolute value of -20 is: 20

Absolute value of -20.83 is: 20.83

**Python all() Function**

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

**Python all() Function Example**

1. # all values true
2. k = [1, 3, 4, 6]
3. print(all(k))
5. # all values false
6. k = [0, False]
7. print(all(k))
9. # one false value
10. k = [1, 3, 7, 0]
11. print(all(k))
13. # one true value
14. k = [0, False, 5]
15. print(all(k))
17. # empty iterable
18. k = []
19. print(all(k))

**Output:**

True

False

False

False

True

**Python bin() Function**

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

**Python bin() Function Example**

1. x =  10
2. y =  bin(x)
3. print (y)

**Output:**

0b1010

**Python bool()**

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

**Python bool() Example**

1. test1 = []
2. print(test1,'is',bool(test1))
3. test1 = [0]
4. print(test1,'is',bool(test1))
5. test1 = 0.0
6. print(test1,'is',bool(test1))
7. test1 = None
8. print(test1,'is',bool(test1))
9. test1 = True
10. print(test1,'is',bool(test1))
11. test1 = 'Easy string'
12. print(test1,'is',bool(test1))

**Output:**

[] is False

[0] is True

0.0 is False

None is False

True is True

Easy string is True

**Python bytes()**

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

**Python bytes() Example**

1. string = "Hello World."
2. array = bytes(string, 'utf-8')
3. print(array)

**Output:**

b ' Hello World.'

**Python callable() Function**

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

**Python callable() Function Example**

1. x = 8
2. print(callable(x))

**Output:**

False

**Python compile() Function**

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

**Python compile() Function Example**

1. # compile string source to code
2. code\_str = 'x=5\ny=10\nprint("sum =",x+y)'
3. code = compile(code\_str, 'sum.py', 'exec')
4. print(type(code))
5. exec(code)
6. exec(x)

**Output:**

<class 'code'>

sum = 15

**Python exec() Function**

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

**Python exec() Function Example**

1. x = 8
2. exec('print(x==8)')
3. exec('print(x+4)')

**Output:**

True

12

**Python sum() Function**

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

**Python sum() Function Example**

1. s = sum([1, 2,4 ])
2. print(s)
4. s = sum([1, 2, 4], 10)
5. print(s)

**Output:**

7

17

**Python any() Function**

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

**Python any() Function Example**

1. l = [4, 3, 2, 0]
2. print(any(l))
4. l = [0, False]
5. print(any(l))
7. l = [0, False, 5]
8. print(any(l))
10. l = []
11. print(any(l))

**Output:**

True

False

True

False

**Python ascii() Function**

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

**Python ascii() Function Example**

1. normalText = 'Python is interesting'
2. print(ascii(normalText))
4. otherText = 'Pythön is interesting'
5. print(ascii(otherText))
7. print('Pyth\xf6n is interesting')

**Output:**

'Python is interesting'

'Pyth\xf6n is interesting'

Pythön is interesting

**Python bytearray()**

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

**Python bytearray() Example**

1. string = "Python is a programming language."
3. # string with encoding 'utf-8'
4. arr = bytearray(string, 'utf-8')
5. print(arr)

**Output:**

bytearray(b'Python is a programming language.')

**Python eval() Function**

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

**Python eval() Function Example**

1. x = 8
2. print(eval('x + 1'))

**Output:**

9

**Python float()**

The python **float()** function returns a floating-point number from a number or string.

**Python float() Example**

1. # for integers
2. print(float(9))
4. # for floats
5. print(float(8.19))
7. # for string floats
8. print(float("-24.27"))
10. # for string floats with whitespaces
11. print(float("     -17.19\n"))
13. # string float error
14. print(float("xyz"))

**Output:**

9.0

8.19

-24.27

-17.19

ValueError: could not convert string to float: 'xyz'

**Python format() Function**

The python **format()** function returns a formatted representation of the given value.

**Python format() Function Example**

1. # d, f and b are a type
3. # integer
4. print(format(123, "d"))
6. # float arguments
7. print(format(123.4567898, "f"))
9. # binary format
10. print(format(12, "b"))

**Output:**

123

123.456790

1100

**Python frozenset()**

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

**Python frozenset() Example**

1. # tuple of letters
2. letters = ('m', 'r', 'o', 't', 's')
4. fSet = frozenset(letters)
5. print('Frozen set is:', fSet)
6. print('Empty frozen set is:', frozenset())

**Output:**

Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})

Empty frozen set is: frozenset()

**Python getattr() Function**

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

**Python getattr() Function Example**

1. class Details:
2. age = 22
3. name = "Phill"
5. details = Details()
6. print('The age is:', getattr(details, "age"))
7. print('The age is:', details.age)

**Output:**

The age is: 22

The age is: 22

**Python globals() Function**

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

**Python globals() Function Example**

1. age = 22
3. globals()['age'] = 22
4. print('The age is:', age)

**Output:**

The age is: 22

**Python hasattr() Function**

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

**Python hasattr() Function Example**

1. l = [4, 3, 2, 0]
2. print(any(l))
4. l = [0, False]
5. print(any(l))
7. l = [0, False, 5]
8. print(any(l))
10. l = []
11. print(any(l))

**Output:**

True

False

True

False

**Python iter() Function**

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

**Python iter() Function Example**

1. # list of numbers
2. list = [1,2,3,4,5]
4. listIter = iter(list)
6. # prints '1'
7. print(next(listIter))
9. # prints '2'
10. print(next(listIter))
12. # prints '3'
13. print(next(listIter))
15. # prints '4'
16. print(next(listIter))
18. # prints '5'
19. print(next(listIter))

**Output:**

1

2

3

4

5

**Python len() Function**

The python **len()** function is used to return the length (the number of items) of an object.

**Python len() Function Example**

1. strA = 'Python'
2. print(len(strA))

**Output:**

6

**Python list()**

The python **list()** creates a list in python.

**Python list() Example**

1. # empty list
2. print(list())
4. # string
5. String = 'abcde'
6. print(list(String))
8. # tuple
9. Tuple = (1,2,3,4,5)
10. print(list(Tuple))
11. # list
12. List = [1,2,3,4,5]
13. print(list(List))

**Output:**

[]

['a', 'b', 'c', 'd', 'e']

[1,2,3,4,5]

[1,2,3,4,5]

**Python locals() Function**

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

**Python locals() Function Example**

1. def localsAbsent():
2. return locals()
4. def localsPresent():
5. present = True
6. return locals()
8. print('localsNotPresent:', localsAbsent())
9. print('localsPresent:', localsPresent())

**Output:**

localsAbsent: {}

localsPresent: {'present': True}

**Python map() Function**

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

**Python map() Function Example**

1. def calculateAddition(n):
2. return n+n
4. numbers = (1, 2, 3, 4)
5. result = map(calculateAddition, numbers)
6. print(result)
8. # converting map object to set
9. numbersAddition = set(result)
10. print(numbersAddition)

**Output:**

<map object at 0x7fb04a6bec18>

{8, 2, 4, 6}

**Python memoryview() Function**

The python **memoryview()** function returns a memoryview object of the given argument.

**Python memoryview () Function Example**

1. #A random bytearray
2. randomByteArray = bytearray('ABC', 'utf-8')
4. mv = memoryview(randomByteArray)
6. # access the memory view's zeroth index
7. print(mv[0])
9. # It create byte from memory view
10. print(bytes(mv[0:2]))
12. # It create list from memory view
13. print(list(mv[0:3]))

**Output:**

65

b'AB'

[65, 66, 67]

**Python object()**

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

**Python object() Example**

1. python = object()
3. print(type(python))
4. print(dir(python))

**Output:**

<class 'object'>

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_',

'\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_',

'\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_',

'\_\_str\_\_', '\_\_subclasshook\_\_']

**Python open() Function**

The python **open()** function opens the file and returns a corresponding file object.

**Python open() Function Example**

1. # opens python.text file of the current directory
2. f = open("python.txt")
3. # specifying full path
4. f = open("C:/Python33/README.txt")

**Output:**

Since the mode is omitted, the file is opened in 'r' mode; opens for reading.

