### 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.")
2. a = 10
3. # Two objects are passed in print() function
4. print("a =", a)
5. b = a
6. # Three objects are passed in print function

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: "))
3. 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")
2. tup[2] = "Papaya"
3. 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**

## Prerequisite

Before learning Python, you must have the basic knowledge of programming concepts.

## Audience

Our Python tutorial is designed to help beginners and professionals.

## Problem

We assure that you will not find any problem in this Python tutorial. But if there is any mistake, please post the problem in contact form.

# 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](https://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.

## 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**](https://www.javatpoint.com/tensorflow)**-** It is an artificial intelligence library which allows us to create large scale AI based projects.
* [**Django**](https://www.javatpoint.com/django-tutorial)**-** It is an open source framework that allows us to develop web applications. It is easy, flexible, and simple to manage.
* [**Flask**](https://www.javatpoint.com/flask-tutorial)**-** It is also an open source web framework. It is used to develop lightweight web applications.
* [**Pandas**](https://www.javatpoint.com/python-pandas)**-** It is a Python library which is used to perform scientific computations.
* [**Keras**](https://www.javatpoint.com/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.

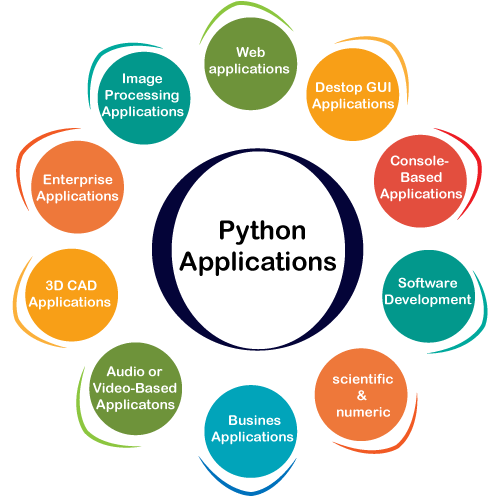
* Desktop Applications
* Web Applications
* [Data Science](https://www.javatpoint.com/data-science)
* [Artificial Intelligence](https://www.javatpoint.com/artificial-intelligence-tutorial)
* [Machine Learning](https://www.javatpoint.com/machine-learning)
* Scientific Computing
* [Robotics](https://www.javatpoint.com/robotics-tutorial)
* [Internet of Things (IoT)](https://www.javatpoint.com/iot-internet-of-things)
* Gaming
* Mobile Apps
* Data Analysis and Preprocessing

In the next topic, we will discuss the [Python Application](https://www.javatpoint.com/python-applications), 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.

# 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

Python Variables

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

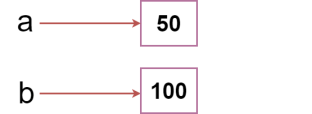
Python Variables

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
4. print(name)
5. print(age)
6. 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"
10. 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)
8. # Calling a function
9. 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
3. # Global variable in function
4. **def** mainFunction():
5. # printing a global variable
6. **global** x
7. **print**(x)
8. # modifying a global variable
9. x = 'Welcome To Javatpoint'
10. **print**(x)
11. mainFunction()
12. **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
3. a = 10000000000000000000000000000000000000000000
4. a = a + 1
5. **print**(type(a))
6. **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))
3. b = 40.5
4. **print**("The type of b", type(b))
5. c = 1+3j
6. **print**("The type of c", type(c))
7. **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))
4. #Printing the list1
5. **print** (list1)
6. # List slicing
7. **print** (list1[3:])
8. # List slicing
9. **print** (list1[0:2])
10. # List Concatenation using + operator
11. **print** (list1 + list1)
12. # List repetation using \* operator
13. **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))
4. #Printing the tuple
5. **print** (tup)
6. # Tuple slicing
7. **print** (tup[1:])
8. **print** (tup[0:1])
9. # Tuple concatenation using + operator
10. **print** (tup + tup)
11. # Tuple repatation using \* operator
12. **print** (tup \* 3)
13. # Adding value to tup. It will throw an error.
14. 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'}
2. # Printing dictionary
3. **print** (d)
4. # Accesing value using keys
5. **print**("1st name is "+d[1])
6. **print**("2nd name is "+ d[4])
7. **print** (d.keys())
8. **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()
3. set2 = {'James', 2, 3,'Python'}
4. #Printing Set value
5. **print**(set2)
6. # Adding element to the set
7. set2.add(10)
8. **print**(set2)
9. #Removing element from the set
10. set2.remove(2)
11. **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
4. # displaying the complete list using "kwlist()."
5. **print**("The set of keywords in this version is: ")
6. **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
5. number = no\_return\_function()
6. **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
4. number = with\_return( 67 )
5. **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)
9. 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)
8. 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):
3. **print**( i + 4, end = " ")
4. # breaking the loop when i = 9
5. **if** i == 9:
6. **break**
7. **print**()
8. # looping from 1 to 15
9. i = 0 # initial condition
10. **while** i < 15:
11. # When i has value 9, loop will jump to next iteration using continue. It will not print
12. **if** i == 9:
13. i += 3
14. **continue**
15. **else**:
16. # when i is not equal to 9, adding 2 and printing the value
17. **print**( i + 2, end = " ")
18. 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
4. # Exception raised in the try section
5. **try**:
6. d = var1 // var2 # this will raise a "divide by zero" exception.
7. **print**( d )
8. # this section will handle exception raised in try block
9. **except** ZeroDivisionError:
10. **print**("We cannot divide by zero")
11. **finally**:
12. # If exception is raised or not, this block will be executed every time
13. **print**("This is inside finally block")
14. # by using assert keyword we will check if var2 is 0
15. **print** ("The value of var1 / var2 is : ")
16. **assert** var2 != 0, "Divide by 0 error"
17. **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
4. **def** func\_with\_no\_return():
5. var = 10
6. **print**( func\_with\_return() )
7. **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
5. #Float Literal
6. float\_1 = 100.5
7. float\_2 = 1.5e2
8. #Complex Literal
9. a = 5+3.14j
10. **print**(x, y, z, u)
11. **print**(float\_1, float\_2)
12. **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
6. **print**("x is", x)
7. **print**("y is", y)
8. **print**("z is", z)
9. **print**("a:", a)
10. **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.

|  |  |
| --- | --- |
| **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
5. hence, a & b = 0011
6. a | b = 0111
7. a ^ b = 0100
8. ~ 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

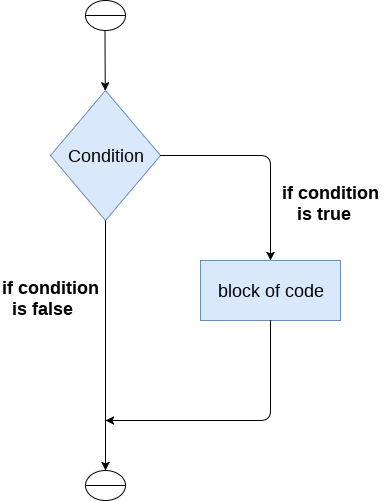
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

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

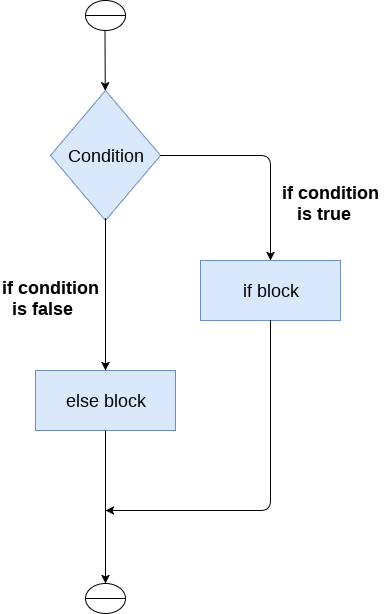
Enter c? 130

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

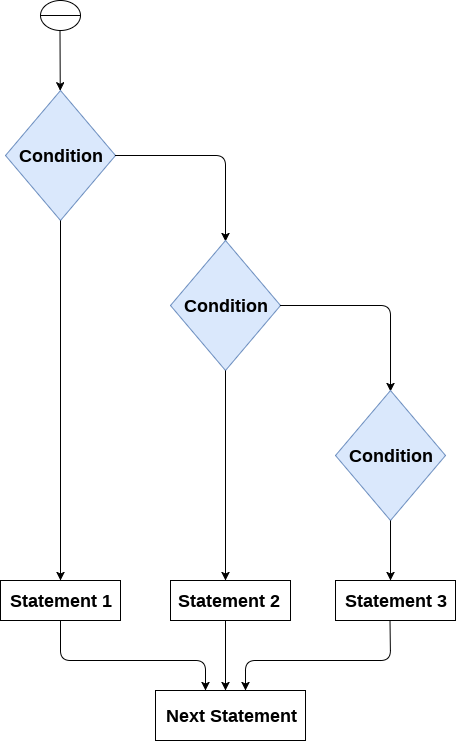
## The elif statement

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
3. **elif** expression 2:
4. # block of statements
5. **elif** expression 3:
6. # block of statements
7. **else**:
8. # 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.

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
2. # Creating a sequence which is a tuple of numbers
3. numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]
4. # variable to store the square of the number
5. square = 0
6. # Creating an empty list
7. squares = []
8. # Creating a for loop
9. **for** value **in** numbers:
10. square = value \*\* 2
11. squares.append(square)
12. **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
2. string = "Python Loop"
3. # Initiating a loop
4. **for** s **in** a string:
5. # giving a condition in if block
6. **if** s == "o":
7. **print**("If block")
8. # if condition is not satisfied then else block will be executed
9. **else**:
10. **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)
5. # Initiating the loop
6. **for** value **in** tuple\_:
7. **if** value % 2 != 0:
8. **print**(value)
9. # giving an else statement
10. **else**:
11. **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
2. **print**(range(15))
3. **print**(list(range(15)))
4. **print**(list(range(4, 9)))
5. **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
2. tuple\_ = ("Python", "Loops", "Sequence", "Condition", "Range")
3. # iterating over tuple\_ using range() function
4. **for** iterator **in** range(len(tuple\_)):
5. **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
3. # Iterating through the while loop
4. **while** (counter < 10):
5. counter = counter + 3
6. **print**("Python Loops") # Executed untile condition is met
7. # Once the condition of while loop gives False this statement will be executed
8. **else**:
9. **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
2. # Initiating the loop
3. **for** string **in** "Python Loops":
4. **if** string == "o" **or** string == "p" **or** string == "t":
5. **continue**
6. **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

### for value in sequence:

1. {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**

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]
5. # initializing a variable that will store the sum
6. sum\_ = 0
7. # using for loop to iterate over the list
8. **for** num **in** numbers:
9. sum\_ = sum\_ + num \*\* 2
10. **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
2. # creating the list of numbers
3. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
4. # initializing a variable that will store the sum
5. sum\_ = 0
6. # using for loop to iterate over list
7. **for** num **in** range( len(numbers) ):
8. sum\_ = sum\_ + numbers[num] \*\* 2
9. **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'
4. # Creating a dictionary of records of the students
5. records = {'Itika': 90, 'Arshia': 92, 'Peter': 46}
6. **def** marks( student\_name ):
7. **for** a\_student **in** record: # for loop will iterate over the keys of the dictionary
8. **if** a\_student == student\_name:
9. **return** records[ a\_student ]
10. **break**
11. **else**:
12. **return** f'There is no student of name {student\_name} in the records'
13. # giving the function marks() name of two students
14. **print**( f"Marks of {student\_name\_1} are: ", marks( student\_name\_1 ) )
15. **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**

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.

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

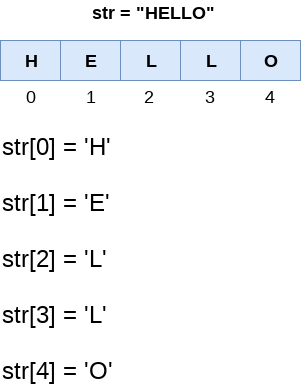
Triple quotes are generally used for

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

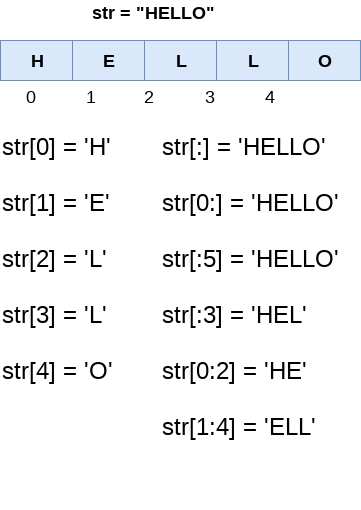
L

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

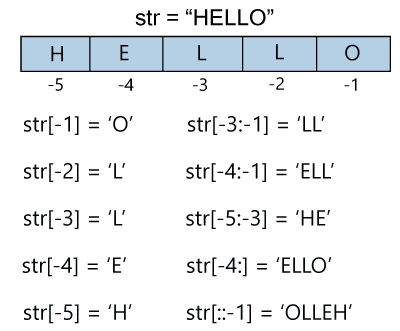
AVAT

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

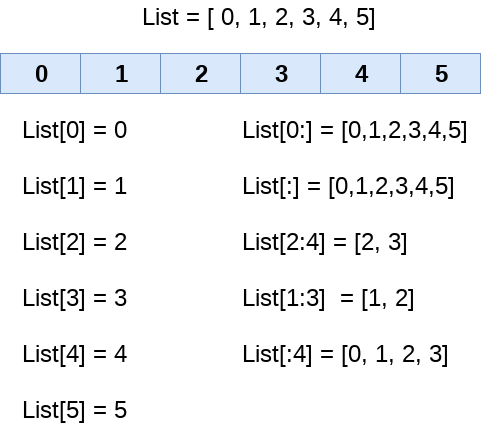
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

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.