**Python chr() Function**

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

**Python chr() Function Example**

1. # Calling function
2. result = chr(102) # It returns string representation of a char
3. result2 = chr(112)
4. # Displaying result
5. print(result)
6. print(result2)
7. # Verify, is it string type?
8. print("is it string type:", type(result) is str)

**Output:**

ValueError: chr() arg not in range(0x110000)

**Python complex()**

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

**Python complex() Example**

1. # Python complex() function example
2. # Calling function
3. a = complex(1) # Passing single parameter
4. b = complex(1,2) # Passing both parameters
5. # Displaying result
6. print(a)
7. print(b)

**Output:**

(1.5+0j)

(1.5+2.2j)

**Python delattr() Function**

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

**Python delattr() Function Example**

1. class Student:
2. id = 101
3. name = "Pranshu"
4. email = "pranshu@abc.com"
5. # Declaring function
6. def getinfo(self):
7. print(self.id, self.name, self.email)
8. s = Student()
9. s.getinfo()
10. delattr(Student,'course') # Removing attribute which is not available
11. s.getinfo() # error: throws an error

**Output:**

101 Pranshu pranshu@abc.com

AttributeError: course

**Python dir() Function**

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named \_\_dir\_\_(), this method will be called and must return the list of attributes. It takes a single object type argument.

**Python dir() Function Example**

1. # Calling function
2. att = dir()
3. # Displaying result
4. print(att)

**Output:**

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_',

'\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

**Python divmod() Function**

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

**Python divmod() Function Example**

1. # Python divmod() function example
2. # Calling function
3. result = divmod(10,2)
4. # Displaying result
5. print(result)

**Output:**

(5, 0)

**Python enumerate() Function**

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

**Python enumerate() Function Example**

1. # Calling function
2. result = enumerate([1,2,3])
3. # Displaying result
4. print(result)
5. print(list(result))

**Output:**

<enumerate object at 0x7ff641093d80>

[(0, 1), (1, 2), (2, 3)]

**Python dict()**

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

* If no argument is passed, it creates an empty dictionary.
* If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
* If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

**Python dict() Example**

1. # Calling function
2. result = dict() # returns an empty dictionary
3. result2 = dict(a=1,b=2)
4. # Displaying result
5. print(result)
6. print(result2)

**Output:**

{}

{'a': 1, 'b': 2}

**Python filter() Function**

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

**Python filter() Function Example**

1. # Python filter() function example
2. def filterdata(x):
3. if x>5:
4. return x
5. # Calling function
6. result = filter(filterdata,(1,2,6))
7. # Displaying result
8. print(list(result))

**Output:**

[6]

**Python hash() Function**

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

**Hashable types:** \* bool \* int \* long \* float \* string \* Unicode \* tuple \* code object.

**Python hash() Function Example**

1. # Calling function
2. result = hash(21) # integer value
3. result2 = hash(22.2) # decimal value
4. # Displaying result
5. print(result)
6. print(result2)

**Output:**

21

461168601842737174

**Python help() Function**

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

**Python help() Function Example**

1. # Calling function
2. info = help() # No argument
3. # Displaying result
4. print(info)

**Output:**

Welcome to Python 3.5's help utility!

**Python min() Function**

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

**Python min() Function Example**

1. # Calling function
2. small = min(2225,325,2025) # returns smallest element
3. small2 = min(1000.25,2025.35,5625.36,10052.50)
4. # Displaying result
5. print(small)
6. print(small2)

**Output:**

325

1000.25

**Python set() Function**

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

**Python set() Function Example**

1. # Calling function
2. result = set() # empty set
3. result2 = set('12')
4. result3 = set('javatpoint')
5. # Displaying result
6. print(result)
7. print(result2)
8. print(result3)

**Output:**

set()

{'1', '2'}

{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}

**Python hex() Function**

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

**Python hex() Function Example**

1. # Calling function
2. result = hex(1)
3. # integer value
4. result2 = hex(342)
5. # Displaying result
6. print(result)
7. print(result2)

**Output:**

0x1

0x156

**Python id() Function**

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

**Python id() Function Example**

1. # Calling function
2. val = id("Javatpoint") # string object
3. val2 = id(1200) # integer object
4. val3 = id([25,336,95,236,92,3225]) # List object
5. # Displaying result
6. print(val)
7. print(val2)
8. print(val3)

**Output:**

139963782059696

139963805666864

139963781994504

**Python setattr() Function**

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

**Python setattr() Function Example**

1. class Student:
2. id = 0
3. name = ""
5. def \_\_init\_\_(self, id, name):
6. self.id = id
7. self.name = name
9. student = Student(102,"Sohan")
10. print(student.id)
11. print(student.name)
12. #print(student.email) product error
13. setattr(student, 'email','sohan@abc.com') # adding new attribute
14. print(student.email)

**Output:**

102

Sohan

sohan@abc.com

**Python slice() Function**

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

**Python slice() Function Example**

1. # Calling function
2. result = slice(5) # returns slice object
3. result2 = slice(0,5,3) # returns slice object
4. # Displaying result
5. print(result)
6. print(result2)

**Output:**

slice(None, 5, None)

slice(0, 5, 3)

**Python sorted() Function**

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

**Python sorted() Function Example**

1. str = "javatpoint" # declaring string
2. # Calling function
3. sorted1 = sorted(str) # sorting string
4. # Displaying result
5. print(sorted1)

**Output:**

['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']

**Python next() Function**

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

**Python next() Function Example**

1. number = iter([256, 32, 82]) # Creating iterator
2. # Calling function
3. item = next(number)
4. # Displaying result
5. print(item)
6. # second item
7. item = next(number)
8. print(item)
9. # third item
10. item = next(number)
11. print(item)

**Output:**

256

32

82

**Python input() Function**

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

**Python input() Function Example**

1. # Calling function
2. val = input("Enter a value: ")
3. # Displaying result
4. print("You entered:",val)

**Output:**

Enter a value: 45

You entered: 45

**Python int() Function**

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

**Python int() Function Example**

1. # Calling function
2. val = int(10) # integer value
3. val2 = int(10.52) # float value
4. val3 = int('10') # string value
5. # Displaying result
6. print("integer values :",val, val2, val3)

**Output:**

integer values : 10 10 10

**Python isinstance() Function**

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

**Python isinstance() function Example**

1. class Student:
2. id = 101
3. name = "John"
4. def \_\_init\_\_(self, id, name):
5. self.id=id
6. self.name=name
8. student = Student(1010,"John")
9. lst = [12,34,5,6,767]
10. # Calling function
11. print(isinstance(student, Student)) # isinstance of Student class
12. print(isinstance(lst, Student))

**Output:**

True

False

**Python oct() Function**

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

**Python oct() function Example**

1. # Calling function
2. val = oct(10)
3. # Displaying result
4. print("Octal value of 10:",val)

**Output:**

Octal value of 10: 0o12

**Python ord() Function**

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

**Python ord() function Example**

1. # Code point of an integer
2. print(ord('8'))
4. # Code point of an alphabet
5. print(ord('R'))
7. # Code point of a character
8. print(ord('&'))

**Output:**

56

82

38

**Python pow() Function**

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

**Python pow() function Example**

1. # positive x, positive y (x\*\*y)
2. print(pow(4, 2))
4. # negative x, positive y
5. print(pow(-4, 2))
7. # positive x, negative y (x\*\*-y)
8. print(pow(4, -2))
10. # negative x, negative y
11. print(pow(-4, -2))

**Output:**

16

16

0.0625

0.0625

**Python print() Function**

The python **print()** function prints the given object to the screen or other standard output devices.

**Python print() function Example**

1. print("Python is programming language.")
3. x = 7
4. # Two objects passed
5. print("x =", x)
7. y = x
8. # Three objects passed
9. print('x =', x, '= y')

**Output:**

Python is programming language.

x = 7

x = 7 = y

**Python range() Function**

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

**Python range() function Example**

1. # empty range
2. print(list(range(0)))
4. # using the range(stop)
5. print(list(range(4)))
7. # using the range(start, stop)
8. print(list(range(1,7 )))

**Output:**

[]

[0, 1, 2, 3]

[1, 2, 3, 4, 5, 6]

**Python reversed() Function**

The python **reversed()** function returns the reversed iterator of the given sequence.

**Python reversed() function Example**

1. # for string
2. String = 'Java'
3. print(list(reversed(String)))
5. # for tuple
6. Tuple = ('J', 'a', 'v', 'a')
7. print(list(reversed(Tuple)))
9. # for range
10. Range = range(8, 12)
11. print(list(reversed(Range)))
13. # for list
14. List = [1, 2, 7, 5]
15. print(list(reversed(List)))

**Output:**

['a', 'v', 'a', 'J']

['a', 'v', 'a', 'J']

[11, 10, 9, 8]

[5, 7, 2, 1]

**Python round() Function**

The python **round()** function rounds off the digits of a number and returns the floating point number.