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")

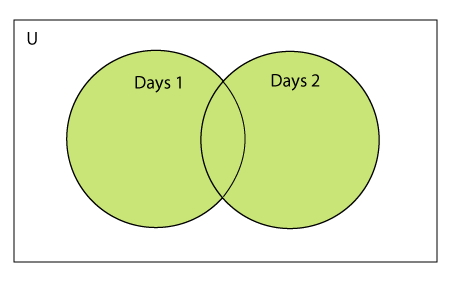
KeyError: 'Jan'

## 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**

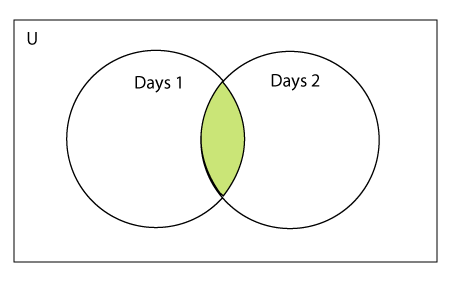
1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. **print**(Days1.union(Days2)) #printing the union of the sets

**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.

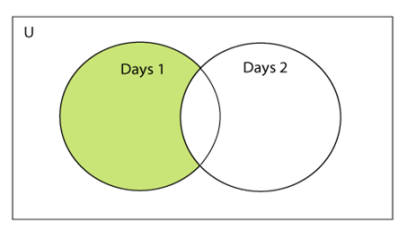
1. a = {"Devansh", "bob", "castle"}
2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
5. a.intersection\_update(b, c)
7. **print**(a)

**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**

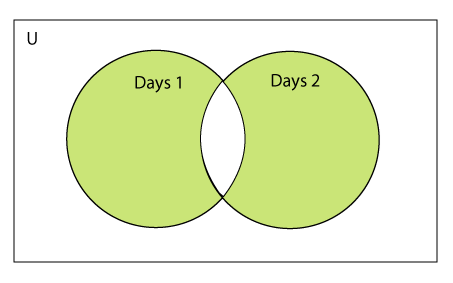
1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print**(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2

**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:**

Play Videox[](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.

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print("Name : %s" %Employee["Name"])

print("Age : %d" %Employee["Age"])

print("Salary : %d" %Employee["salary"])

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:**

# Creating an empty Dictionary

1. Dict = {}
2. print("Empty Dictionary: ")
3. print(Dict)
5. # Adding elements to dictionary one at a time
6. Dict[0] = 'Peter'
7. Dict[2] = 'Joseph'
8. Dict[3] = 'Ricky'
9. print("\nDictionary after adding 3 elements: ")
10. print(Dict)
12. # Adding set of values
13. # with a single Key
14. # The Emp\_ages doesn't exist to dictionary
15. Dict['Emp\_ages'] = 20, 33, 24
16. print("\nDictionary after adding 3 elements: ")
17. print(Dict)
19. # Updating existing Key's Value
20. Dict[3] = 'JavaTpoint'
21. print("\nUpdated key value: ")
22. 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)
12. Output:
13. Empty Dictionary:
14. {}
15. Dictionary after adding 3 elements:
16. {0: 'Peter', 2: 'Joseph', 3: 'Ricky'}
17. Dictionary after adding 3 elements:
18. {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}
19. Updated key value:
20. {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

## Deleting elements using del keyword

1. The items of the dictionary can be deleted by using the **del** keyword as given below.
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
3. print(type(Employee))
4. print("printing Employee data .... ")
5. print(Employee)
6. print("Deleting some of the employee data")
7. del Employee["Name"]
8. del Employee["Company"]
9. print("printing the modified information ")
10. print(Employee)
11. print("Deleting the dictionary: Employee");
12. del Employee
13. print("Lets try to print it again ");
14. print(Employee)
15. **Output:**
16. <class 'dict'>
17. printing Employee data ....
18. {'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
19. Deleting some of the employee data
20. printing the modified information
21. {'Age': 29, 'salary': 25000}
22. Deleting the dictionary: Employee
23. Lets try to print it again
24. NameError: name 'Employee' is not defined
25. The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.
26. **Using pop() method**
27. The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.
28. # Creating a Dictionary
29. Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
30. # Deleting a key
31. # using pop() method
32. pop\_ele = Dict.pop(3)
33. print(Dict)
34. **Output:**
35. {1: 'JavaTpoint', 2: 'Peter'}
36. 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

1. A dictionary can be iterated using for loop as given below.

### Example 1

1. **# for loop to print all the keys of a dictionary**
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
3. for x in Employee:
4. print(x)
5. **Output:**

Name

Age

salary

Company

### Example 2

1. **#for loop to print all the values of the dictionary**
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
3. for x in Employee:
4. print(Employee[x])
5. **Output:**
6. John
7. 29
8. 25000
9. GOOGLE
10. Example - 3
11. **#for loop to print the values of the dictionary by using values() method.**
12. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
13. for x in Employee.values():
14. print(x)
15. **Output:**
16. John
17. 29
18. 25000
19. GOOGLE
20. Example 4
21. **#for loop to print the items of the dictionary by using items() method.**
22. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
23. for x in Employee.items():
24. print(x)
25. **Output:**
26. ('Name', 'John')
27. ('Age', 29)
28. ('salary', 25000)
29. ('Company', 'GOOGLE')
30. Properties of Dictionary keys
31. 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.
32. Consider the following example.
33. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
34. for x,y in Employee.items():
35. print(x,y)
36. **Output:**
37. Name John
38. Age 29
39. Salary 25000
40. Company GOOGLE
41. 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.
42. Consider the following example.
43. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
44. for x,y in Employee.items():
45. print(x,y)
46. **Output:**
47. Traceback (most recent call last):
48. File "dictionary.py", line 1, in
49. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
50. 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.

* 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:**

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.

### 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.

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

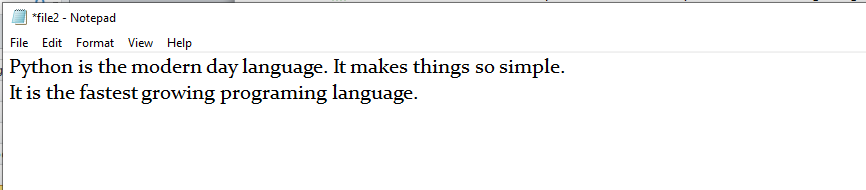
1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")
4. # appending the content to the file
5. fileptr.write('''''Python is the modern day language. It makes things so simple.
6. It is the fastest-growing programing language''')
8. # closing 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

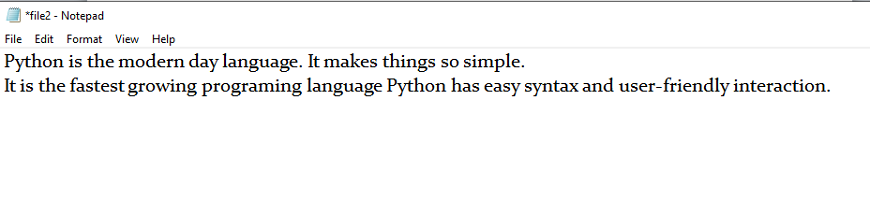
1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
4. #overwriting the content of the file
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
7. #closing the opened file
8. fileptr.close()

**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.

**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:

**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

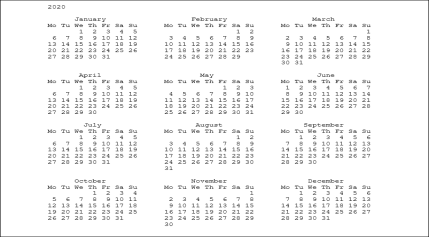
Printing the calendar of whole year

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.

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 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.

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 OOPs Concepts

Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In [Python](https://www.javatpoint.com/python-tutorial), we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

Major principles of object-oriented programming system are given below.

* Class
* Object
* Method
* Inheritance
* Polymorphism
* Data Abstraction
* Encapsulation

## Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

**Syntax**

1. **class** ClassName:
2. <statement-1>
3. .
4. .
5. <statement-N>

## Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute \_\_doc\_\_, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

**Example:**

1. **class** car:
2. **def** \_\_init\_\_(self,modelname, year):
3. self.modelname = modelname
4. self.year = year
5. **def** display(self):
6. **print**(self.modelname,self.year)
8. c1 = car("Toyota", 2016)
9. c1.display()

**Output:**

Toyota 2016

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values. We will learn more about class and object in the next tutorial.

## Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

## Inheritance

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

## Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

## Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

## Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

## Object-oriented vs. Procedure-oriented Programming languages

The difference between object-oriented and procedure-oriented programming is given below:

|  |  |  |
| --- | --- | --- |
| **Index** | **Object-oriented Programming** | **Procedural Programming** |
| 1. | Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| 2. | It makes the development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when the project becomes lengthy. |
| 3. | It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions. |
| 4. | It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way for data binding, so it is less secure. |
| 5. | Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Example of procedural languages are: C, Fortran, Pascal, VB etc. |

# Python Class and Objects

We have already discussed in previous tutorial, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

In this section of the tutorial, we will discuss creating classes and objects in Python. We will also discuss how a class attribute is accessed by using the object.

## Creating classes in Python

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

**Syntax**

1. **class** ClassName:
2. #statement\_suite

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.\_\_doc\_\_.** A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

**Example**

1. **class** Employee:
2. id = 10
3. name = "Devansh"
4. **def** display (self):
5. **print**(self.id,self.name)

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

### The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

## Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

1. <object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

**Example**

1. **class** Employee:
2. id = 10
3. name = "John"
4. **def** display (self):
5. **print**("ID: %d \nName: %s"%(self.id,self.name))
6. # Creating a emp instance of Employee class
7. emp = Employee()
8. emp.display()

**Output:**

ID: 10

Name: John

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

## Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

**Example**

1. **class** Employee:
2. id = 10
3. name = "John"
5. **def** display(self):
6. **print**("ID: %d \nName: %s" % (self.id, self.name))
7. # Creating a emp instance of Employee class
9. emp = Employee()
11. # Deleting the property of object
12. **del** emp.id
13. # Deleting the object itself
14. **del** emp
15. emp.display()

It will through the Attribute error because we have deleted the object **emp**.

# Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

## Creating the constructor in python

In Python, the method the **\_\_init\_\_()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **\_\_init\_\_()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

### Example

1. **class** Employee:
2. **def** \_\_init\_\_(self, name, id):
3. self.id = id
4. self.name = name
6. **def** display(self):
7. **print**("ID: %d \nName: %s" % (self.id, self.name))

10. emp1 = Employee("John", 101)
11. emp2 = Employee("David", 102)
13. # accessing display() method to print employee 1 information
15. emp1.display()
17. # accessing display() method to print employee 2 information
18. emp2.display()

**Output:**

ID: 101

Name: John

ID: 102

Name: David

### Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

### Example

1. **class** Student:
2. count = 0
3. **def** \_\_init\_\_(self):
4. Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. **print**("The number of students:",Student.count)

**Output:**

The number of students: 3

## Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

### Example

1. **class** Student:
2. # Constructor - non parameterized
3. **def** \_\_init\_\_(self):
4. **print**("This is non parametrized constructor")
5. **def** show(self,name):
6. **print**("Hello",name)
7. student = Student()
8. student.show("John")

## Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

### Example

1. **class** Student:
2. # Constructor - parameterized
3. **def** \_\_init\_\_(self, name):
4. **print**("This is parametrized constructor")
5. self.name = name
6. **def** show(self):
7. **print**("Hello",self.name)
8. student = Student("John")
9. student.show()

**Output:**

This is parametrized constructor

Hello John

## Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

### Example

1. **class** Student:
2. roll\_num = 101
3. name = "Joseph"
5. **def** display(self):
6. **print**(self.roll\_num,self.name)
8. st = Student()
9. st.display()

**Output:**

101 Joseph

## More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

### Example

1. **class** Student:
2. **def** \_\_init\_\_(self):
3. **print**("The First Constructor")
4. **def** \_\_init\_\_(self):
5. **print**("The second contructor")
7. st = Student()

**Output:**

The Second Constructor

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

#### Note: The constructor overloading is not allowed in Python.

## Python built-in class functions

The built-in functions defined in the class are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | getattr(obj,name,default) | It is used to access the attribute of the object. |
| 2 | setattr(obj, name,value) | It is used to set a particular value to the specific attribute of an object. |
| 3 | delattr(obj, name) | It is used to delete a specific attribute. |
| 4 | hasattr(obj, name) | It returns true if the object contains some specific attribute. |

### Example

1. **class** Student:
2. **def** \_\_init\_\_(self, name, id, age):
3. self.name = name
4. self.id = id
5. self.age = age
7. # creates the object of the class Student
8. s = Student("John", 101, 22)
10. # prints the attribute name of the object s
11. **print**(getattr(s, 'name'))
13. # reset the value of attribute age to 23
14. setattr(s, "age", 23)
16. # prints the modified value of age
17. **print**(getattr(s, 'age'))
19. # prints true if the student contains the attribute with name id
21. **print**(hasattr(s, 'id'))
22. # deletes the attribute age
23. delattr(s, 'age')
25. # this will give an error since the attribute age has been deleted
26. **print**(s.age)

**Output:**

John

23

True

AttributeError: 'Student' object has no attribute 'age'

## Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | \_\_dict\_\_ | It provides the dictionary containing the information about the class namespace. |
| 2 | \_\_doc\_\_ | It contains a string which has the class documentation |
| 3 | \_\_name\_\_ | It is used to access the class name. |
| 4 | \_\_module\_\_ | It is used to access the module in which, this class is defined. |
| 5 | \_\_bases\_\_ | It contains a tuple including all base classes. |

### Example

1. **class** Student:
2. **def** \_\_init\_\_(self,name,id,age):
3. self.name = name;
4. self.id = id;
5. self.age = age
6. **def** display\_details(self):
7. **print**("Name:%s, ID:%d, age:%d"%(self.name,self.id))
8. s = Student("John",101,22)
9. **print**(s.\_\_doc\_\_)
10. **print**(s.\_\_dict\_\_)
11. **print**(s.\_\_module\_\_)

**Output:**

None

{'name': 'John', 'id': 101, 'age': 22}

\_\_main\_\_

# Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Python Inheritance

### Syntax

1. **class** derived-**class**(base **class**):
2. <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

### Syntax

1. **class** derive-**class**(<base **class** 1>, <base **class** 2>, ..... <base **class** n>):
2. <**class** - suite>

### Example 1

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. d = Dog()
9. d.bark()
10. d.speak()

**Output:**

dog barking

Animal Speaking

## Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Python Inheritance

The syntax of multi-level inheritance is given below.

### Syntax

1. **class** class1:
2. <**class**-suite>
3. **class** class2(class1):
4. <**class** suite>
5. **class** class3(class2):
6. <**class** suite>
7. .
8. .

### Example

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. **class** DogChild(Dog):
10. **def** eat(self):
11. **print**("Eating bread...")
12. d = DogChild()
13. d.bark()
14. d.speak()
15. d.eat()

**Output:**

dog barking

Animal Speaking

Eating bread...

## Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

Python Inheritance

The syntax to perform multiple inheritance is given below.