**Python round() Function Example**

1. #  for integers
2. print(round(10))
4. #  for floating point
5. print(round(10.8))
7. #  even choice
8. print(round(6.6))

**Output:**

10

11

7

**Python issubclass() Function**

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

**Python issubclass() Function Example**

1. class Rectangle:
2. def \_\_init\_\_(rectangleType):
3. print('Rectangle is a ', rectangleType)
5. class Square(Rectangle):
6. def \_\_init\_\_(self):
7. Rectangle.\_\_init\_\_('square')
9. print(issubclass(Square, Rectangle))
10. print(issubclass(Square, list))
11. print(issubclass(Square, (list, Rectangle)))
12. print(issubclass(Rectangle, (list, Rectangle)))

**Output:**

True

False

True

True

**Python str**

The python **str()** converts a specified value into a string.

**Python str() Function Example**

1. str('4')

**Output:**

'4'

**Python tuple() Function**

The python **tuple()** function is used to create a tuple object.

**Python tuple() Function Example**

1. t1 = tuple()
2. print('t1=', t1)
4. # creating a tuple from a list
5. t2 = tuple([1, 6, 9])
6. print('t2=', t2)
8. # creating a tuple from a string
9. t1 = tuple('Java')
10. print('t1=',t1)
12. # creating a tuple from a dictionary
13. t1 = tuple({4: 'four', 5: 'five'})
14. print('t1=',t1)

**Output:**

t1= ()

t2= (1, 6, 9)

t1= ('J', 'a', 'v', 'a')

t1= (4, 5)

**Python type()**

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

**Python type() Function Example**

1. List = [4, 5]
2. print(type(List))
4. Dict = {4: 'four', 5: 'five'}
5. print(type(Dict))
7. class Python:
8. a = 0
10. InstanceOfPython = Python()
11. print(type(InstanceOfPython))

**Output:**

<class 'list'>

<class 'dict'>

<class '\_\_main\_\_.Python'>

**Python vars() function**

The python **vars()** function returns the \_\_dict\_\_ attribute of the given object.

**Python vars() Function Example**

1. class Python:
2. def \_\_init\_\_(self, x = 7, y = 9):
3. self.x = x
4. self.y = y
6. InstanceOfPython = Python()
7. print(vars(InstanceOfPython))

**Output:**

{'y': 9, 'x': 7}

**Python zip() Function**

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

**Python zip() Function Example**

1. numList = [4,5, 6]
2. strList = ['four', 'five', 'six']
4. # No iterables are passed
5. result = zip()
7. # Converting itertor to list
8. resultList = list(result)
9. print(resultList)
11. # Two iterables are passed
12. result = zip(numList, strList)
14. # Converting itertor to set
15. resultSet = set(result)
16. print(resultSet)

**Output:**

[]

{(5, 'five'), (4, 'four'), (6, 'six')}

**Python Lambda Functions**

In this tutorial, we'll study anonymous functions, commonly called lambda functions. We'll understand what they are, how to execute them, and their syntax.

**What are Lambda Functions in Python?**

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. We can also use the lambda keyword in Python to define an unnamed function.

**Syntax of Python Lambda Function**

1. lambda arguments: expression

This function accepts any count of inputs but only evaluates and returns one expression.

Lambda functions can be used whenever function arguments are necessary. In addition to other forms of formulations in functions, it has a variety of applications in certain coding domains. It's important to remember that according to syntax, lambda functions are limited to a single statement.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Example of Lambda Function in Python**

An example of a lambda function that adds 4 to the input number is shown below.

**Code**

1. # Code to demonstrate how we can use a lambda function
2. add = lambda num: num + 4
3. print( add(6) )

**Output:**

10

The lambda function is "lambda num: num+4" in the given programme. The parameter is num, and the computed and returned equation is num \* 4.

There is no label for this function. It generates a function object associated with the "add" identifier. We can now refer to it as a standard function. The lambda statement, "lambda num: num+4", is nearly the same as:

**Code**

1. def add( num ):
2. return num + 4
3. print( add(6) )

**Output:**

10

**What's the Distinction Between Lambda and Def Functions?**

Let's glance at this instance to see how a conventional def defined function differs from a function defined using the lambda keyword. This program calculates the reciprocal of a given number:

**Code**

1. # Python code to show the reciprocal of the given number to highlight the difference between def() and lambda().
2. def reciprocal( num ):
3. return 1 / num
5. lambda\_reciprocal = lambda num: 1 / num
7. # using the function defined by def keyword
8. print( "Def keyword: ", reciprocal(6) )
10. # using the function defined by lambda keyword
11. print( "Lambda keyword: ", lambda\_reciprocal(6) )

**Output:**

Def keyword: 0.16666666666666666

Lambda keyword: 0.16666666666666666

The reciprocal() and lambda\_reciprocal() functions act similarly and as expected in the preceding scenario. Let's take a closer look at the sample above:

Both of these yield the reciprocal of a given number without employing Lambda. However, we wanted to declare a function with the name reciprocal and send a number to it while executing def. We were also required to use the return keyword to provide the output from wherever the function was invoked after being executed.

Using Lambda: Instead of a "return" statement, Lambda definitions always include a statement given at output. The beauty of lambda functions is their convenience. We need not allocate a lambda expression to a variable because we can put it at any place a function is requested.

**Using Lambda Function with filter()**

The filter() method accepts two arguments in Python: a function and an iterable such as a list.

The function is called for every item of the list, and a new iterable or list is returned that holds just those elements that returned True when supplied to the function.

Here's a simple illustration of using the filter() method to return only odd numbers from a list.

**Code**

1. # Code to filter odd numbers from a given list
2. list\_ = [34, 12, 64, 55, 75, 13, 63]
4. odd\_list = list(filter( lambda num: (num % 2 != 0) , list\_ ))
6. print(odd\_list)

**Output:**

[55, 75, 13, 63]

**Using Lambda Function with map()**

A method and a list are passed to Python's map() function.

The function is executed for all of the elements within the list, and a new list is produced with elements generated by the given function for every item.

The map() method is used to square all the entries in a list in this example.

**Code**

1. #Code to calculate the square of each number of a list using the map() function
3. numbers\_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]
5. squared\_list = list(map( lambda num: num \*\* 2 , numbers\_list ))
7. print( squared\_list )

**Output:**

[4, 16, 25, 1, 9, 49, 64, 81, 100]

**Using Lambda Function with List Comprehension**

We'll apply the lambda function combined with list comprehension and lambda keyword with a for loop in this instance. We'll attempt to print the square of numbers in the range 0 to 11.

**Code**

1. #Code to calculate square of each number of list using list comprehension
2. squares = [lambda num = num: num \*\* 2 for num in range(0, 11)]
4. for square in squares:
5. print( square(), end = " ")

**Output:**

0 1 4 9 16 25 36 49 64 81 100

**Using Lambda Function with if-else**

We will use the lambda function with the if-else block.

**Code**

1. # Code to use lambda function with if-else
2. Minimum = lambda x, y : x if (x < y) else y
4. print(Minimum( 35, 74 ))

**Output:**

35

**Using Lambda with Multiple Statements**

Multiple expressions are not allowed in lambda functions, but we can construct 2 lambda functions or more and afterward call the second lambda expression as an argument to the first. Let's use lambda to discover the third maximum element.

**Code**

1. # Code to print the third-largest number of the given list using the lambda function
3. my\_List = [ [3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5] ]
5. # sorting every sublist of the above list
6. sort\_List = lambda num : ( sorted(n) for n in num )
8. # Getting the third largest number of the sublist
9. third\_Largest = lambda num, func : [ l[ len(l) - 2] for l in func(num)]
10. result = third\_Largest( my\_List, sort\_List)
12. print( result )

**Output:**

[6, 54, 5]

**Python File Handling**

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

* Open a file
* Read or write - Performing operation
* Close the file

**Opening a file**

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

**Syntax:**

1. file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

|  |  |  |
| --- | --- | --- |
| **SN** | **Access mode** | **Description** |
| 1 | r | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2 | rb | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. |
| 3 | r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| 4 | rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 5 | w | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 6 | wb | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file. |
| 7 | w+ | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. |
| 8 | wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 9 | a | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. |
| 10 | ab | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. |
| 11 | a+ | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. |
| 12 | ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

**Example**

1. #opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
4. if fileptr:
5. print("file is opened successfully")

**Output:**

<class '\_io.TextIOWrapper'>

file is opened successfully

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

**The close() method**

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

**Syntax**

1. fileobject.close()

Consider the following example.

1. # opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
4. if fileptr:
5. print("file is opened successfully")
7. #closes the opened file
8. fileptr.close()

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

1. try:
2. fileptr = open("file.txt")
3. # perform file operations
4. finally:
5. fileptr.close()

**The with statement**

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

1. with open(<file name>, <access mode>) as <file-pointer>:
2. #statement suite

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

**Example**

1. with open("file.txt",'r') as f:
2. content = f.read();
3. print(content)

**Writing the file**

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

**a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

**Example**

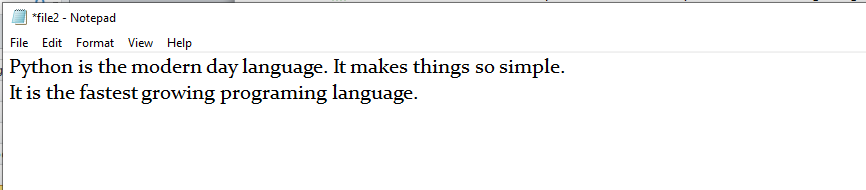
1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")
4. # appending the content to the file
5. fileptr.write('''''Python is the modern day language. It makes things so simple.
6. It is the fastest-growing programing language''')
8. # closing the opened the file
9. fileptr.close()

**Output:**

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

**Snapshot of the file2.txt**



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

**Example 2**

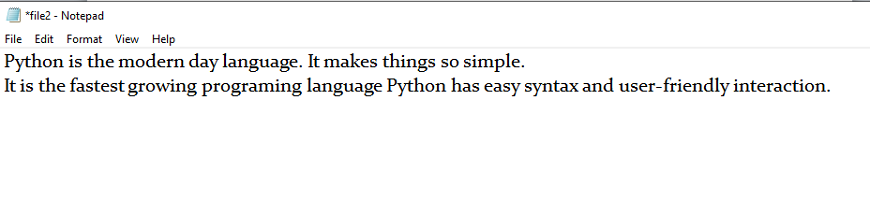
1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
4. #overwriting the content of the file
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
7. #closing the opened file
8. fileptr.close()

**Output:**

Python is the modern day language. It makes things so simple.

It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.

**Snapshot of the file2.txt**



We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

**Syntax:**

1. fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

**Example**

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r")
3. #stores all the data of the file into the variable content
4. content = fileptr.read(10)
5. # prints the type of the data stored in the file
6. print(type(content))
7. #prints the content of the file
8. print(content)
9. #closes the opened file
10. fileptr.close()

**Output:**

<class 'str'>

Python is

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

1. content = fileptr.read()
2. print(content)

**Output:**

Python is the modern-day language. It makes things so simple.

It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.

**Read file through for loop**

We can read the file using for loop. Consider the following example.

1. #open the file.txt in read mode. causes an error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #running a for loop
4. for i in fileptr:
5. print(i) # i contains each line of the file

**Output:**

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

**Read Lines of the file**

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

**Example 1: Reading lines using readline() function**

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #stores all the data of the file into the variable content
4. content = fileptr.readline()
5. content1 = fileptr.readline()
6. #prints the content of the file
7. print(content)
8. print(content1)
9. #closes the opened file
10. fileptr.close()

**Output:**

Python is the modern day language.

It makes things so simple.

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

**Example 2: Reading Lines Using readlines() function**

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
4. #stores all the data of the file into the variable content
5. content = fileptr.readlines()
7. #prints the content of the file
8. print(content)
10. #closes the opened file
11. fileptr.close()

**Output:**

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

**Creating a new file**

The new file can be created by using one of the following access modes with the function open().

**x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

**Example 1**

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","x")
3. print(fileptr)
4. if fileptr:
5. print("File created successfully")

**Output:**

<\_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

**File Pointer positions**

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
7. #reading the content of the file
8. content = fileptr.read();
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.
12. print("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at: 117

**Modifying file pointer position**

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

The syntax to use the seek() method is given below.

**Syntax:**

1. <file-ptr>.seek(offset[, from)

The seek() method accepts two parameters:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

**Example**

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
10. #tell() returns the location of the fileptr.
11. print("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at: 10

**Python OS module**

**Renaming the file**

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

**Syntax:**

1. rename(current-name, new-name)

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

**Example 1:**

1. import os
3. #rename file2.txt to file3.txt
4. os.rename("file2.txt","file3.txt")

**Output:**

The above code renamed current **file2.txt** to **file3.txt**

**Removing the file**

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

1. remove(file-name)

**Example 1**

1. import os;
2. #deleting the file named file3.txt
3. os.remove("file3.txt")

**Creating the new directory**

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

**Syntax:**

1. mkdir(directory name)

**Example 1**

1. import os
3. #creating a new directory with the name new
4. os.mkdir("new")

**The getcwd() method**

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

**Syntax**

1. os.getcwd()

**Example**

1. import os
2. os.getcwd()

**Output:**

'C:\\Users\\DEVANSH SHARMA'

**Changing the current working directory**

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

**Syntax**

1. chdir("new-directory")

**Example**

1. import os
2. # Changing current directory with the new directiory
3. os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
4. #It will display the current working directory
5. os.getcwd()

**Output:**

'C:\\Users\\DEVANSH SHARMA\\Documents'

**Deleting directory**

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

**Syntax**

1. os.rmdir(directory name)

**Example 1**

1. import os
2. #removing the new directory
3. os.rmdir("directory\_name")

It will remove the specified directory.

**Writing Python output to the files**

In Python, there are the requirements to write the output of a Python script to a file.

The **check\_call()** method of module **subprocess** is used to execute a Python script and write the output of that script to a file.

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt.**

**Example**

**file.py**

1. temperatures=[10,-20,-289,100]
2. def c\_to\_f(c):
3. if c< -273.15:
4. return "That temperature doesn't make sense!"
5. else:
6. f=c\*9/5+32
7. return f
8. for t in temperatures:
9. print(c\_to\_f(t))

**file.py**

1. import subprocess
3. with open("output.txt", "wb") as f:
4. subprocess.check\_call(["python", "file.py"], stdout=f)

**The file related methods**

The file object provides the following methods to manipulate the files on various operating systems.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | file.close() | It closes the opened file. The file once closed, it can't be read or write anymore. |
| 2 | File.fush() | It flushes the internal buffer. |
| 3 | File.fileno() | It returns the file descriptor used by the underlying implementation to request I/O from the OS. |
| 4 | File.isatty() | It returns true if the file is connected to a TTY device, otherwise returns false. |
| 5 | File.next() | It returns the next line from the file. |
| 6 | File.read([size]) | It reads the file for the specified size. |
| 7 | File.readline([size]) | It reads one line from the file and places the file pointer to the beginning of the new line. |
| 8 | File.readlines([sizehint]) | It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function. |
| 9 | File.seek(offset[,from) | It modifies the position of the file pointer to a specified offset with the specified reference. |
| 10 | File.tell() | It returns the current position of the file pointer within the file. |
| 11 | File.truncate([size]) | It truncates the file to the optional specified size. |
| 12 | File.write(str) | It writes the specified string to a file |
| 13 | File.writelines(seq) | It writes a sequence of the strings to a file. |

**Python Modules**

This tutorial will explain how to construct and import custom Python modules. Additionally, we may import or integrate Python's built-in modules via various methods.

**What is Modular Programming?**

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Modularizing our code in a big application has a lot of benefits.

**Simplification:** A module often concentrates on one comparatively small area of the overall problem instead of the full task. We will have a more manageable design problem to think about if we are only concentrating on one module. Program development is now simpler and much less vulnerable to mistakes.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Flexibility:** Modules are frequently used to establish conceptual separations between various problem areas. It is less likely that changes to one module would influence other portions of the program if modules are constructed in a fashion that reduces interconnectedness. (We might even be capable of editing a module despite being familiar with the program beyond it.) It increases the likelihood that a group of numerous developers will be able to collaborate on a big project.

**Reusability:** Functions created in a particular module may be readily accessed by different sections of the assignment (through a suitably established api). As a result, duplicate code is no longer necessary.

**Scope:** Modules often declare a distinct namespace to prevent identifier clashes in various parts of a program.

In Python, modularization of the code is encouraged through the use of functions, modules, and packages.

**What are Modules in Python?**

A document with definitions of functions and various statements written in Python is called a Python module.

In Python, we can define a module in one of 3 ways:

* Python itself allows for the creation of modules.
* Similar to the re (regular expression) module, a module can be primarily written in C programming language and then dynamically inserted at run-time.
* A built-in module, such as the itertools module, is inherently included in the interpreter.

A module is a file containing Python code, definitions of functions, statements, or classes. An example\_module.py file is a module we will create and whose name is example\_module.

We employ modules to divide complicated programs into smaller, more understandable pieces. Modules also allow for the reuse of code.