### Syntax

1. **class** Base1:
2. <**class**-suite>
4. **class** Base2:
5. <**class**-suite>
6. .
7. .
8. .
9. **class** BaseN:
10. <**class**-suite>
12. **class** Derived(Base1, Base2, ...... BaseN):
13. <**class**-suite>

### Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(d.Summation(10,20))
12. **print**(d.Multiplication(10,20))
13. **print**(d.Divide(10,20))

**Output:**

30

200

0.5

## The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

### Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(issubclass(Derived,Calculation2))
12. **print**(issubclass(Calculation1,Calculation2))

**Output:**

True

False

## The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

### Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a\*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(isinstance(d,Derived))

**Output:**

True

## Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

### Example

1. **class** Animal:
2. **def** speak(self):
3. **print**("speaking")
4. **class** Dog(Animal):
5. **def** speak(self):
6. **print**("Barking")
7. d = Dog()
8. d.speak()

**Output:**

Barking

### Real Life Example of method overriding

1. **class** Bank:
2. **def** getroi(self):
3. **return** 10;
4. **class** SBI(Bank):
5. **def** getroi(self):
6. **return** 7;
8. **class** ICICI(Bank):
9. **def** getroi(self):
10. **return** 8;
11. b1 = Bank()
12. b2 = SBI()
13. b3 = ICICI()
14. **print**("Bank Rate of interest:",b1.getroi());
15. **print**("SBI Rate of interest:",b2.getroi());
16. **print**("ICICI Rate of interest:",b3.getroi());

**Output:**

Bank Rate of interest: 10

SBI Rate of interest: 7

ICICI Rate of interest: 8

## Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (\_\_\_) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

### Example

1. **class** Employee:
2. \_\_count = 0;
3. **def** \_\_init\_\_(self):
4. Employee.\_\_count = Employee.\_\_count+1
5. **def** display(self):
6. **print**("The number of employees",Employee.\_\_count)
7. emp = Employee()
8. emp2 = Employee()
9. **try**:
10. **print**(emp.\_\_count)
11. **finally**:
12. emp.display()

**Output:**

The number of employees 2

AttributeError: 'Employee' object has no attribute '\_\_count'

# Abstraction in Python

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**

In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialing, etc., but we don't know how these operations are happening in the background. Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.

That is exactly the abstraction that works in the [object-oriented concept](https://www.javatpoint.com/python-oops-concepts).

## Why Abstraction is Important?

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency. Next, we will learn how we can achieve abstraction using the [Python program](https://www.javatpoint.com/python-programs).

## Abstraction classes in Python

In [Python](https://www.javatpoint.com/python-tutorial), abstraction can be achieved by using abstract classes and interfaces.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

**Syntax**

1. from abc **import** ABC
2. **class** ClassName(ABC):

We import the ABC class from the **abc** module.

## Abstract Base Classes

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

## Working of the Abstract Classes

Unlike the other high-level language, Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base. We use the **@abstractmethod** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method. Let's understand the following example.

**Example -**

1. # Python program demonstrate
2. # **abstract** base **class** work
3. from abc **import** ABC, abstractmethod
4. **class** Car(ABC):
5. def mileage(self):
6. pass
8. **class** Tesla(Car):
9. def mileage(self):
10. print("The mileage is 30kmph")
11. **class** Suzuki(Car):
12. def mileage(self):
13. print("The mileage is 25kmph ")
14. **class** Duster(Car):
15. def mileage(self):
16. print("The mileage is 24kmph ")
18. **class** Renault(Car):
19. def mileage(self):
20. print("The mileage is 27kmph ")
22. # Driver code
23. t= Tesla ()
24. t.mileage()
26. r = Renault()
27. r.mileage()
29. s = Suzuki()
30. s.mileage()
31. d = Duster()
32. d.mileage()

**Output:**

The mileage is 30kmph

The mileage is 27kmph

The mileage is 25kmph

The mileage is 24kmph

**Explanation -**

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Let's understand another example.

Let's understand another example.

**Example -**

1. # Python program to define
2. # **abstract** **class**
4. from abc **import** ABC
6. **class** Polygon(ABC):
8. # **abstract** method
9. def sides(self):
10. pass
12. **class** Triangle(Polygon):

15. def sides(self):
16. print("Triangle has 3 sides")
18. **class** Pentagon(Polygon):

21. def sides(self):
22. print("Pentagon has 5 sides")
24. **class** Hexagon(Polygon):
26. def sides(self):
27. print("Hexagon has 6 sides")
29. **class** square(Polygon):
31. def sides(self):
32. print("I have 4 sides")
34. # Driver code
35. t = Triangle()
36. t.sides()
38. s = square()
39. s.sides()
41. p = Pentagon()
42. p.sides()
44. k = Hexagon()
45. K.sides()

**Output:**

Triangle has 3 sides

Square has 4 sides

Pentagon has 5 sides

Hexagon has 6 sides

**Explanation -**

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()** method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

## Points to Remember

Below are the points which we should remember about the abstract base class in Python.

* An Abstract class can contain the both method normal and abstract method.
* An Abstract cannot be instantiated; we cannot create objects for the abstract class.

Abstraction is essential to hide the core functionality from the users. We have covered the all the basic concepts of Abstraction in Python.

# Environment Setup

To build the real world applications, connecting with the databases is the necessity for the programming languages. However, python allows us to connect our application to the databases like MySQL, SQLite, MongoDB, and many others.

In this section of the tutorial, we will discuss Python - MySQL connectivity, and we will perform the database operations in python. We will also cover the Python connectivity with the databases like MongoDB and SQLite later in this tutorial.

## Install mysql.connector

To connect the python application with the MySQL database, we must import the mysql.connector module in the program.

The mysql.connector is not a built-in module that comes with the python installation. We need to install it to get it working.

Execute the following command to install it using pip installer.

1. >  python -m pip install mysql-connector

**Or follow the following steps.**

1. Click the link:

<https://files.pythonhosted.org/packages/8f/6d/fb8ebcbbaee68b172ce3dfd08c7b8660d09f91d8d5411298bcacbd309f96/mysql-connector-python-8.0.13.tar.gz> to download the source code.

2. Extract the archived file.

3. Open the terminal (CMD for windows) and change the present working directory to the source code directory.

1. $  cd mysql-connector-python-8.0.13/

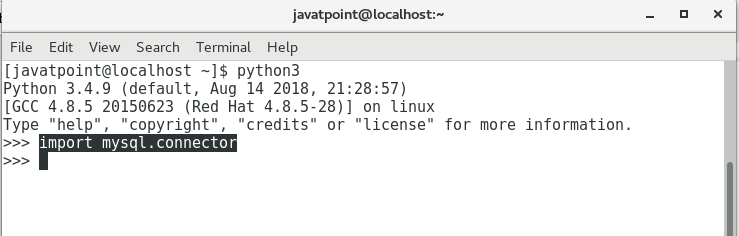
4. Run the file named setup.py with python (python3 in case you have also installed python 2) with the parameter build.

1. $ python setup.py build

5. Run the following command to install the mysql-connector.

1. $ python setup.py install

This will take a bit of time to install mysql-connector for python. We can verify the installation once the process gets over by importing mysql-connector on the python shell.



Hence, we have successfully installed mysql-connector for python on our system.

# Database Connection

In this section of the tutorial, we will discuss the steps to connect the python application to the database.

There are the following steps to connect a python application to our database.

1. Import mysql.connector module
2. Create the connection object.
3. Create the cursor object
4. Execute the query

## Creating the connection

To create a connection between the MySQL database and the python application, the connect() method of mysql.connector module is used.

Pass the database details like HostName, username, and the database password in the method call. The method returns the connection object.

The syntax to use the connect() is given below.

1. Connection-Object= mysql.connector.connect(host = <host-name> , user = <username> , passwd = <password> )

Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
6. #printing the connection object
7. **print**(myconn)

**Output:**

<mysql.connector.connection.MySQLConnection object at 0x7fb142edd780>

Here, we must notice that we can specify the database name in the connect() method if we want to connect to a specific database.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", database = "mydb")
6. #printing the connection object
7. **print**(myconn)

**Output:**

<mysql.connector.connection.MySQLConnection object at 0x7ff64aa3d7b8>

## Creating a cursor object

The cursor object can be defined as an abstraction specified in the Python DB-API 2.0. It facilitates us to have multiple separate working environments through the same connection to the database. We can create the cursor object by calling the 'cursor' function of the connection object. The cursor object is an important aspect of executing queries to the databases.

The syntax to create the cursor object is given below.

1. <my\_cur>  = conn.cursor()

### Example

1. **import** mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", database = "mydb")
5. #printing the connection object
6. **print**(myconn)
8. #creating the cursor object
9. cur = myconn.cursor()
11. **print**(cur)

**Output:**

<mysql.connector.connection.MySQLConnection object at 0x7faa17a15748>

MySQLCursor: (Nothing executed yet)

# Creating new databases

In this section of the tutorial, we will create the new database PythonDB.

## Getting the list of existing databases

We can get the list of all the databases by using the following MySQL query.

1. >  show databases;

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. dbs = cur.execute("show databases")
11. **except**:
12. myconn.rollback()
13. **for** x **in** cur:
14. **print**(x)
15. myconn.close()

**Output:**

('EmployeeDB',)

('Test',)

('TestDB',)

('information\_schema',)

('javatpoint',)

('javatpoint1',)

('mydb',)

('mysql',)

('performance\_schema',)

('testDB',)

## Creating the new database

The new database can be created by using the following SQL query.

1. >  create database <database-name>

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #creating a new database
11. cur.execute("create database PythonDB2")
13. #getting the list of all the databases which will now include the new database PythonDB
14. dbs = cur.execute("show databases")
16. **except**:
17. myconn.rollback()
19. **for** x **in** cur:
20. **print**(x)
22. myconn.close()

**Output:**

('EmployeeDB',)

('PythonDB',)

('Test',)

('TestDB',)

('anshika',)

('information\_schema',)

('javatpoint',)

('javatpoint1',)

('mydb',)

('mydb1',)

('mysql',)

('performance\_schema',)

('testDB',)

# Creating the table

In this section of the tutorial, we will create the new table Employee. We have to mention the database name while establishing the connection object.

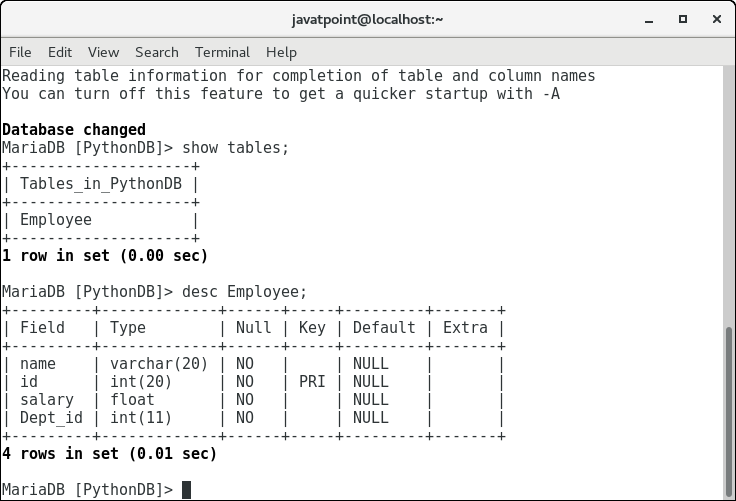
We can create the new table by using the CREATE TABLE statement of SQL. In our database PythonDB, the table Employee will have the four columns, i.e., name, id, salary, and department\_id initially.

The following query is used to create the new table Employee.

1. >  create table Employee (name varchar(20) **not** null, id int primary key, salary float **not** null, Dept\_Id int **not** null)

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Creating a table with name Employee having four columns i.e., name, id, salary, and department id
11. dbs = cur.execute("create table Employee(name varchar(20) not null, id int(20) not null primary key, salary float not null, Dept\_id int not null)")
12. **except**:
13. myconn.rollback()
15. myconn.close()



Now, we may check that the table Employee is present in the database.

## Alter Table

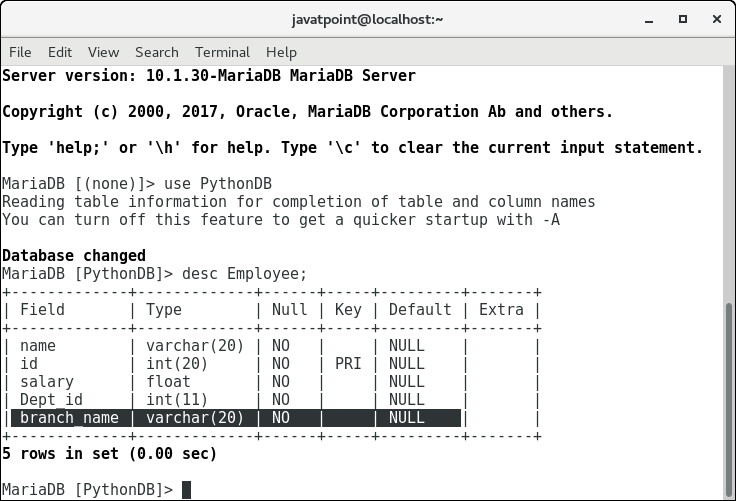
Sometimes, we may forget to create some columns, or we may need to update the table schema. The alter statement used to alter the table schema if required. Here, we will add the column branch\_name to the table Employee. The following SQL query is used for this purpose.

1. alter table Employee add branch\_name varchar(20) **not** null

Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #adding a column branch name to the table Employee
11. cur.execute("alter table Employee add branch\_name varchar(20) not null")
12. **except**:
13. myconn.rollback()
15. myconn.close()



# Insert Operation

## Adding a record to the table

The **INSERT INTO** statement is used to add a record to the table. In python, we can mention the format specifier (%s) in place of values.

We provide the actual values in the form of tuple in the execute() method of the cursor.

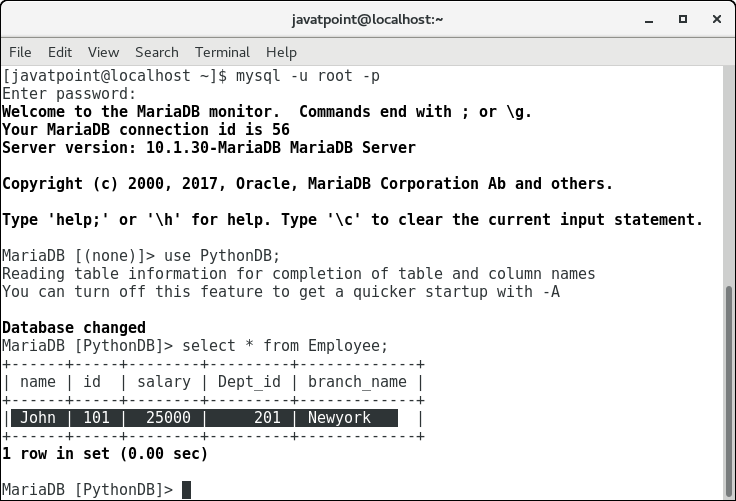
Consider the following example.

### Example

1. **import** mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
6. sql = "insert into Employee(name, id, salary, dept\_id, branch\_name) values (%s, %s, %s, %s, %s)"
8. #The row values are provided in the form of tuple
9. val = ("John", 110, 25000.00, 201, "Newyork")
11. **try**:
12. #inserting the values into the table
13. cur.execute(sql,val)
15. #commit the transaction
16. myconn.commit()
18. **except**:
19. myconn.rollback()
21. **print**(cur.rowcount,"record inserted!")
22. myconn.close()

**Output:**

1 record inserted!



**Insert multiple rows**

We can also insert multiple rows at once using the python script. The multiple rows are mentioned as the list of various tuples.

Each element of the list is treated as one particular row, whereas each element of the tuple is treated as one particular column value (attribute).

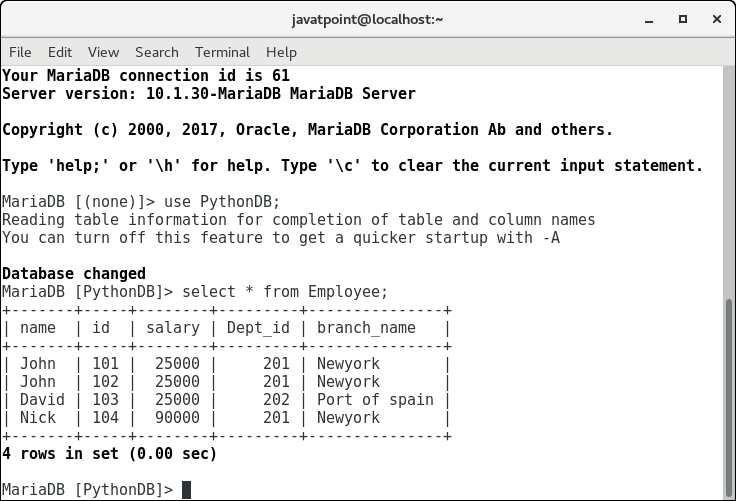
Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
8. sql = "insert into Employee(name, id, salary, dept\_id, branch\_name) values (%s, %s, %s, %s, %s)"
9. val = [("John", 102, 25000.00, 201, "Newyork"),("David",103,25000.00,202,"Port of spain"),("Nick",104,90000.00,201,"Newyork")]
11. **try**:
12. #inserting the values into the table
13. cur.executemany(sql,val)
15. #commit the transaction
16. myconn.commit()
17. **print**(cur.rowcount,"records inserted!")
19. **except**:
20. myconn.rollback()
22. myconn.close()

**Output:**

3 records inserted!



**Row ID**

In SQL, a particular row is represented by an insertion id which is known as row id. We can get the last inserted row id by using the attribute lastrowid of the cursor object.

Consider the following example.

### Example

1. **import** mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
7. sql = "insert into Employee(name, id, salary, dept\_id, branch\_name) values (%s, %s, %s, %s, %s)"
9. val = ("Mike",105,28000,202,"Guyana")
11. **try**:
12. #inserting the values into the table
13. cur.execute(sql,val)
15. #commit the transaction
16. myconn.commit()
18. #getting rowid
19. **print**(cur.rowcount,"record inserted! id:",cur.lastrowid)
21. **except**:
22. myconn.rollback()
24. myconn.close()

**Output:**

1 record inserted! Id: 0

[**Next →**](https://www.javatpoint.com/python-mysql-update-operation)[**← Prev**](https://www.javatpoint.com/python-mysql-insert-operation)

# Read Operation

The SELECT statement is used to read the values from the databases. We can restrict the output of a select query by using various clause in SQL like where, limit, etc.

Python provides the fetchall() method returns the data stored inside the table in the form of rows. We can iterate the result to get the individual rows.

In this section of the tutorial, we will extract the data from the database by using the python script. We will also format the output to print it on the console.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select \* from Employee")
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
15. #printing the result
17. **for** x **in** result:
18. **print**(x);
19. **except**:
20. myconn.rollback()
22. myconn.close()

**Output:**

('John', 101, 25000.0, 201, 'Newyork')

('John', 102, 25000.0, 201, 'Newyork')

('David', 103, 25000.0, 202, 'Port of spain')

('Nick', 104, 90000.0, 201, 'Newyork')

('Mike', 105, 28000.0, 202, 'Guyana')

## Reading specific columns

We can read the specific columns by mentioning their names instead of using star (\*).

In the following example, we will read the name, id, and salary from the Employee table and print it on the console.

### Example

1. **import** mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
6. **try**:
7. #Reading the Employee data
8. cur.execute("select name, id, salary from Employee")
10. #fetching the rows from the cursor object
11. result = cur.fetchall()
12. #printing the result
13. **for** x **in** result:
14. **print**(x);
15. **except**:
16. myconn.rollback()
17. myconn.close()

**Output:**

('John', 101, 25000.0)

('John', 102, 25000.0)

('David', 103, 25000.0)

('Nick', 104, 90000.0)

('Mike', 105, 28000.0)

## The fetchone() method

The fetchone() method is used to fetch only one row from the table. The fetchone() method returns the next row of the result-set.

Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select name, id, salary from Employee")
13. #fetching the first row from the cursor object
14. result = cur.fetchone()
16. #printing the result
17. **print**(result)
19. **except**:
20. myconn.rollback()
22. myconn.close()

**Output:**

('John', 101, 25000.0)

## Formatting the result

We can format the result by iterating over the result produced by the fetchall() or fetchone() method of cursor object since the result exists as the tuple object which is not readable.

Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
11. #Reading the Employee data
12. cur.execute("select name, id, salary from Employee")
14. #fetching the rows from the cursor object
15. result = cur.fetchall()
17. **print**("Name    id    Salary");
18. **for** row **in** result:
19. **print**("%s    %d    %d"%(row[0],row[1],row[2]))
20. **except**:
21. myconn.rollback()
23. myconn.close()

**Output:**

Name id Salary

John 101 25000

John 102 25000

David 103 25000

Nick 104 90000

Mike 105 28000

## Using where clause

We can restrict the result produced by the select statement by using the where clause. This will extract only those columns which satisfy the where condition.

Consider the following example.

### Example: printing the names that start with j

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select name, id, salary from Employee where name like 'J%'")
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
16. **print**("Name    id    Salary");
18. **for** row **in** result:
19. **print**("%s    %d    %d"%(row[0],row[1],row[2]))
20. **except**:
21. myconn.rollback()
23. myconn.close()

**Output:**

Name id Salary

John 101 25000

John 102 25000

### Example: printing the names with id = 101, 102, and 103

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select name, id, salary from Employee where id in (101,102,103)")
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
16. **print**("Name    id    Salary");
18. **for** row **in** result:
19. **print**("%s    %d    %d"%(row[0],row[1],row[2]))
20. **except**:
21. myconn.rollback()
23. myconn.close()

**Output:**

Name id Salary

John 101 25000

John 102 25000

David 103 2500

## Ordering the result

The ORDER BY clause is used to order the result. Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select name, id, salary from Employee order by name")
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
16. **print**("Name    id    Salary");
18. **for** row **in** result:
19. **print**("%s    %d    %d"%(row[0],row[1],row[2]))
20. **except**:
21. myconn.rollback()
23. myconn.close()

**Output:**

Name id Salary

David 103 25000

John 101 25000

John 102 25000

Mike 105 28000

Nick 104 90000

## Order by DESC

This orders the result in the decreasing order of a particular column.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Reading the Employee data
11. cur.execute("select name, id, salary from Employee order by name desc")
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
16. #printing the result
17. **print**("Name    id    Salary");
18. **for** row **in** result:
19. **print**("%s    %d    %d"%(row[0],row[1],row[2]))
21. **except**:
22. myconn.rollback()
24. myconn.close()

**Output:**

Name id Salary

Nick 104 90000

Mike 105 28000

John 101 25000

John 102 25000

David 103 25000

# Update Operation

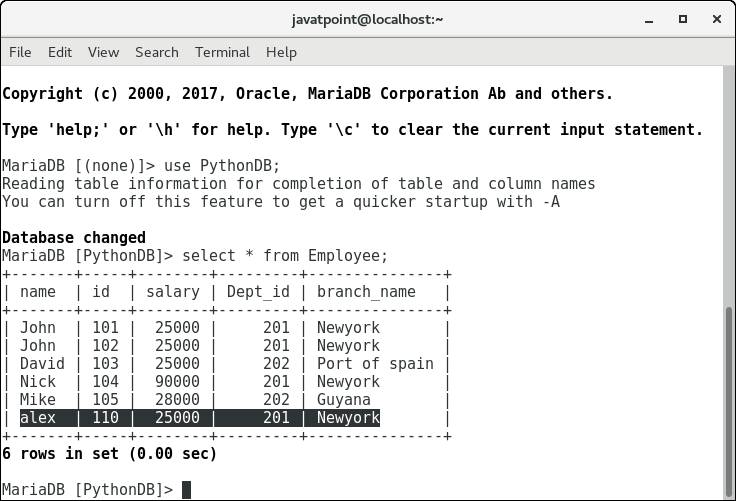
The UPDATE-SET statement is used to update any column inside the table. The following SQL query is used to update a column.

1. >  update Employee set name = 'alex' where id = 110

Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #updating the name of the employee whose id is 110
11. cur.execute("update Employee set name = 'alex' where id = 110")
12. myconn.commit()
13. **except**:
15. myconn.rollback()
17. myconn.close()



## Delete Operation

The DELETE FROM statement is used to delete a specific record from the table. Here, we must impose a condition using WHERE clause otherwise all the records from the table will be removed.

The following SQL query is used to delete the employee detail whose id is 110 from the table.

1. >  delete **from** Employee where id = 110

Consider the following example.

### Example

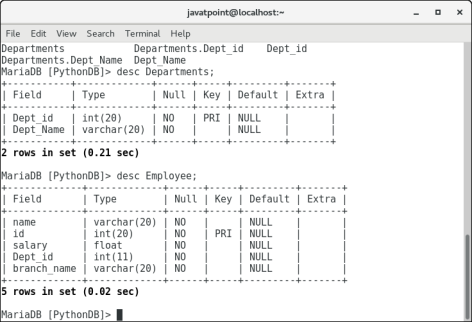
1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #Deleting the employee details whose id is 110
11. cur.execute("delete from Employee where id = 110")
12. myconn.commit()
13. **except**:
15. myconn.rollback()
17. myconn.close()

# Join Operation

We can combine the columns from two or more tables by using some common column among them by using the join statement.

We have only one table in our database, let's create one more table Departments with two columns department\_id and department\_name.

1. create table Departments (Dept\_id int(20) primary key **not** null, Dept\_Name varchar(20) **not** null);

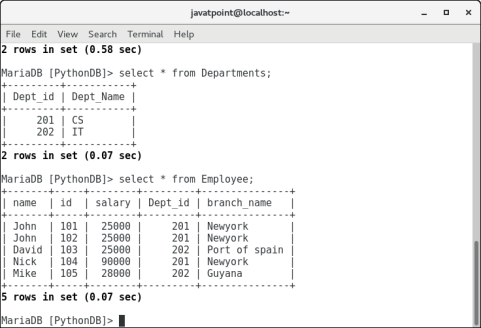


As we have created a new table Departments as shown in the above image. However, we haven't yet inserted any value inside it.

Let's insert some Departments ids and departments names so that we can map this to our Employee table.

1. insert into Departments values (201, "CS");
2. insert into Departments values (202, "IT");

Let's look at the values inserted in each of the tables. Consider the following image.



Now, let's create a python script that joins the two tables on the common column, i.e., dept\_id.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #joining the two tables on departments\_id
11. cur.execute("select Employee.id, Employee.name, Employee.salary, Departments.Dept\_id, Departments.Dept\_Name from Departments join Employee on Departments.Dept\_id = Employee.Dept\_id")
12. **print**("ID    Name    Salary    Dept\_Id    Dept\_Name")
13. **for** row **in** cur:
14. **print**("%d    %s    %d    %d    %s"%(row[0], row[1],row[2],row[3],row[4]))
16. **except**:
17. myconn.rollback()
19. myconn.close()

**Output:**

ID Name Salary Dept\_Id Dept\_Name

101 John 25000 201 CS

102 John 25000 201 CS

103 David 25000 202 IT

104 Nick 90000 201 CS

105 Mike 28000 202 IT

## Right Join

Right join shows all the columns of the right-hand side table as we have two tables in the database PythonDB, i.e., Departments and Employee. We do not have any Employee in the table who is not working for any department (Employee for which department id is null). However, to understand the concept of right join let's create the one.

Execute the following query on the MySQL server.

1. insert into Employee(name, id, salary, branch\_name) values ("Alex",108,29900,"Mumbai");

This will insert an employee Alex who doesn't work for any department (department id is null).

Now, we have an employee in the Employee table whose department id is not present in the Departments table. Let's perform the right join on the two tables now.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #joining the two tables on departments\_id
11. result = cur.execute("select Employee.id, Employee.name, Employee.salary, Departments.Dept\_id, Departments.Dept\_Name from Departments right join Employee on Departments.Dept\_id = Employee.Dept\_id")
13. **print**("ID    Name    Salary    Dept\_Id    Dept\_Name")
15. **for** row **in** cur:
16. **print**(row[0],"    ", row[1],"    ",row[2],"    ",row[3],"    ",row[4])


20. **except**:
21. myconn.rollback()
23. myconn.close()

**Output:**

ID Name Salary Dept\_Id Dept\_Name

101 John 25000.0 201 CS

102 John 25000.0 201 CS

103 David 25000.0 202 IT

104 Nick 90000.0 201 CS

105 Mike 28000.0 202 IT

108 Alex 29900.0 None None

## Left Join

The left join covers all the data from the left-hand side table. It has just opposite effect to the right join. Consider the following example.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. #joining the two tables on departments\_id
11. result = cur.execute("select Employee.id, Employee.name, Employee.salary, Departments.Dept\_id, Departments.Dept\_Name from Departments left join Employee on Departments.Dept\_id = Employee.Dept\_id")
12. **print**("ID    Name    Salary    Dept\_Id    Dept\_Name")
13. **for** row **in** cur:
14. **print**(row[0],"    ", row[1],"    ",row[2],"    ",row[3],"    ",row[4])


18. **except**:
19. myconn.rollback()
21. myconn.close()

**Output:**

ID Name Salary Dept\_Id Dept\_Name

101 John 25000.0 201 CS

102 John 25000.0 201 CS

103 David 25000.0 202 IT

104 Nick 90000.0 201 CS

105 Mike 28000.0 202 IT

# Performing Transactions

Transactions ensure the data consistency of the database. We have to make sure that more than one applications must not modify the records while performing the database operations. The transactions have the following properties.