Rather than duplicating their definitions into several applications, we may define our most frequently used functions in a separate module and then import the complete module.

Let's construct a module. Save the file as example\_module.py after entering the following.

**Code**

1. # Python program to show how to create a module.
2. # defining a function in the module to reuse it
3. def square( number ):
4. """This function will square the number passed to it"""
6. result = number \*\* 2
7. return result

Here, a module called example\_module contains the definition of the function square(). The function returns the square of a given number.

**How to Import Modules in Python?**

In Python, we may import functions from one module into our program, or as we say into, another module.

For this, we make use of the import Python keyword. In the Python window, we add the next to import keyword, the name of the module we need to import. We will import the module we defined earlier example\_module.

**Code**

1. import example\_module

The functions that we defined in the example\_module are not immediately imported into the present program. Only the name of the module, i.e., example\_ module, is imported here.

We may use the dot operator to use the functions using the module name. For instance:

**Code**

1. result = example\_module.square(  4  )
2. print( "By using the module square of number is: ", result )

**Output:**

By using the module square of number is: 16

There are several standard modules for Python. The complete list of Python standard modules is available. The list can be seen using the help command.

Similar to how we imported our module, a user-defined module, we can use an import statement to import other standard modules.

Importing a module can be done in a variety of ways. Below is a list of them.

**Python import Statement**

Using the import Python keyword and the dot operator, we may import a standard module and can access the defined functions within it. Here's an illustration.

**Code**

1. # Python program to show how to import a standard module
2. # We will import the math module which is a standard module
4. import math
5. print( "The value of euler's number is", math.e )

**Output:**

The value of euler's number is 2.718281828459045

**Importing and also Renaming**

While importing a module, we can change its name too. Here is an example to show.

**Code**

1. # Python program to show how to import a module and rename it
2. # We will import the math module and give a different name to it
4. import math as mt
5. print( "The value of euler's number is", mt.e )

**Output:**

The value of euler's number is 2.718281828459045

The math module is now named mt in this program. In some circumstances, it might help us type faster in case of modules having long names.

Please take note that now the scope of our program does not include the term math. Thus, mt.pi is the proper implementation of the module, whereas math.pi is invalid.

**Python from...import Statement**

We can import specific names from a module without importing the module as a whole. Here is an example.

**Code**

1. # Python program to show how to import specific objects from a module
2. # We will import euler's number from the math module using the from keyword
4. from math import e
5. print( "The value of euler's number is", e )

**Output:**

The value of euler's number is 2.718281828459045

Only the e constant from the math module was imported in this case.

We avoid using the dot (.) operator in these scenarios. As follows, we may import many attributes at the same time:

**Code**

1. # Python program to show how to import multiple objects from a module
2. from math import e, tau
3. print( "The value of tau constant is: ", tau )
4. print( "The value of the euler's number is: ", e )

**Output:**

The value of tau constant is: 6.283185307179586

The value of the euler's number is: 2.718281828459045

**Import all Names - From import \* Statement**

To import all the objects from a module within the present namespace, use the \* symbol and the from and import keyword.

**Syntax:**

1. from name\_of\_module import \*

There are benefits and drawbacks to using the symbol \*. It is not advised to use \* unless we are certain of our particular requirements from the module; otherwise, do so.

Here is an example of the same.

**Code**

1. # importing the complete math module using \*
2. from math import \*
4. # accessing functions of math module without using the dot operator
5. print( "Calculating square root: ", sqrt(25) )
6. print( "Calculating tangent of an angle: ", tan(pi/6) ) # here pi is also imported from the math module

**Output:**

Calculating square root: 5.0

Calculating tangent of an angle: 0.5773502691896257

**Locating Path of Modules**

The interpreter searches numerous places when importing a module in the Python program. Several directories are searched if the built-in module is not present. The list of directories can be accessed using sys.path. The Python interpreter looks for the module in the way described below:

The module is initially looked for in the current working directory. Python then explores every directory in the shell parameter PYTHONPATH if the module cannot be located in the current directory. A list of folders makes up the environment variable known as PYTHONPATH. Python examines the installation-dependent set of folders set up when Python is downloaded if that also fails.

Here is an example to print the path.

**Code**

1. # We will import the sys module
2. import sys
4. # we will import sys.path
5. print(sys.path)

**Output:**

['/home/pyodide', '/home/pyodide/lib/Python310.zip', '/lib/Python3.10', '/lib/Python3.10/lib-dynload', '', '/lib/Python3.10/site-packages']

**The dir() Built-in Function**

We may use the dir() method to identify names declared within a module.

For instance, we have the following names in the standard module str. To print the names, we will use the dir() method in the following way:

**Code**

1. # Python program to print the directory of a module
2. print( "List of functions:\n ", dir( str ), end=", " )

**Output:**

List of functions:

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

**Namespaces and Scoping**

Objects are represented by names or identifiers called variables. A namespace is a dictionary containing the names of variables (keys) and the objects that go with them (values).

Both local and global namespace variables can be accessed by a Python statement. When two variables with the same name are local and global, the local variable takes the role of the global variable. There is a separate local namespace for every function. The scoping rule for class methods is the same as for regular functions. Python determines if parameters are local or global based on reasonable predictions. Any variable that is allocated a value in a method is regarded as being local.

Therefore, we must use the global statement before we may provide a value to a global variable inside of a function. Python is informed that Var\_Name is a global variable by the line global Var\_Name. Python stops looking for the variable inside the local namespace.

We declare the variable Number, for instance, within the global namespace. Since we provide a Number a value inside the function, Python considers a Number to be a local variable. UnboundLocalError will be the outcome if we try to access the value of the local variable without or before declaring it global.

**Code**

1. Number = 204
2. def AddNumber():
3. # accessing the global namespace
4. global Number
5. Number = Number + 200
7. print( Number )
8. AddNumber()
9. print( Number )

**Output:**

204

404

**Python Exceptions**

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception. We will see what an exception is. Also, we will see the difference between a syntax error and an exception in this tutorial. Following that, we will learn about trying and except blocks and how to raise exceptions and make assertions. After that, we will see the Python exceptions list.

**What is an Exception?**

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

**Exceptions versus Syntax Errors**

When the interpreter identifies a statement that has an error, syntax errors occur. Consider the following scenario:

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Code**

1. #Python code after removing the syntax error
2. string = "Python Exceptions"
4. for s in string:
5. if (s != o:
6. print( s )

**Output:**

if (s != o:

^

SyntaxError: invalid syntax

The arrow in the output shows where the interpreter encountered a syntactic error. There was one unclosed bracket in this case. Close it and rerun the program:

**Code**

1. #Python code after removing the syntax error
2. string = "Python Exceptions"
4. for s in string:
5. if (s != o):
6. print( s )

**Output:**

2 string = "Python Exceptions"

4 for s in string:

----> 5 if (s != o):

6 print( s )

NameError: name 'o' is not defined

We encountered an exception error after executing this code. When syntactically valid Python code produces an error, this is the kind of error that arises. The output's last line specified the name of the exception error code encountered. Instead of displaying just "exception error", Python displays information about the sort of exception error that occurred. It was a NameError in this situation. Python includes several built-in exceptions. However, Python offers the facility to construct custom exceptions.

**Try and Except Statement - Catching Exceptions**

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

**Code**

1. # Python code to catch an exception and handle it using try and except code blocks
3. a = ["Python", "Exceptions", "try and except"]
4. try:
5. #looping through the elements of the array a, choosing a range that goes beyond the length of the array
6. for i in range( 4 ):
7. print( "The index and element from the array is", i, a[i] )
8. #if an error occurs in the try block, then except block will be executed by the Python interpreter
9. except:
10. print ("Index out of range")

**Output:**

The index and element from the array is 0 Python

The index and element from the array is 1 Exceptions

The index and element from the array is 2 try and except

Index out of range

The code blocks that potentially produce an error are inserted inside the try clause in the preceding example. The value of i greater than 2 attempts to access the list's item beyond its length, which is not present, resulting in an exception. The except clause then catches this exception and executes code without stopping it.