1. **Atomicity**  
   Either the transaction completes, or nothing happens. If a transaction contains 4 queries then all these queries must be executed, or none of them must be executed.
2. **Consistency**  
   The database must be consistent before the transaction starts and the database must also be consistent after the transaction is completed.
3. **Isolation**  
   Intermediate results of a transaction are not visible outside the current transaction.
4. **Durability**  
   Once a transaction was committed, the effects are persistent, even after a system failure.

## Python commit() method

Python provides the commit() method which ensures the changes made to

the database consistently take place.

The syntax to use the commit() method is given below.

1. conn.commit() #conn is the connection object

All the operations that modify the records of the database do not take place until the commit() is called.

## Python rollback() method

The rollback() method is used to revert the changes that are done to the database. This method is useful in the sense that, if some error occurs during the database operations, we can rollback that transaction to maintain the database consistency.

The syntax to use the rollback() is given below.

1. Conn.rollback()

## Closing the connection

We need to close the database connection once we have done all the operations regarding the database. Python provides the close() method. The syntax to use the close() method is given below.

1. conn.close()

In the following example, we are deleting all the employees who are working for the CS department.

### Example

1. **import** mysql.connector
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
6. #creating the cursor object
7. cur = myconn.cursor()
9. **try**:
10. cur.execute("delete from Employee where Dept\_id = 201")
11. myconn.commit()
12. **print**("Deleted !")
13. **except**:
14. **print**("Can't delete !")
15. myconn.rollback()
17. myconn.close()

**Output:**

Deleted !

Python read csv file

CSV File

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.

Python CSV Module Functions

The CSV module work is used to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

* **csv.field\_size\_limit -** It returns the current maximum field size allowed by the parser.
* **csv.get\_dialect -** It returns the dialect associated with a name.
* **csv.list\_dialects -** It returns the names of all registered dialects.
* **csv.reader -** It read the data from a csv file
* **csv.register\_dialect -** It associates dialect with a name. The name must be a string or a Unicode object.
* **csv.writer -** It writes the data to a csv file
* **o csv.unregister\_dialect -** It deletes the dialect which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
* **csv.QUOTE\_ALL -** It instructs the writer objects to quote all fields. csv.QUOTE\_MINIMAL - It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
* **csv.QUOTE\_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.
* **csv.QUOTE\_NONE -** It instructs the writer object never to quote the fields.

Reading CSV files

Python provides various functions to read csv file. We are describing few method of reading function.

* **Using csv.reader() function**

In Python, the **csv.reader()** module is used to read the csv file. It takes each row of the file and makes a list of all the columns.

We have taken a txt file named as python.txt that have default delimiter **comma(,)** with the following data:

1. name,department,birthday month
2. Parker,Accounting,November
3. Smith,IT,October

**Example**

1. **import** csv
2. with open('python.csv') as csv\_file:
3. csv\_reader = csv.reader(csv\_file, delimiter=',')
4. line\_count = 0
5. **for** row **in** csv\_reader:
6. **if** line\_count == 0:
7. **print**(f'Column names are {", ".join(row)}')
8. line\_count += 1

**Output:**

Column names are name, department, birthday month

Parker works in the Accounting department, and was born in November.

Smith works in the IT department, and was born in October.

Processed 3 lines.

In the above code, we have opened 'python.csv' using the **open()** function. We used **csv.reader()** function to read the file, that returns an iterable reader object. The **reader** object have consisted the data and we iterated using **for** loop to print the content of each row

Read a CSV into a Dictionar

We can also use **DictReader()** function to read the csv file directly into a dictionary rather than deal with a list of individual string elements.

Again, our input file, python.txt is as follows:

1. name,department,birthday month
2. Parker,Accounting,November
3. Smith,IT,October

**Example**

1. **import** csv
2. with open('python.txt', mode='r') as csv\_file:
3. csv\_reader = csv.DictReader(csv\_file)
4. line\_count = 0
5. **for** row **in** csv\_reader:
6. **if** line\_count == 0:
7. **print**(f'The Column names are as follows {", ".join(row)}')
8. line\_count += 1
9. **print**(f'\t{row["name"]} works in the {row["department"]} department, and was born in {row["birthday month"]}.')
10. line\_count += 1
11. **print**(f'Processed {line\_count} lines.')

**Output:**

The Column names are as follows name, department, birthday month

Parker works in the Accounting department, and was born in November.

Smith works in the IT department, and was born in October.

Processed 3 lines.

Reading csv files with Pandas

The Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user.

Reading the csv file into a pandas **DataFrame** is quick and straight forward. We don't need to write enough lines of code to open, analyze, and read the csv file in pandas and it stores the data in **DataFrame**.

Here, we are taking a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

1. Name,Hire Date,Salary,Leaves Remaining
2. John Idle,08/15/14,50000.00,10
3. Smith Gilliam,04/07/15,65000.00,8
4. Parker Chapman,02/21/14,45000.00,10
5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

**Example**

1. **import** pandas
2. df = pandas.read\_csv('hrdata.csv')
3. **print**(df)

In the above code, the three lines are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read\_csv()

**Output:**

Name Hire Date Salary Leaves Remaining

0 John Idle 03/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 8

2 Parker Chapman 05/12/14 45000.0 10

3 Jones Palin 11/01/13 70000.0 3

4 Terry Gilliam 08/12/14 48000.0 7

5 Michael Palin 05/23/13 66000.0 8

[**Next →**](https://www.javatpoint.com/python-read-excel-file)[**← Prev**](https://www.javatpoint.com/python-read-csv-file)

# Python Write CSV File

## CSV File

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 standard format for data interchange. The CSV file opens into the excel sheet, and the rows and columns data define the standard format.

## Python CSV Module Functions

The CSV module work is to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

* **csv.field\_size\_limit -** It returns the current maximum field size allowed by the parser.
* **csv.get\_dialect -** Returns the dialect associated with a name.
* **csv.list\_dialects -** Returns the names of all registered dialects.
* **csv.reader -** Read the data from a CSV file
* **csv.register\_dialect -** It associates dialect with a name, and name must be a string or a Unicode object.
* **csv.writer -** Write the data to a CSV file
* **csv.unregister\_dialect -** It deletes the dialect, which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
* **csv.QUOTE\_ALL -** It instructs the writer objects to quote all fields.
* **csv.QUOTE\_MINIMAL -** It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
* **csv.QUOTE\_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.
* **csv.QUOTE\_NONE -** It instructs the writer object never to quote the fields.

## Writing CSV Files

We can also write any new and existing CSV files in Python by using the csv.writer() module. It is similar to the csv.reader() module and also has two methods, i.e., **writer** function or the **Dict Writer** class.

It presents two functions, i.e., **writerow()** and **writerows()**. The **writerow()** function only write one row, and the **writerows()** function write more than one row.  
**Dialects**

It is defined as a construct that allows you to create, store, and re-use various formatting parameters. It supports several attributes; the most frequently used are:

* **Dialect.delimiter:** This attribute is used as the separating character between the fields. The default value is a comma (,).
* **Dialect.quotechar:** This attribute is used to quote fields that contain special characters.
* **Dialect.lineterminator:** It is used to create new lines, and the default value is '\r\n'.

Let's write the following data to a CSV File.

1. data = [{'Rank': 'B', 'first\_name': 'Parker', 'last\_name': 'Brian'},
2. {'Rank': 'A', 'first\_name': 'Smith', 'last\_name': 'Rodriguez'},
3. {'Rank': 'C', 'first\_name': 'Tom', 'last\_name': 'smith'},
4. {'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Oscar'},
5. {'Rank': 'A', 'first\_name': 'Alex', 'last\_name': 'Tim'}]

### Example -

1. **import** csv
3. with open('Python.csv', 'w') as csvfile:
4. fieldnames = ['first\_name', 'last\_name', 'Rank']
5. writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
7. writer.writeheader()
8. writer.writerow({'Rank': 'B', 'first\_name': 'Parker', 'last\_name': 'Brian'})
9. writer.writerow({'Rank': 'A', 'first\_name': 'Smith',
10. 'last\_name': 'Rodriguez'})
11. writer.writerow({'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Oscar'})
12. writer.writerow({'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Loive'})
14. print("Writing complete")

**Output:**

Writing complete

It returns the file named as 'Python.csv' that contains the following data:

1. first\_name,last\_name,Rank
2. Parker,Brian,B
3. Smith,Rodriguez,A
4. Jane,Oscar,B
5. Jane,Loive,B

## Write a CSV into a Dictionary

We can also use the class **DictWriter** to write the CSV file directly into a dictionary.

A file named as python.csv contains the following data:

Parker, Accounting, November

Smith, IT, October

### Example -

1. **import** csv
2. with open('python.csv', mode='w') as csv\_file:
3. fieldnames = ['emp\_name', 'dept', 'birth\_month']
4. writer = csv.DictWriter(csv\_file, fieldnames=fieldnames)
5. writer.writeheader()
6. writer.writerow({'emp\_name': 'Parker', 'dept': 'Accounting', 'birth\_month': 'November'})
7. writer.writerow({'emp\_name': 'Smith', 'dept': 'IT', 'birth\_month': 'October'})

**Output:**

emp\_name,dept,birth\_month

Parker,Accounting,November

Smith,IT,October

## Writing CSV Files Using Pandas

Pandas is defined as an open source library which is built on the top of Numpy library. It provides fast analysis, data cleaning and preparation of the data for the user.

It is as easy as reading the CSV file using pandas. You need to create the DataFrame, which is a two-dimensional, heterogeneous tabular data structure and consists of three main components- data, columns, and rows. Here, we take a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

1. Name,Hire Date,Salary,Leaves Remaining
2. John Idle,08/15/14,50000.00,10
3. Smith Gilliam,04/07/15,65000.00,8
4. Parker Chapman,02/21/14,45000.00,10
5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

### Example -

1. **import** pandas
2. df = pandas.read\_csv('hrdata.csv',
3. index\_col='Employee',
4. parse\_dates=['Hired'],
5. header=0,
6. names=['Employee', 'Hired', 'Salary', 'Sick Days'])
7. df.to\_csv('hrdata\_modified.csv')

**Output:**

Employee, Hired, Salary, Sick Days

John Idle, 2014-03-15, 50000.0,10

Smith Gilliam, 2015-06-01, 65000.0,8

Parker Chapman, 2014-05-12, 45000.0,10

Jones Palin, 2013-11-01, 70000.0,3

Terry Gilliam, 2014-08-12 , 48000.0,7

Michael Palin, 2013-05-23, 66000.0,8

# Python read excel file

Excel is a spreadsheet application which is developed by Microsoft. It is an easily accessible tool to organize, analyze, and store the data in tables. It is widely used in many different applications all over the world. From Analysts to CEOs, various professionals use Excel for both quick stats and serious data crunching.

## Excel Documents

An Excel spreadsheet document is called a workbook which is saved in a file with **.xlsx** extension. The first row of the spreadsheet is mainly reserved for the header, while the first column identifies the sampling unit. Each workbook can contain multiple sheets that are also called a worksheets. A box at a particular column and row is called a cell, and each cell can include a number or text value. The grid of cells with data forms a sheet.

The active sheet is defined as a sheet in which the user is currently viewing or last viewed before closing Excel.

## Reading from an Excel file

First, you need to write a command to install the **xlrd** module.

# Python Write Excel File

The Python write excel file is used to perform the multiple operations on a spreadsheet using the **xlwt** module. It is an ideal way to write data and format information to files with .xls extension.

If you want to write data to any file and don't want to go through the trouble of doing everything by yourself, then you can use a for loop to automate the whole process a little bit.

## Write Excel File Using xlsxwriter Module

We can also write the excel file using the **xlsxwriter** module. It is defined as a Python module for writing the files in the XLSX file format. It can also be used to write text, numbers, and formulas to multiple worksheets. Also, it supports features such as charts, formatting, images, page setup, auto filters, conditional formatting, and many others.

We need to use the following command to install xlsxwriter module:

1. pip install xlsxwriter

#### Note- Throughout XlsxWriter, rows, and columns are zero-indexed. The first cell in a worksheet is listed as, A1 is (0,0), B1 is (0,1), A2 is (1,0), B2 is (1,1)......,and so on.

## Write Excel File Using openpyxl Module

It is defined as a package which is generally recommended if you want to read and write .xlsx, xlsm, xltx, and xltm files. You can check it by running **type(wb)**.

The load\_workbook() function takes an argument and returns a workbook object, which represents the file. Make sure that you are in the same directory where your spreadsheet is located. Otherwise, you will get an error while importing.

You can easily use a for loop with the help of the range() function to help you to print out the values of the rows that have values in column 2. If those particular cells are empty, you will get None.

## Writing data to Excel files with xlwt

You can use the xlwt package, apart from the XlsxWriter package to create the spreadsheets that contain your data. It is an alternative package for writing data, formatting information, etc. and ideal for writing the data and format information to files with .xls extension. It can perform multiple operations on the spreadsheet.

It supports features such as formatting, images, charts, page setup, auto filters, conditional formatting, and many others.

Pandas have excellent methods for reading all kinds of data from excel files. We can also import the results back to pandas.

## Writing Files with pyexcel

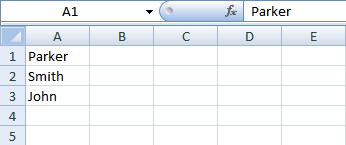
You can easily export your arrays back to a spreadsheet by using the save\_as() function and pass the array and name of the destination file to the dest\_file\_name argument.

It allows us to specify the delimiter and add dest\_delimiter argument. You can pass the symbol that you want to use as a delimiter in-between " ".

**Code**

1. # **import** xlsxwriter module
2. **import** xlsxwriter
4. book = xlsxwriter.Book('Example2.xlsx')
5. sheet = book.add\_sheet()
7. # Rows and columns are zero indexed.
8. row = 0
9. column = 0
11. content = ["Parker", "Smith", "John"]
13. # iterating through the content list
14. **for** item in content :
16. # write operation perform
17. sheet.write(row, column, item)
19. # incrementing the value of row by one with each iterations.
20. row += 1
22. book.close()

**Output:**



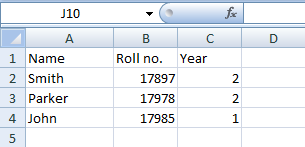
1. pip install xlrd

## Creating a Workbook

A workbook contains all the data in the excel file. You can create a new workbook from scratch, or you can easily create a workbook from the excel file that already exists.

**Input File**

We have taken the snapshot of the workbook.



**Code**

1. # Import the xlrd module
2. **import** xlrd
4. # Define the location of the file
5. loc = ("path of file")
7. # To open the Workbook
8. wb = xlrd.open\_workbook(loc)
9. sheet = wb.sheet\_by\_index(0)
11. # For row 0 and column 0
12. sheet.cell\_value(0, 0)

**Explanation:** In the above example, firstly, we have imported the xlrd module and defined the location of the file. Then we have opened the workbook from the excel file that already exists.

## Reading from the Pandas

Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user and supports both xls and xlsx extensions from the URL.

It is a python package which provides a beneficial data structure called a data frame.

**Example**

1. Example -
2. **import** pandas as pd
4. # Read the file
5. data = pd.read\_csv(".csv", low\_memory=False)
7. # Output the number of rows
8. **print**("Total rows: {0}".format(len(data)))
10. # See which headers are available
11. **print**(list(data))

## Reading from the openpyxl

First, we need to install an openpyxl module using pip from the command line.

1. pip install openpyxl

After that, we need to import the module.

We can also read data from the existing spreadsheet using openpyxl. It also allows the user to perform calculations and add content that was not part of the original dataset.

**Example**

1. **import** openpyxl
2. my\_wb = openpyxl.Workbook()
3. my\_sheet = my\_wb.active
4. my\_sheet\_title = my\_sheet.title
5. **print**("My sheet title: " + my\_sheet\_title)

**Output:**

My sheet title: Sheet

To learn more about openpyxl, visit our complete tutorial [Click Here](https://www.javatpoint.com/python-openpyxl). We have discussed essential detail in this tutorial.

# Python Write Excel File

The Python write excel file is used to perform the multiple operations on a spreadsheet using the **xlwt** module. It is an ideal way to write data and format information to files with .xls extension.

If you want to write data to any file and don't want to go through the trouble of doing everything by yourself, then you can use a for loop to automate the whole process a little bit.

## Write Excel File Using xlsxwriter Module

We can also write the excel file using the **xlsxwriter** module. It is defined as a Python module for writing the files in the XLSX file format. It can also be used to write text, numbers, and formulas to multiple worksheets. Also, it supports features such as charts, formatting, images, page setup, auto filters, conditional formatting, and many others.

We need to use the following command to install xlsxwriter module:

1. pip install xlsxwriter

#### Note- Throughout XlsxWriter, rows, and columns are zero-indexed. The first cell in a worksheet is listed as, A1 is (0,0), B1 is (0,1), A2 is (1,0), B2 is (1,1)......,and so on.

## Write Excel File Using openpyxl Module

It is defined as a package which is generally recommended if you want to read and write .xlsx, xlsm, xltx, and xltm files. You can check it by running **type(wb)**.

The load\_workbook() function takes an argument and returns a workbook object, which represents the file. Make sure that you are in the same directory where your spreadsheet is located. Otherwise, you will get an error while importing.

You can easily use a for loop with the help of the range() function to help you to print out the values of the rows that have values in column 2. If those particular cells are empty, you will get None.

## Writing data to Excel files with xlwt

You can use the xlwt package, apart from the XlsxWriter package to create the spreadsheets that contain your data. It is an alternative package for writing data, formatting information, etc. and ideal for writing the data and format information to files with .xls extension. It can perform multiple operations on the spreadsheet.

It supports features such as formatting, images, charts, page setup, auto filters, conditional formatting, and many others.

Pandas have excellent methods for reading all kinds of data from excel files. We can also import the results back to pandas.

## Writing Files with pyexcel

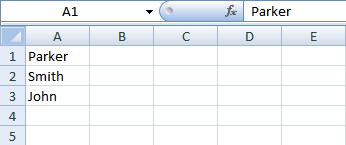
You can easily export your arrays back to a spreadsheet by using the save\_as() function and pass the array and name of the destination file to the dest\_file\_name argument.

It allows us to specify the delimiter and add dest\_delimiter argument. You can pass the symbol that you want to use as a delimiter in-between " ".

**Code**

1. # **import** xlsxwriter module
2. **import** xlsxwriter
4. book = xlsxwriter.Book('Example2.xlsx')
5. sheet = book.add\_sheet()
7. # Rows and columns are zero indexed.
8. row = 0
9. column = 0
11. content = ["Parker", "Smith", "John"]
13. # iterating through the content list
14. **for** item in content :
16. # write operation perform
17. sheet.write(row, column, item)
19. # incrementing the value of row by one with each iterations.
20. row += 1
22. book.close()

**Output:**



# Python Assert Keyword

Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program. For example, while writing a division function, the divisor should not be zero, and you assert that the divisor is not equal to zero.

It is merely a Boolean expression that has a condition or expression checks if the condition returns true or false. If it is true, the program does not do anything, and it moves to the next line of code. But if it is false, it raises an **AssertionError** exception with an optional error message.

The main task of assertions is to inform the developers about unrecoverable errors in the program like "file not found", and it is right to say that assertions are internal self-checks for the program. It is the most essential for the testing or quality assurance in any application development area. The syntax of the assert keyword is given below.

**Syntax**

1. **assert** condition, error\_message(optional)

## Why Assertion is used

It is a debugging tool, and its primary task is to check the condition. If it finds that the condition is true, it moves to the next line of code, and If not, then stops all its operations and throws an error. It points out the error in the code.

## Where Assertion in Python used

* Checking the outputs of the functions.
* Used for testing the code.
* In checking the values of arguments.Checking the valid input.

### Example1

This example shows the working of assert with the error message.

1. **def** avg(scores):
2. **assert** len(scores) != 0,"The List is empty."
3. **return** sum(scores)/len(scores)
5. scores2 = [67,59,86,75,92]
6. **print**("The Average of scores2:",avg(scores2))
8. scores1 = []
9. **print**("The Average of scores1:",avg(scores1))

**Output:**

The Average of scores2: 75.8

AssertionError: The List is empty.

**Explanation:** In the above example, we have passed a non-empty list **scores2** and an empty list **scores1** to the **avg()** function. We received an output for **scores2** list successfully, but after that, we got an error **AssertionError: List is empty**. The assert condition is satisfied by the **scores2** list and lets the program continue to run. However, **scores1** doesn't satisfy the condition and gives an AssertionError.

### Example2:

This example shows the "Divide by 0 error" in the console.

1. # initializing number
2. x = 7
3. y = 0
4. # It uses assert to check for 0
5. **print** ("x / y value is : ")
6. **assert** y != 0, "Divide by 0 error"
7. **print** (x / y)

**Output:**

x / y value is :

### Runtime Exception :

Traceback (most recent call last):

File "main.py", line 6, in <module>

assert y != 0, "Divide by 0 error"

AssertionError: Divide by 0 error

**Explanation:**

In the above example, we have initialized an integer variable, i.e., x=7, y=0, and try to print the value of x/y as an output. The Python interpreter generated a Runtime Exception because of the assert keyword found the divisor as zero then displayed **"Divide by 0 error"** in the console.

Python List Comprehension

Python is known for helping us produce code that is elegant, simple to write, and reads almost as well as plain English. List comprehension is one of the language's most distinguishing features, allowing us to develop sophisticated functionality with just one line of code. On the other hand, many Python writers struggle to fully utilize the more complex aspects of list comprehension. Sometimes programmers may overuse them, resulting in much less efficient and difficult-to-read code.

Using List Comprehension

1. newlist = [expression **for** item **in** iterable **if** condition == True]

Here we are showing basic use of list comprehension.

**Code**

1. #using for loop to iterate through items in list
2. numbers = [3, 5, 1, 7, 3, 9]
3. num = []
5. **for** n **in** numbers:
6. num.append(n\*\*2)
8. **print**(num)

**Output:**

[9, 25, 1, 49, 9, 81]

All of this can be accomplished with only single line of code using list comprehension.

**Code**

1. #using list comprehension to iterate through list items
2. numbers = [3, 5, 1, 7, 3, 9]
4. num = [n\*\*2 **for** n **in** numbers]
6. **print**(num)

**Output:**

[9, 25, 1, 49, 9, 81]

Benefits of Using List Comprehensions

Loops and maps are typically regarded as more Pythonic than list comprehensions. But, rather than taking that judgment at face value, it's worth considering the advantages of utilizing a list comprehension in Python over the alternatives. We'll learn about a couple of cases when the alternatives are preferable options later on.

One of the most important advantages of utilizing a list comprehension in Python is that it is a single tool that can be used in various circumstances. We don't need to adopt a new strategy for each situation. List comprehensions may be leveraged for mapping or filtering and basic list generation.

List comprehensions are regarded as Pythonic, as Python emphasizes simple, effective tools that can be used in many scenarios. As a bonus, we won't have to remember the appropriate order of parameters when using a list comprehension in Python, as we would when calling map().

List comprehensions are easier to read and grasp than loops since they are more declarative. We must concentrate on how exactly the list is constructed while using loops. We must manually build an empty list, then loop over the list's entries, and add each one to the list's end. Instead, using a list comprehension in Python, we can concentrate on what we want to put in the list and allow Python to handle the list generation.

**Code**

1. # Import module to keep track of time
2. **import** time
4. # defining function to execute for loop
5. **def** for\_loop(num):
6. l = []
7. **for** i **in** range(num):
8. l.append(i + 10)
9. **return** l
11. # defining function to execute list comprehension
12. **def** list\_comprehension(num):
13. **return** [i + 10 **for** i **in** range(num)]
15. # Giving values to the functions
17. # Calculating time taken by for loop
18. start = time.time()
19. for\_loop(10000000)
20. end = time.time()
22. **print**('Time taken by for loop:', (end - start))
24. # Calculating time taken by list comprehension
25. start = time.time()
26. list\_comprehension(10000000)
27. end = time.time()
29. **print**('Time taken by list comprehension:', (end - start))

**Output:**

Time taken by for loop: 7.005999803543091

Time taken by list comprehension: 2.822999954223633

Using List Comprehension to Iterate through String

List comprehension can be used in case of strings also as they are iterables too.

**Code**

1. letters = [ alpha **for** alpha **in** 'javatpoint' ]
2. **print**( letters)

**Output:**

['j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't']

Using Conditions in List Comprehension

Conditional statements can be used by list comprehensions to change existing lists (or other tuples). We'll make a list with mathematical operators, numbers, and a range of values.

**Code**

1. number\_list = [ num **for** num **in** range(30) **if** num % 2 != 0]
2. **print**(number\_list)

**Output:**

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

Nested List Comprehensions

Nested List Comprehensions are similar to nested for loops in that they are a list comprehension inside another list comprehension. The programme that implements nested loop is as follows:

**Code**

1. nested\_list = []
3. **for** \_ **in** range(3):
5. # Append an empty sublist inside the list
6. nested\_list.append([])
8. **for** \_\_ **in** range(5):
9. nested\_list[\_].append(\_\_ + \_)
11. **print**(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

The same result may now be created in less lines of code by utilizing layered list comprehensions.

**Code**

1. # Nested list comprehension
2. nested\_list = [[\_ + \_\_ **for** \_ **in** range(5)] **for** \_\_ **in** range(3)]
4. **print**(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

List comprehension is a powerful tool for describing and creating new lists based on existing ones. In general, list comprehension is lighter and easier to use than traditional list construction functions and loops. To provide user-friendly code, we should avoid writing large codes for list comprehensions. Every interpretation of the list or other iterables can be recast in a for loop, but not all the for loops can be rebuilt in the framework of list comprehension.

Python List Comprehension

Python is known for helping us produce code that is elegant, simple to write, and reads almost as well as plain English. List comprehension is one of the language's most distinguishing features, allowing us to develop sophisticated functionality with just one line of code. On the other hand, many Python writers struggle to fully utilize the more complex aspects of list comprehension. Sometimes programmers may overuse them, resulting in much less efficient and difficult-to-read code.

Using List Comprehension

1. newlist = [expression **for** item **in** iterable **if** condition == True]

Here we are showing basic use of list comprehension.

**Code**

1. #using for loop to iterate through items in list
2. numbers = [3, 5, 1, 7, 3, 9]
3. num = []
5. **for** n **in** numbers:
6. num.append(n\*\*2)
8. **print**(num)

**Output:**

[9, 25, 1, 49, 9, 81]

All of this can be accomplished with only single line of code using list comprehension.

**Code**

1. #using list comprehension to iterate through list items
2. numbers = [3, 5, 1, 7, 3, 9]
4. num = [n\*\*2 **for** n **in** numbers]
6. **print**(num)

**Output:**

[9, 25, 1, 49, 9, 81]

Benefits of Using List Comprehensions

Loops and maps are typically regarded as more Pythonic than list comprehensions. But, rather than taking that judgment at face value, it's worth considering the advantages of utilizing a list comprehension in Python over the alternatives. We'll learn about a couple of cases when the alternatives are preferable options later on.

One of the most important advantages of utilizing a list comprehension in Python is that it is a single tool that can be used in various circumstances. We don't need to adopt a new strategy for each situation. List comprehensions may be leveraged for mapping or filtering and basic list generation.

List comprehensions are regarded as Pythonic, as Python emphasizes simple, effective tools that can be used in many scenarios. As a bonus, we won't have to remember the appropriate order of parameters when using a list comprehension in Python, as we would when calling map().

List comprehensions are easier to read and grasp than loops since they are more declarative. We must concentrate on how exactly the list is constructed while using loops. We must manually build an empty list, then loop over the list's entries, and add each one to the list's end. Instead, using a list comprehension in Python, we can concentrate on what we want to put in the list and allow Python to handle the list generation.

**Code**

1. # Import module to keep track of time
2. **import** time
4. # defining function to execute for loop
5. **def** for\_loop(num):
6. l = []
7. **for** i **in** range(num):
8. l.append(i + 10)
9. **return** l
11. # defining function to execute list comprehension
12. **def** list\_comprehension(num):
13. **return** [i + 10 **for** i **in** range(num)]
15. # Giving values to the functions
17. # Calculating time taken by for loop
18. start = time.time()
19. for\_loop(10000000)
20. end = time.time()
22. **print**('Time taken by for loop:', (end - start))
24. # Calculating time taken by list comprehension
25. start = time.time()
26. list\_comprehension(10000000)
27. end = time.time()
29. **print**('Time taken by list comprehension:', (end - start))

**Output:**

Time taken by for loop: 7.005999803543091

Time taken by list comprehension: 2.822999954223633

Using List Comprehension to Iterate through String

List comprehension can be used in case of strings also as they are iterables too.