**How to Raise an Exception**

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

**Code**

1. #Python code to show how to raise an exception in Python
2. num = [3, 4, 5, 7]
3. if len(num) > 3:
4. raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )

**Output:**

1 num = [3, 4, 5, 7]

2 if len(num) > 3:

----> 3 raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )

Exception: Length of the given list must be less than or equal to 3 but is 4

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

**Assertions in Python**

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

**The assert Statement**

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

**The syntax for the assert clause is −**

1. assert Expressions[, Argument]

Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

**Code**

1. #Python program to show how to use assert keyword
2. # defining a function
3. def square\_root( Number ):
4. assert ( Number < 0), "Give a positive integer"
5. return Number\*\*(1/2)
7. #Calling function and passing the values
8. print( square\_root( 36 ) )
9. print( square\_root( -36 ) )

**Output:**

7 #Calling function and passing the values

----> 8 print( square\_root( 36 ) )

9 print( square\_root( -36 ) )

Input In [23], in square\_root(Number)

3 def square\_root( Number ):

----> 4 assert ( Number < 0), "Give a positive integer"

5 return Number\*\*(1/2)

AssertionError: Give a positive integer

**Try with Else Clause**

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

Here is an instance of a try clause with an else clause.

**Code**

1. # Python program to show how to use else clause with try and except clauses
3. # Defining a function which returns reciprocal of a number
4. def reciprocal( num1 ):
5. try:
6. reci = 1 / num1
7. except ZeroDivisionError:
8. print( "We cannot divide by zero" )
9. else:
10. print ( reci )
11. # Calling the function and passing values
12. reciprocal( 4 )
13. reciprocal( 0 )

**Output:**

0.25

We cannot divide by zero

**Finally Keyword in Python**

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Here is an example of finally keyword with try-except clauses:

**Code**

1. # Python code to show the use of finally clause
3. # Raising an exception in try block
4. try:
5. div = 4 // 0
6. print( div )
7. # this block will handle the exception raised
8. except ZeroDivisionError:
9. print( "Atepting to divide by zero" )
10. # this will always be executed no matter exception is raised or not
11. finally:
12. print( 'This is code of finally clause' )

**Output:**

Atepting to divide by zero

This is code of finally clause

**User-Defined Exceptions**

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.

We raise a user-defined exception in the try block and then handle the exception in the except block. An example of the class EmptyError is created using the variable var.

**Code**

1. class EmptyError( RuntimeError ):
2. def \_\_init\_\_(self, argument):
3. self.arguments = argument
4. Once the preceding class has been created, the following is how to raise an exception:
5. Code
6. var = " "
7. try:
8. raise EmptyError( "The variable is empty" )
9. except (EmptyError, var):
10. print( var.arguments )

**Output:**

2 try:

----> 3 raise EmptyError( "The variable is empty" )

4 except (EmptyError, var):

EmptyError: The variable is empty

**Python Exceptions List**

Here is the complete list of Python in-built exceptions.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the Exception** | **Description of the Exception** |
| **1** | **Exception** | All exceptions of Python have a base class. |
| **2** | **StopIteration** | If the next() method returns null for an iterator, this exception is raised. |
| **3** | **SystemExit** | The sys.exit() procedure raises this value. |
| **4** | **StandardError** | Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions. |
| **5** | **ArithmeticError** | All mathematical computation errors belong to this base class. |
| **6** | **OverflowError** | This exception is raised when a computation surpasses the numeric data type's maximum limit. |
| **7** | **FloatingPointError** | If a floating-point operation fails, this exception is raised. |
| **8** | **ZeroDivisionError** | For all numeric data types, its value is raised whenever a number is attempted to be divided by zero. |
| **9** | **AssertionError** | If the Assert statement fails, this exception is raised. |
| **10** | **AttributeError** | This exception is raised if a variable reference or assigning a value fails. |
| **11** | **EOFError** | When the endpoint of the file is approached, and the interpreter didn't get any input value by raw\_input() or input() functions, this exception is raised. |
| **12** | **ImportError** | This exception is raised if using the import keyword to import a module fails. |
| **13** | **KeyboardInterrupt** | If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised. |
| **14** | **LookupError** | LookupErrorBase is the base class for all search errors. |
| **15** | **IndexError** | This exception is raised when the index attempted to be accessed is not found. |
| **16** | **KeyError** | When the given key is not found in the dictionary to be found in, this exception is raised. |
| **17** | **NameError** | This exception is raised when a variable isn't located in either local or global namespace. |
| **18** | **UnboundLocalError** | This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value. |
| **19** | **EnvironmentError** | All exceptions that arise beyond the Python environment have this base class. |
| **20** | **IOError** | If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised. |
| **22** | **SyntaxError** | This exception is raised whenever a syntax error occurs in our program. |
| **23** | **IndentationError** | This exception was raised when we made an improper indentation. |
| **24** | **SystemExit** | This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code. |
| **25** | **TypeError** | This exception is raised whenever a data type-incompatible action or function is tried to be executed. |
| **26** | **ValueError** | This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values. |
| **27** | **RuntimeError** | This exception is raised when an error that occurred during the program's execution cannot be classified. |
| **28** | **NotImplementedError** | If an abstract function that the user must define in an inherited class is not defined, this exception is raised. |

**Summary**

We learned about different methods to raise, catch, and handle Python exceptions after learning the distinction between syntax errors and exceptions. We learned about these clauses in this tutorial:

* We can throw an exception at any line of code using the raise keyword.
* Using the assert keyword, we may check to see if a specific condition is fulfilled and raise an exception if it is not.
* All statements are carried out in the try clause until an exception is found.
* The try clause's exception(s) are detected and handled using the except function.
* If no exceptions are thrown in the try code block, we can write code to be executed in the else code block.

Here is the syntax of try, except, else, and finally clauses.

**Syntax:**

1. try:
2. # Code block
3. # These statements are those which can probably have some error
5. except:
6. # This block is optional.
7. # If the try block encounters an exception, this block will handle it.
9. else:
10. # If there is no exception, this code block will be executed by the Python interpreter
12. finally:
13. # Python interpreter will always execute this code.

**Python Date and time**

Python provides the **datetime** module work with real dates and times. In real-world applications, we need to work with the date and time. Python enables us to schedule our Python script to run at a particular timing.

In Python, the date is not a data type, but we can work with the date objects by importing the module named with **datetime, time, and calendar**.

In this section of the tutorial, we will discuss how to work with the date and time objects in Python.

The **datetime** classes are classified in the six main classes.

* **date** - It is a naive ideal date. It consists of the year, month, and day as attributes.
* **time** - It is a perfect time, assuming every day has precisely 24\*60\*60 seconds. It has hour, minute, second, microsecond, and **tzinfo** as attributes.
* **datetime** - It is a grouping of date and time, along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
* **timedelta -** It represents the difference between two dates, time or datetime instances to microsecond resolution.
* **tzinfo** - It provides time zone information objects.
* **timezone -** It is included in the new version of Python. It is the class that implements the **tzinfo** abstract base class.

**Tick**

In Python, the time instants are counted since 12 AM, 1st January 1970. The function **time()** of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Consider the following example

1. import time;
2. #prints the number of ticks spent since 12 AM, 1st January 1970
3. print(time.time())

**Output:**

1585928913.6519969

**How to get the current time?**

The localtime() functions of the time module are used to get the current time tuple. Consider the following example.

**Example**

1. import time;
3. #returns a time tuple
5. print(time.localtime(time.time()))

**Output:**

time.struct\_time(tm\_year=2020, tm\_mon=4, tm\_mday=3, tm\_hour=21, tm\_min=21, tm\_sec=40, tm\_wday=4, tm\_yday=94, tm\_isdst=0)

**Time tuple**

The time is treated as the tuple of 9 numbers. Let's look at the members of the time tuple.

|  |  |  |
| --- | --- | --- |
| **Index** | **Attribute** | **Values** |
| 0 | Year | 4 digit (for example 2018) |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 60 |
| 6 | Day of weak | 0 to 6 |
| 7 | Day of year | 1 to 366 |
| 8 | Daylight savings | -1, 0, 1 , or -1 |

**Getting formatted time**

The time can be formatted by using the **asctime()** function of the time module. It returns the formatted time for the time tuple being passed.

**Example**

1. import time
2. #returns the formatted time
4. print(time.asctime(time.localtime(time.time())))

**Output:**

Tue Dec 18 15:31:39 2018

**Python sleep time**

The **sleep()** method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds provided as the float.

Consider the following example.

**Example**

1. import time
2. for i in range(0,5):
3. print(i)
4. #Each element will be printed after 1 second
5. time.sleep(1)

**Output:**

0

1

2

3

4

**The datetime Module**

The **datetime** module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import **the datetime** module into the python source code.

Consider the following example to get the **datetime** object representation for the current time.

**Example**

1. import datetime
2. #returns the current datetime object
3. print(datetime.datetime.now())

**Output:**

2020-04-04 13:18:35.252578

**Creating date objects**

We can create the date objects bypassing the desired date in the datetime constructor for which the date objects are to be created.