**Code**

1. letters = [ alpha **for** alpha **in** 'javatpoint' ]
2. **print**( letters)

**Output:**

['j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't']

Using Conditions in List Comprehension

Conditional statements can be used by list comprehensions to change existing lists (or other tuples). We'll make a list with mathematical operators, numbers, and a range of values.

**Code**

1. number\_list = [ num **for** num **in** range(30) **if** num % 2 != 0]
2. **print**(number\_list)

**Output:**

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

Nested List Comprehensions

Nested List Comprehensions are similar to nested for loops in that they are a list comprehension inside another list comprehension. The programme that implements nested loop is as follows:

**Code**

1. nested\_list = []
3. **for** \_ **in** range(3):
5. # Append an empty sublist inside the list
6. nested\_list.append([])
8. **for** \_\_ **in** range(5):
9. nested\_list[\_].append(\_\_ + \_)
11. **print**(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

The same result may now be created in less lines of code by utilizing layered list comprehensions.

**Code**

1. # Nested list comprehension
2. nested\_list = [[\_ + \_\_ **for** \_ **in** range(5)] **for** \_\_ **in** range(3)]
4. **print**(nested\_list)

**Output:**

[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]

List comprehension is a powerful tool for describing and creating new lists based on existing ones. In general, list comprehension is lighter and easier to use than traditional list construction functions and loops. To provide user-friendly code, we should avoid writing large codes for list comprehensions. Every interpretation of the list or other iterables can be recast in a for loop, but not all the for loops can be rebuilt in the framework of list comprehension.

# Python Math Module

Mathematical calculations may occasionally be required when dealing with certain fiscal or rigorous scientific tasks. Python has a math module that can handle these complex calculations. Both simple mathematical calculations like addition (+), and subtraction (-), and advanced mathematical calculations like trigonometric operations, and logarithmic operations can be performed by the functions in the math module.

This tutorial teaches us about applying the math module from fundamentals to more advanced concepts with the support of easy examples to understand the concepts fully. We have included the list of all built-in functions defined in this module for better understanding.

## What is Math Module in Python?

Python has a built-in math module. It is a standard module, so we don't need to install it separately. We only have to import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations.

Since the source code of this module is in the C language, it provides access to the functionalities of the underlying C library. For instance,

**Code**

1. # This program will show the calculation of square root using the math module
2. # importing the math module
3. **import** math
4. **print**(math.sqrt( 9 ))

**Output:**

3.0

This Python module does not accept complex data types. The more complicated equivalent is the cmath module.

We can, for example, calculate all trigonometric ratios for any given angle using the built-in functions in the math module. We must provide angles in radians to these trigonometric functions (sin, cos, tan, etc.). However, we are accustomed to measuring angles in terms of degrees. The math module provides two methods to convert angles from radians to degrees and vice versa.

## Constants in Math Module

The value of numerous constants, including pi and tau, is provided in the math module so that we do not have to remember them. Using these constants eliminates the need to precisely and repeatedly write down the value of each constant. The math module includes the following constants:

1. Euler's Number
2. Tau
3. Infinity
4. Pi
5. Not a Number (NaN)

Let's go over each of them one by one.

### Euler's Number

The value 2.71828182845 of Euler's number is returned by the math.e constant.

**Syntax of this is:**

1. math.e

**Code**

1. # importing the required library
2. **import** math
4. # printing the value of Euler's number using the math module
5. **print**( "The value of Euler's Number is: ", math.e )

**Output:**

The value of Euler's Number is: 2.718281828459045

### Tau

The ratio of a circle's circumference to its radius is known as tau. The value tau returned by the tau constant is 6.283185307179586.

**Syntax of this is:**

1. math.tau

**Code**

1. # Importing the required library
2. **import** math
4. # Printing the value of tau using math module
5. **print** ( "The value of Tau is: ", math.tau )

**Output:**

The value of Tau is: 6.283185307179586

### Infinity

Infinity refers to anything limitless or never-ending in both directions of the actual number line. Numbers cannot adequately represent it. The math.inf returns positive infinity constant. We can use -math.inf to print negative infinity.

**Syntax of this is:**

1. math.inf

**Code**

1. # Importing the required library
2. **import** math
4. # Printing the value of positive infinity using the math module
5. **print**( math.inf )
7. # Printing the value of negative infinity using the math module
8. **print**( -math.inf )

**Output:**

inf

-inf

Further, we are comparing a very large floating-point number with positive and negative infinity values.

**Code**

1. # Importing the required library
2. **import** math
4. # comparing the value of infinity
5. **print**( math.inf > 10e109 )
6. **print**( -math.inf < -10e109 )

**Output:**

True

True

### Pi

Pi is known to everyone. It is mathematically represented as either the fraction 22/7 or the decimal number 3.14. math.pi gives the most accurate value of pi.

**Syntax of this is:**

1. math.pi

**Code**

1. # Importing the required library
2. **import** math
4. # Printing the value of pi using the math module
5. **print**( "The value of pi is ", math.pi )

**Output:**

The value of pi is 3.141592653589793

Let us calculate the circumference of a circle.

**Code**

1. # Importing the required library
2. **import** math
4. # radius of the circle
5. r = 4
7. # value of pi
8. pi\_value = math.pi
10. # circumference of the circle
11. **print**(2 \* pi\_value \* r)

**Code**

25.132741228718345

### NaN

The math.nan gives us a floating-point nan (Not a Number) value. This amount is not a valid numeric value. Float("nan") and the nan constant are comparable.

**Code**

1. # Importing the required library
2. **import** math
4. # Printing the value of nan using the math module
5. **print**( math.nan )

**Output:**

nan

## Mathematical Operations with Math Module

The functions that are required in representation theory and number theory, such as calculating the factorial of an integer, will be covered in this part.

### Calculating the Ceiling and the Floor Value

The terms "ceiling value" and "floor value" refer to the smallest integral value larger than the number and the largest integral value less than the number, respectively. The ceil() and floor() methods simplify calculating this.

**Code**

1. # Python program to show how to use floor() and ceil() functions.
3. # importing the math module
4. **import** math
6. x = 4.346
8. # returning the ceiling value of 4.346
9. **print**("The ceiling value of 4.346 is : ", end="")
10. **print**( math.ceil(x) )
12. # returning the floor value of 4.346
13. **print**("The floor value of 4.346 is : ", end="")
14. **print**( math.floor(x) )

**Output:**

The ceiling value of 4.346 is : 5

The floor value of 4.346 is : 4

### Calculating the Factorial of the Number

We may determine the factorial of a given integer in a one-liner code by using the math.factorial() function. The Python interpreter will send a message if the given number is not integral.

**Code**

1. # Python program to show how to use function() functions.
3. # importing the math module
4. **import** math
6. x = 6
8. # returning the factorial of 6
9. **print**( "The factorial of 6 is : ", math.factorial(x) )
11. # passing a non integral number
12. **try**:
13. **print**( "The factorial of 6.5 in: ", math.factorial(6.5) )
14. **except**:
15. **print**( "Cannot calculate factorial of a non-integral number" )

**Output:**

The factorial of 6 is : 720

Cannot calculate factorial of a non-integral number

### Calculating the Absolute Value

The method math.fabs() returns the absolute number of the number given to the function.

**Code**

1. # Python program to show how to use fabs() functions.
3. # importing the math module
4. **import** math
6. x = -45
8. # returning x's absolute value.
9. **print**( "The absolute value of -45 is: ", math.fabs(x) )

**Output:**

The absolute value of -45 is: 45.0

### Calculating the Exponential

x to the power of e, often known as the exponential of a number x, is calculated using the exp() function.

**Code**

1. # Python program to show how to use the exp() function.
3. # importing the math module
4. **import** math
6. # declaring some value
7. num1 = 4
8. num2 = -3
9. num3 = 0.00
11. # passing above values to the exp() function
12. **print**( f"The exponenetial value of {num1} is: ", math.exp(num1) )
13. **print**( f"The exponenetial value of {num2} is: ", math.exp(num2) )
14. **print**( f"The exponenetial value of {num3} is: ", math.exp(num3) )

**Output:**

The exponenetial value of 4 is: 54.598150033144236

The exponenetial value of -3 is: 0.049787068367863944

The exponenetial value of 0.0 is: 1.0

### Calculating the Power of a Number

x\*\*y is computed via the pow() function. This function calculates the value of the power after converting its inputs into floats.

**Code**

1. # Python program to show how to use the pow() function.
3. # importing the math module
4. **import** math
6. x = 4
7. y = 5
8. # returning x to the power of y.
9. **print**( f"The value of {x} to the power of {y} is: ", math.pow(x,y) )

**Output:**

The value of 4 to the power of 5 is: 1024.0

### Calculating Sine, Cosine, and Tangent

The values of sine, cosine, and tangent of an angle, which are supplied as an input to the function, are returned by the sin(), cos(), and tan() methods. This function expects a value that is provided in radians.

**Code**

1. # Python program to show how to use the sin(), cos(), tan() function.
3. # importing the math module
4. **import** math
6. angle = math.pi / 4
8. # returning the sine of pi/4
9. **print**( "The sine of pi/4 is : ", math.sin( angle ) )
11. # returning the cosine of pi/4
12. **print**( "The cosine of pi/4 is : ", math.cos( angle ) )
14. # returning the tangent of pi/4
15. **print**("The tangent of pi/4 is : ", math.tan( angle ))

**Output:**

The sine of pi/4 is : 0.7071067811865475

The cosine of pi/4 is : 0.7071067811865476

The tangent of pi/4 is : 0.9999999999999999

## The dir( ) Function

A sorted list of strings comprising the identifiers of the functions defined by a module is what the built-in method dir() delivers.

The list includes the names of modules, each specified constants, functions, and methods. Here is a straightforward illustration:

**Code**

1. # Importing the math module
2. **import** math
4. functions = dir(math)
5. **print**( functions )

**Output:**

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']

## Description of all the Functions in Python Math Module

Here is a list of all the properties and functions specified in the math module, along with a brief description of what each one does.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **ceil(x)** | The lowest integer bigger than or equal to x is returned. |
| **copysign(x, y)** | gives x back with the sign of y. |
| **fabs(x)** | gives x's absolute value back. |
| **factorial(x)** | provides the x factorial back. |
| **floor(x)** | gives back the biggest integer that is less than or equal to x. |
| **fmod(x, y)** | returns the leftover value after dividing x by y. |
| **frexp(x)** | returns the pair of the mantissa and exponent of x. (m, e) |
| **fsum(iterable)** | returns the iterable's correct floating point sum of all values. |
| **isfinite(x)** | If x is neither an infinity nor a NaN, it returns True (Not a Number) |
| **isinf(x)** | If x is a positive or negative infinity, it returns True. |
| **isnan(x)** | If x is a NaN, it returns True. |
| **ldexp(x, i)** | gives back x \* (2\*\*i). |
| **modf(x)** | gives x's fractional and integer components back. |
| **trunc(x)** | x's shortened integer value is returned. |
| **exp(x)** | delivers e\*\*x |
| **expm1(x)** | yields e\*\*x - 1 |
| **log(x[, b])** | gives back the x logarithm in base b. (defaults to e) |
| **log1p(x)** | the natural logarithm of 1 + x is returned. |
| **log2(x)** | gives x's base-2 logarithm back. |
| **log10(x)** | provides x's base-10 logarithm. |
| **pow(x, y)** | gives x raised to the power of y back. |
| **sqrt(x)** | gives x's square root back. |
| **acos(x)** | gives the arc cosine of x back. |
| **asin(x)** | gives the arc sine of x back. |
| **atan(x)** | gives the arc tangent of x back. |
| **atan2(y, x)** | gives back atan(y / x). |
| **cos(x)** | returns the x's cosine. |
| **hypot(x, y)** | returns sqrt(x\*x + y\*y), the Euclidean norm. |
| **sin(x)** | gives the sine of x back. |
| **tan(x)** | gives the tangent of x back. |
| **degrees(x)** | Angle x is transformed from radians to degrees. |
| **radians(x)** | Angle x is transformed from degrees to radians. |
| **acosh(x)** | x's inverse hyperbolic cosine is returned. |
| **asinh(x)** | x's inverse hyperbolic sine is returned. |
| **atanh(x)** | x's inverse hyperbolic tangent is returned. |
| **cosh(x)** | gives x's hyperbolic cosine. |
| **sinh(x)** | gives x's hyperbolic cosine. |
| **tanh(x)** | gives x's hyperbolic tangent back. |
| **erf(x)** | the error function at x is returned. |
| **erfc(x)** | a function that gives the complementary error at x |
| **gamma(x)** | the Gamma function at x is returned. |
| **lgamma(x)** | gives the natural logarithm of the gamma function's absolute value at x. |
| **pi** | The ratio of a circle's circumference to its diameter is a mathematical constant (3.14159...) |
| **e** | e is a constant in mathematics (2.71828...) |

Python OS Module

Python OS module provides the facility to establish the interaction between the user and the operating system. It offers many useful OS functions that are used to perform OS-based tasks and get related information about operating system.

The OS comes under Python's standard utility modules. This module offers a portable way of using operating system dependent functionality.

The Python OS module lets us work with the files and directories.

1. To work with the OS module, we need to **import** the OS module.
2. **import** os

There are some functions in the OS module which are given below:

os.name()

This function provides the name of the operating system module that it imports.

Currently, it registers 'posix', 'nt', 'os2', 'ce', 'java' and 'riscos'.

**Example**

1. **import** os
2. **print**(os.name)

**Output:**

nt

os.mkdir()

The **os.mkdir()** function is used to create new directory. Consider the following example.

1. **import** os
2. os.mkdir("d:\\newdir")

It will create the new directory to the path in the string argument of the function in the D drive named folder newdir.

os.getcwd()

It returns the current working directory(CWD) of the file.

**Example**

1. **import** os
2. **print**(os.getcwd())

**Output:**

C:\Users\Python\Desktop\ModuleOS

os.chdir()

The **os** module provides the **chdir()** function to change the current working directory.

1. **import** os
2. os.chdir("d:\\")

**Output:**

d:\\

os.rmdir()

The **rmdir()** function removes the specified directory with an absolute or related path. First, we have to change the current working directory and remove the folder.