Consider the following example.

**Example**

1. import datetime
2. #returns the datetime object for the specified date
3. print(datetime.datetime(2020,04,04))

**Output:**

2020-04-04 00:00:00

We can also specify the time along with the date to create the datetime object. Consider the following example.

**Example**

1. import datetime
3. #returns the datetime object for the specified time
5. print(datetime.datetime(2020,4,4,1,26,40))

**Output:**

2020-04-04 01:26:40

In the above code, we have passed in **datetime()** function year, month, day, hour, minute, and millisecond attributes in a sequential manner.

**Comparison of two dates**

We can compare two dates by using the comparison operators like >, >=, <, and <=.

Consider the following example.

**Example**

1. from datetime import datetime as dt
2. #Compares the time. If the time is in between 8AM and 4PM, then it prints working hours otherwise it prints fun hours
3. if dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):
4. print("Working hours....")
5. else:
6. print("fun hours")

**Output:**

fun hours

**The calendar module**

Python provides a calendar object that contains various methods to work with the calendars.

Consider the following example to print the calendar for the last month of 2018.

**Example**

1. import calendar;
2. cal = calendar.month(2020,3)
3. #printing the calendar of December 2018
4. print(cal)

**Output:**

March 2020

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

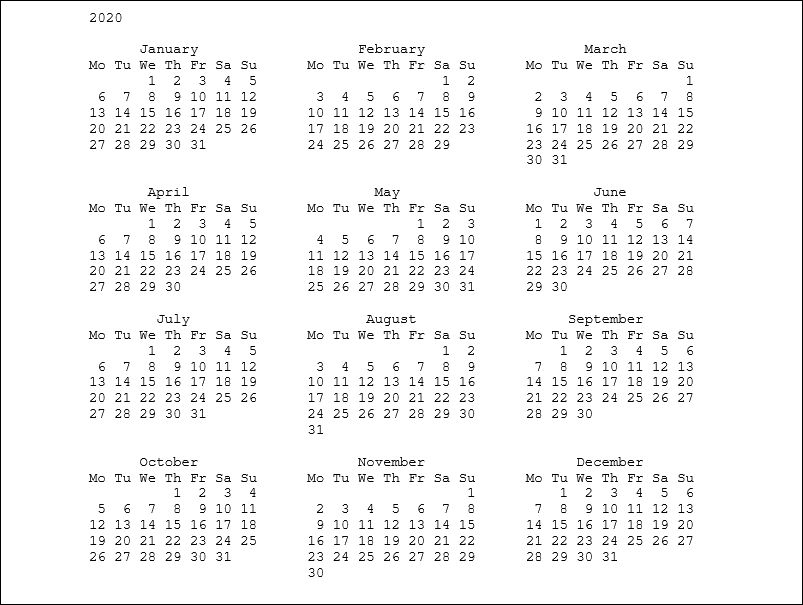
**Printing the calendar of whole year**

The prcal() method of calendar module is used to print the calendar of the entire year. The year of which the calendar is to be printed must be passed into this method.

**Example**

1. import calendar
2. #printing the calendar of the year 2019
3. s = calendar.prcal(2020)

**Output:**



**Python Regex**

A regular expression is a set of characters with highly specialized syntax that we can use to find or match other characters or groups of characters. In short, regular expressions, or Regex, are widely used in the UNIX world.

The re-module in Python gives full support for regular expressions of Pearl style. The re module raises the re.error exception whenever an error occurs while implementing or using a regular expression.

We'll go over two crucial functions utilized to deal with regular expressions. But first, a minor point: many letters have a particular meaning when utilized in a regular expression.

**re.match()**

Python's re.match() function finds and delivers the very first appearance of a regular expression pattern. In Python, the RegEx Match function solely searches for a matching string at the beginning of the provided text to be searched. The matching object is produced if one match is found in the first line. If a match is found in a subsequent line, the Python RegEx Match function gives output as null.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Examine the implementation for the re.match() method in Python. The expressions ".w\*" and ".w\*?" will match words that have the letter "w," and anything that does not has the letter "w" will be ignored. The for loop is used in this Python re.match() illustration to inspect for matches for every element in the list of words.

**Matching Characters**

The majority of symbols and characters will easily match. (A case-insensitive feature can be enabled, allowing this RE to match Python or PYTHON.) The regular expression check, for instance, will match exactly the string check.

There are some exceptions to this general rule; certain symbols are special metacharacters that don't match. Rather, they indicate that they must compare something unusual, or they have an effect on other parts of the RE by recurring or modifying their meaning.

Here's the list of the metacharacters;

1. . ^ $ \* + ? { } [ ] \ | ( )

**Repeating Things**

The ability to match different sets of symbols will be the first feature regular expressions can achieve that's not previously achievable with string techniques. On the other hand, Regexes isn't much of an improvement if that had been their only extra capacity. We can also define that some sections of the RE must be reiterated a specified number of times.

The first metacharacter we'll examine for recurring occurrences is \*. Instead of matching the actual character '\*,' \* signals that the preceding letter can be matched 0 or even more times, rather than exactly one.

Ba\*t, for example, matches 'bt' (zero 'a' characters), 'bat' (one 'a' character), 'baaat' (three 'a' characters), etc.

Greedy repetitions, such as \*, cause the matching algorithm to attempt to replicate the RE as many times as feasible. If later elements of the sequence fail to match, the matching algorithm will retry with lesser repetitions.

This is the syntax of re.match() function -

1. re.match(pattern, string, flags=0)

**Parameters**

**pattern:-** this is the expression that is to be matched. It must be a regular expression

**string:-** This is the string that will be compared to the pattern at the start of the string.

**flags:-** Bitwise OR (|) can be used to express multiple flags. These are modifications, and the table below lists them.

**Code**

1. import re
2. line = "Learn Python through tutorials on javatpoint"
3. match\_object = re.match( r'.w\* (.w?) (.w\*?)', line, re.M|re.I)
5. if match\_object:
6. print ("match object group : ", match\_object.group())
7. print ("match object 1 group : ", match\_object.group(1))
8. print ("match object 2 group : ", match\_object.group(2))
9. else:
10. print ( "There isn't any match!!" )

**Output:**

There isn't any match!!

**re.search()**

The re.search() function will look for the first occurrence of a regular expression sequence and deliver it. It will verify all rows of the supplied string, unlike Python's re.match(). If the pattern is matched, the re.search() function produces a match object; otherwise, it returns "null."

To execute the search() function, we must first import the Python re-module and afterward run the program. The "sequence" and "content" to check from our primary string are passed to the Python re.search() call.

This is the syntax of re.search() function -

1. re.search(pattern, string, flags=0)

Here is the description of the parameters -

**pattern:-** this is the expression that is to be matched. It must be a regular expression

**string:-** The string provided is the one that will be searched for the pattern wherever within it.

**flags:-** Bitwise OR (|) can be used to express multiple flags. These are modifications, and the table below lists them.

**Code**

1. import re
3. line = "Learn Python through tutorials on javatpoint";
5. search\_object = re.search( r' .\*t? (.\*t?) (.\*t?)', line)
6. if search\_object:
7. print("search object group : ", search\_object.group())
8. print("search object group 1 : ", search\_object.group(1))
9. print("search object group 2 : ", search\_object.group(2))
10. else:
11. print("Nothing found!!")

**Output:**

search object group : Python through tutorials on javatpoint

search object group 1 : on

search object group 2 : javatpoint

**Matching Versus Searching**

Python has two primary regular expression functions: match and search. Match looks for a match only where the string commencements, whereas search looks for a match everywhere in the string (this is the default function of Perl).

**Code**

1. import re
3. line = "Learn Python through tutorials on javatpoint"
5. match\_object = re.match( r'through', line, re.M|re.I)
6. if match\_object:
7. print("match object group : ", match\_object.group())
8. else:
9. print( "There isn't any match!!")
11. search\_object = re.search( r' .\*t? ', line, re.M|re.I)
12. if searchObj:
13. print("search object group : ", search\_object.group())
14. else:
15. print("Nothing found!!")

**Output:**

There isn't any match!!

search object group : Python through tutorials on

**re.findall()**

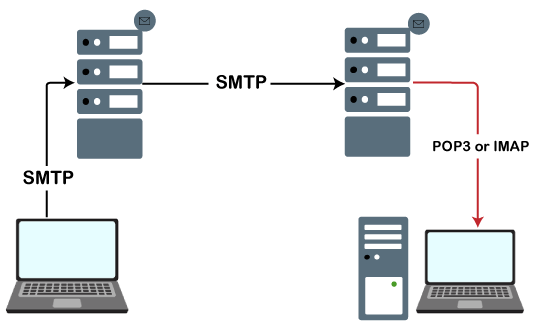
The findall() function is often used to look for "all" appearances of a pattern. The search() module, on the other hand, will only provide the earliest occurrence that matches the description. In a single operation, findall() will loop over all the rows of the document and provide all non-overlapping regular matches.

We have a line of text, and we want to get all of the occurrences from the content, so we use Python's re.findall() function. It will search the entire content provided to it.

Using the re-package isn't always a good idea. If we're only searching a fixed string or a specific character class, and we're not leveraging any re features like the IGNORECASE flag, regular expressions' full capability would not be needed. Strings offer various ways for doing tasks with fixed strings, and they're generally considerably faster than the larger, more generalized regular expression solver because the execution is a simple short C loop that has been optimized for the job.

**Python Sending Email using SMTP**

Simple Mail Transfer Protocol (SMTP) is used as a protocol to handle the email transfer using Python. It is used to route emails between email servers. It is an application layer protocol which allows to users to send mail to another. The receiver retrieves email using the protocols **POP(Post Office Protocol)** and **IMAP(Internet Message Access Protocol)**.



When the server listens for the TCP connection from a client, it initiates a connection on port 587.

Python provides a **smtplib** module, which defines an the SMTP client session object used to send emails to an internet machine. For this purpose, we have to import the **smtplib** module using the import statement.

1. $ import smtplib

The SMTP object is used for the email transfer. The following syntax is used to create the smtplib object.

x[Icon

Description automatically generated](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. import smtplib
2. smtpObj = smtplib.SMTP(host, port, local\_hostname)

It accepts the following parameters.

* **host:** It is the hostname of the machine which is running your SMTP server. Here, we can specify the IP address of the server like ([https://www.javatpoint.com](https://www.javatpoint.com/)) or localhost. It is an optional parameter.
* **port:** It is the port number on which the host machine is listening to the SMTP connections. It is 25 by default.
* **local\_hostname:** If the SMTP server is running on your local machine, we can mention the hostname of the local machine.

The sendmail() method of the SMTP object is used to send the mail to the desired machine. The syntax is given below.

1. smtpObj.sendmail(sender, receiver, message)

**Example**

1. #!/usr/bin/python3
2. import smtplib
3. sender\_mail = 'sender@fromdomain.com'
4. receivers\_mail = ['reciever@todomain.com']
5. message = """From: From Person %s
6. To: To Person %s
7. Subject: Sending SMTP e-mail
8. This is a test e-mail message.
9. """%(sender\_mail,receivers\_mail)
10. try:
11. smtpObj = smtplib.SMTP('localhost')
12. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
13. print("Successfully sent email")
14. except Exception:
15. print("Error: unable to send email")

**Sending email from gmail**

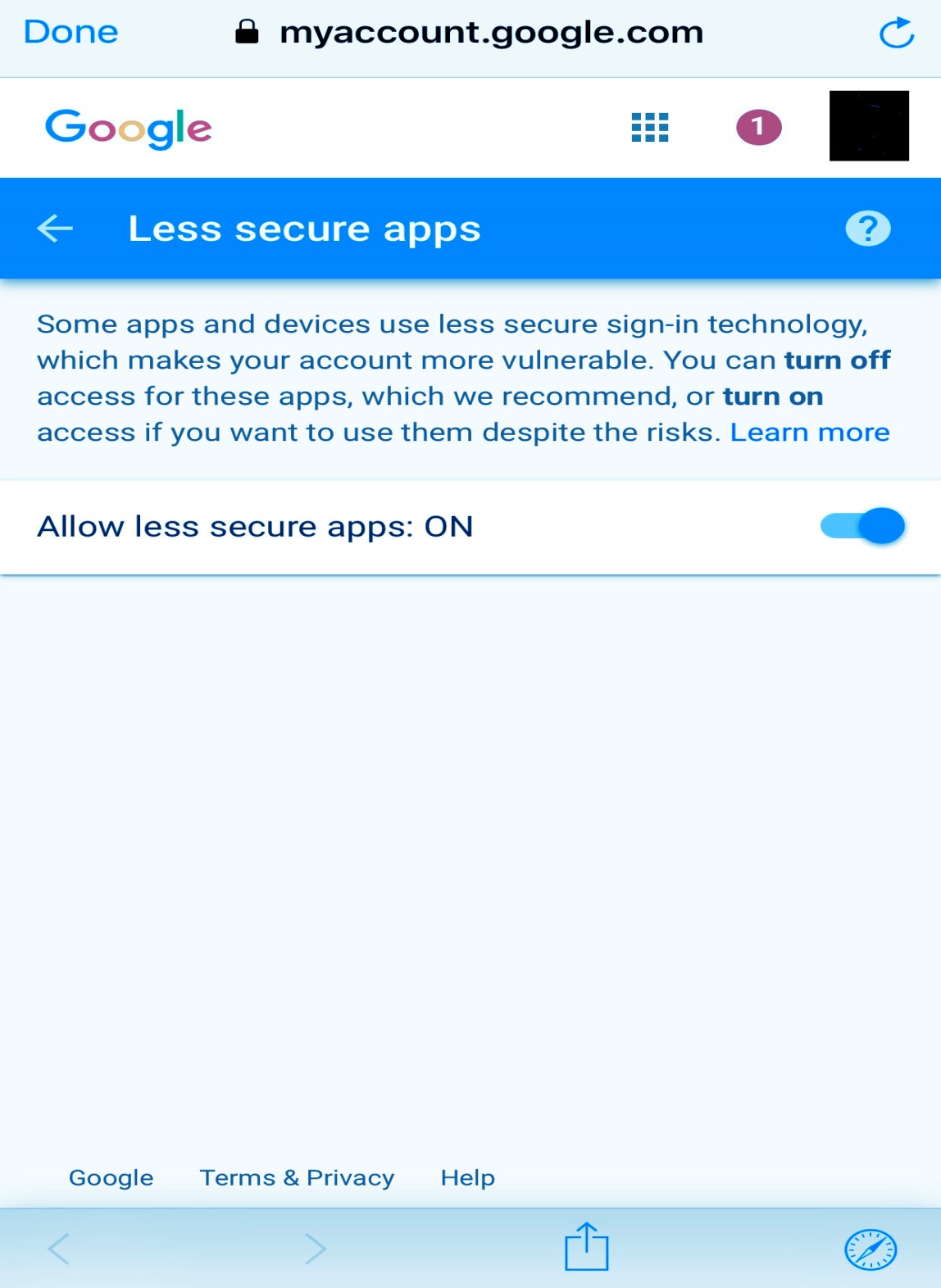
There are cases where the emails are sent using the Gmail SMTP server. In this case, we can pass Gmail as the SMTP server instead of using the localhost with the port 587.

Use the following syntax.

1. $ smtpObj = smtplib.SMTP("gmail.com", 587)

Here, we need to login to the Gmail account using Gmail user name and password. For this purpose, the smtplib provide the login() method, which accepts the username and password of the sender.

This may make your Gmail ask you for access to less secure apps if you're using Gmail. You will need to turn this ON temporarily for this to work.



Consider the following example.

**Example**

1. #!/usr/bin/python3
2. import smtplib
3. sender\_mail = 'sender@gmail.com'
4. receivers\_mail = ['reciever@gmail.com']
5. message = """From: From Person %s
6. To: To Person %s
7. Subject: Sending SMTP e-mail
8. This is a test e-mail message.
9. """%(sender\_mail,receivers\_mail)
10. try:
11. password = input('Enter the password');
12. smtpObj = smtplib.SMTP('gmail.com',587)
13. smtpobj.login(sender\_mail,password)
14. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
15. print("Successfully sent email")
16. except Exception:
17. print("Error: unable to send email")

**Sending HTML in email**

We can format the HTML in the message by specifying the MIME version, content-type, and character set to send the HTML.

Consider the following example.

**Example**

1. #!/usr/bin/python3
2. import smtplib
3. sender\_mail = 'sender@fromdomain.com'
4. receivers\_mail = ['reciever@todomain.com']
5. message = """From: From Person %s
6. To: To Person %s
8. MIME-Version:1.0
9. Content-type:text/html

12. Subject: Sending SMTP e-mail
14. <h3>Python SMTP</h3>
15. <strong>This is a test e-mail message.</strong>
16. """%(sender\_mail,receivers\_mail)
17. try:
18. smtpObj = smtplib.SMTP('localhost')
19. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
20. print("Successfully sent email")
21. except Exception:
22. print("Error: unable to send email")