**Example**

1. **import** os
2. # It will throw a Permission error; that's why we have to change the current working directory.
3. os.rmdir("d:\\newdir")
4. os.chdir("..")
5. os.rmdir("newdir")

os.error()

The os.error() function defines the OS level errors. It raises OSError in case of invalid or inaccessible file names and path etc.

**Example**

1. **import** os
3. **try**:
4. # If file does not exist,
5. # then it throw an IOError
6. filename = 'Python.txt'
7. f = open(filename, 'rU')
8. text = f.read()
9. f.close()
11. # The Control jumps directly to here if
12. # any lines throws IOError.
13. **except** IOError:
15. # print(os.error) will <class 'OSError'>
16. **print**('Problem reading: ' + filename)

**Output:**

Problem reading: Python.txt

os.popen()

This function opens a file or from the command specified, and it returns a file object which is connected to a pipe.

**Example**

1. **import** os
2. fd = "python.txt"
4. # popen() is similar to open()
5. file = open(fd, 'w')
6. file.write("This is awesome")
7. file.close()
8. file = open(fd, 'r')
9. text = file.read()
10. **print**(text)
12. # popen() provides gateway and accesses the file directly
13. file = os.popen(fd, 'w')
14. file.write("This is awesome")
15. # File not closed, shown in next function.

**Output:**

This is awesome

os.close()

This function closes the associated file with descriptor **fr**.

**Example**

1. **import** os
2. fr = "Python1.txt"
3. file = open(fr, 'r')
4. text = file.read()
5. **print**(text)
6. os.close(file)

**Output:**

Traceback (most recent call last):

File "main.py", line 3, in

file = open(fr, 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'Python1.txt'

os.rename()

A file or directory can be renamed by using the function **os.rename()**. A user can rename the file if it has privilege to change the file.

**Example**

1. **import** os
2. fd = "python.txt"
3. os.rename(fd,'Python1.txt')
4. os.rename(fd,'Python1.txt')

**Output:**

Traceback (most recent call last):

File "main.py", line 3, in

os.rename(fd,'Python1.txt')

FileNotFoundError: [Errno 2] No such file or directory: 'python.txt' -> 'Python1.txt'

os.access()

This function uses real **uid/gid** to test if the invoking user has access to the path.

**Example**

1. **import** os
2. **import** sys
4. path1 = os.access("Python.txt", os.F\_OK)
5. **print**("Exist path:", path1)
7. # Checking access with os.R\_OK
8. path2 = os.access("Python.txt", os.R\_OK)
9. **print**("It access to read the file:", path2)
11. # Checking access with os.W\_OK
12. path3 = os.access("Python.txt", os.W\_OK)
13. **print**("It access to write the file:", path3)
15. # Checking access with os.X\_OK
16. path4 = os.access("Python.txt", os.X\_OK)
17. **print**("Check if path can be executed:", path4)

**Output:**

Exist path: False

It access to read the file: False

It access to write the file: False

Check if path can be executed: False

Python Random module

The Python Random module is a built-in module for generating random integers in Python. These are sort of fake random numbers which do not possess true randomness. We can therefore use this module to generate random numbers, display a random item for a list or string, and so on.

Generate Random Floats

The random.random() function gives a float number that ranges from 0.0 to 1.0. There are no parameters required for this function.

**random.random():-** Returns The second random floating point value within [0.0 and 1) is returned.

**random.uniform(a, b):-** Generates a random floating point R in which a <= R <= b if a <= b and b <= R <= a if b < a.

**random.expovariate(lambda):-** Returns the random value according to exponential distribution.

**random.gauss(mu, sigma):-** Returns the random value according to gaussian distribution.

There are other distributions also, such as Gamma Distribution, Normal Distribution, etc.

**Code**

1. **import** random
2. num=random.random()
3. **print**(num)

**Output:**

0.3232640977876686

Generate Random Integers

The random.randint() function generates a random integer from the range of numbers supplied.

**Code**

1. **import** random
2. num = random.randint(1, 500)
3. **print**( num )

**Output:**

215

Generate Random Numbers within a Defined Range

The random.randrange() function selects an item randomly from the given range defined by the start, the stop, and the step parameters. By default, the start is set to 0. Likewise, the step is set to 1 by default.

**Code**

1. **import** random
3. num = random.randrange(1, 10)
4. **print**( num )
5. num = random.randrange(1, 10, 2)
6. **print**( num )
7. num = random.randrange(0, 101, 10)
8. **print**( num )

**Output:**

4

9

20

Select Random Elements

The random.choice() function selects an item from a non-empty series at random. An IndexError is thrown when the parameter is an empty series.

**Code**

1. **import** random
2. random\_s = random.choice('Random Module') #a string
3. **print**( random\_s )
4. random\_l = random.choice([23, 54, 765, 23, 45, 45]) #a list
5. **print**( random\_l )
6. random\_s = random.choice((12, 64, 23, 54, 34)) #a set
7. **print**( random\_s )

**Output:**

M

765

54

Shuffle Elements Randomly

A general sequence, like integers or floating-point series, can be a group of things like a List / Set. The random module contains methods that we can use to add randomization to the series.

The random.shuffle() function shuffles the entries in a list at random.

**Code**

1. a\_list = [34, 23, 65, 86, 23, 43]
2. random.shuffle( a\_list )
3. **print**( a\_list )
4. random.shuffle( a\_list )
5. **print**( a\_list )

**Output:**

[23, 43, 86, 65, 34, 23]

[65, 23, 86, 23, 34, 43]

Random Seed

We normally use the time of the system to ensure that the software delivers a different output each time we execute it because pseudorandom synthesis is dependent on the preceding number. As a result, we employ seeds.

We can specify a seed to have an initial number using Python's random.seed() function. This seed number determines a random number generator's outcome; therefore, if it stays the same, the outcome will continue to be the same.

**Code**

1. **import** random
2. random.seed(2)
3. **print**('Generating 5 random numbers: ')
4. **print**([ random.randint(1, 300) **for** r **in** range(6)])
6. # Reseting the seed value to 1
7. random.seed(2)
9. # We will get the same numbers as before
10. **print**([random.randint(1, 300) **for** i **in** range(6)])

**Output:**

Generating 5 random numbers:

[29, 47, 44, 185, 87, 158]

[29, 47, 44, 185, 87, 158]

Various Functions of Random Module

Following is the list of functions available in the random module.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **seed(a=None, version=2)** | This function creates a new random number. |
| **getstate()** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **setstate(state)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **getrandbits(k)** | This function provides a Python integer having k random bits. This is important for random number production algorithms like randrange(), which can manage arbitrarily huge ranges. |
| **randrange(start, stop[, step])** | From the range, it produces a random integer. |
| **randint(a, b)** | Provides an integer within a and b at random (both inclusive). If a > b, a ValueError is thrown. |
| **choice(seq)** | Produce a non-empty series item at random. |
| **shuffle(seq)** | Change the order. |
| **sample(population, k)** | Display a list of k-size unique entries from the population series. |
| **random()** | This function creates a new random number. |
| **uniform(a, b)** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **triangular(low, high, mode)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **betavariate(alpha, beta)** | Beta distribution |
| **expovariate(lambd)** | Exponential distribution |
| **gammavariate(alpha, beta)** | Gamma distribution |
| **gauss(mu, sigma)** | Gaussian distribution |
| **lognormvariate(mu, sigma)** | Log normal distribution |
| **normalvariate(mu, sigma)** | Normal distribution |
| **vonmisesvariate(mu, kappa)** | Vonmises distribution |
| **paretovariate(alpha)** | Pareto distribution |
| **weibullvariate(alpha, beta)** | Weibull distribution |

We learned about various methods that Python's random module provides us with for dealing with Integers, floating-point numbers, and other sequences like Lists, tuples, etc. We also looked at how the seed affects the pseudo - random number pattern.

Python statistics module

Python statistics module provides the functions to mathematical statistics of numeric data. There are some popular statistical functions defined in this module.

mean() function

The mean() function is used to calculate the arithmetic mean of the numbers in the list.

**Example**

1. **import** statistics
2. # list of positive integer numbers
3. datasets = [5, 2, 7, 4, 2, 6, 8]
4. x = statistics.mean(datasets)
5. # Printing the mean
6. **print**("Mean is :", x)

**Output:**

Mean is : 4.857142857142857

median() function

The median() function is used to return the middle value of the numeric data in the list.

**Example**

1. **import** statistics
2. datasets = [4, -5, 6, 6, 9, 4, 5, -2]
3. # Printing median of the
4. # random data-set
5. **print**("Median of data-set is : % s "
6. % (statistics.median(datasets)))

**Output:**

Median of data-set is : 4.5

mode() function

The mode() function returns the most common data that occurs in the list.

**Example**

1. **import** statistics
2. # declaring a simple data-set consisting of real valued positive integers.
3. dataset =[2, 4, 7, 7, 2, 2, 3, 6, 6, 8]
4. # Printing out the mode of given data-set
5. **print**("Calculated Mode % s" % (statistics.mode(dataset)))

**Output:**

Calculated Mode 2

stdev() function

The stdev() function is used to calculate the standard deviation on a given sample which is available in the form of the list.

**Example**

1. **import** statistics
2. # creating a simple data - set
3. sample = [7, 8, 9, 10, 11]
4. # Prints standard deviation
5. **print**("Standard Deviation of sample is % s "
6. % (statistics.stdev(sample)))

**Output:**

Standard Deviation of sample is 1.5811388300841898

median\_low()

The median\_low function is used to return the low median of numeric data in the list.

**Example**

1. **import** statistics
2. # simple list of a set of integers
3. set1 = [4, 6, 2, 5, 7, 7]
4. # Note: low median will always be a member of the data-set.
5. # Print low median of the data-set
6. **print**("Low median of data-set is % s "
7. % (statistics.median\_low(set1)))

**Output:**

Low median of the data-set is 5

median\_high()

The median\_high function is used to return the high median of numeric data in the list.

**Example**

1. **import** statistics
2. # list of set of the integers
3. dataset = [2, 1, 7, 6, 1, 9]
4. **print**("High median of data-set is %s "
5. % (statistics.median\_high(dataset)))

**Output:**

High median of the data-set is 6

Python sys module

The python sys module provides functions and variables which are used to manipulate different parts of the Python Runtime Environment. It lets us access system-specific parameters and functions.

**import sys**

First, we have to import the sys module in our program before running any functions.

**sys.modules**

This function provides the name of the existing python modules which have been imported.

**sys.argv**

This function returns a list of command line arguments passed to a Python script. The name of the script is always the item at index 0, and the rest of the arguments are stored at subsequent indices.

**sys.base\_exec\_prefix**

This function provides an efficient way to the same value as exec\_prefix. If not running a virtual environment, the value will remain the same.

**sys.base\_prefix**

It is set up during Python startup, before site.py is run, to the same value as prefix.

**sys.byteorder**

It is an indication of the native byteorder that provides an efficient way to do something.

**sys.maxsize**

This function returns the largest integer of a variable.

**sys.path**

This function shows the PYTHONPATH set in the current system. It is an environment variable that is a search path for all the python modules.

**sys.stdin**

It is an object that contains the original values of stdin at the start of the program and used during finalization. It can restore the files.

**sys.getrefcount**

This function returns the reference count of an object.

**sys.exit**

This function is used to exit from either the Python console or command prompt, and also used to exit from the program in case of an exception.

**sys executable**

The value of this function is the absolute path to a Python interpreter. It is useful for knowing where python is installed on someone else machine.

**sys.platform**

This value of this function is used to identify the platform on which we are working.

Python Arrays

An array is defined as a collection of items that are stored at contiguous memory locations. It is a container which can hold a fixed number of items, and these items should be of the same type. An array is popular in most programming languages like C/C++, JavaScript, etc.

Array is an idea of storing multiple items of the same type together and it makes easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in single variable. If you have a list of items that are stored in their corresponding variables like this:

car1 = "Lamborghini"

car2 = "Bugatti"

car3 = "Koenigsegg"

If you want to loop through cars and find a specific one, you can use the array.

The array can be handled in Python by a module named **array**. It is useful when we have to manipulate only specific data values. Following are the terms to understand the concept of an array:

**Element** - Each item stored in an array is called an element.

**Index** - The location of an element in an array has a numerical index, which is used to identify the position of the element.

Array Representation

An array can be declared in various ways and different languages. The important points that should be considered are as follows:

* Index starts with 0.
* We can access each element via its index.
* The length of the array defines the capacity to store the elements.

Array operations

Some of the basic operations supported by an array are as follows:

* **Traverse** - It prints all the elements one by one.
* **Insertion** - It adds an element at the given index.
* **Deletion** - It deletes an element at the given index.
* **Search** - It searches an element using the given index or by the value.
* **Update** - It updates an element at the given index.

The Array can be created in Python by importing the array module to the python program.

1. from array **import** \*
2. arrayName = array(typecode, [initializers])

**Accessing array elements**

We can access the array elements using the respective indices of those elements.

1. **import** array as arr
2. a = arr.array('i', [2, 4, 6, 8])
3. **print**("First element:", a[0])
4. **print**("Second element:", a[1])
5. **print**("Second last element:", a[-1])

**Output:**

First element: 2

Second element: 4

Second last element: 8

**Explanation:** In the above example, we have imported an array, defined a variable named as "a" that holds the elements of an array and print the elements by accessing elements through indices of an array.

How to change or add elements

Arrays are mutable, and their elements can be changed in a similar way like lists.

1. **import** array as arr
2. numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
4. # changing first element
5. numbers[0] = 0
6. print(numbers)    # Output: array('i', [0, 2, 3, 5, 7, 10])
8. # changing 3rd to 5th element
9. numbers[2:5] = arr.array('i', [4, 6, 8])
10. print(numbers)    # Output: array('i', [0, 2, 4, 6, 8, 10])

**Output:**

array('i', [0, 2, 3, 5, 7, 10])

array('i' ,[0, 2, 4, 6, 8, 10])

**Explanation:** In the above example, we have imported an array and defined a variable named as "numbers" which holds the value of an array. If we want to change or add the elements in an array, we can do it by defining the particular index of an array where you want to change or add the elements.

Why to use arrays in Python?

A combination of arrays saves a lot of time. The array can reduce the overall size of the code.

How to delete elements from an array?

The elements can be deleted from an array using Python's **del** statement. If we want to delete any value from the array, we can do that by using the indices of a particular element.

1. **import** array as arr
2. number = arr.array('i', [1, 2, 3, 3, 4])
3. del number[2]                           # removing third element
4. print(number)                           # Output: array('i', [1, 2, 3, 4])

**Output:**

array('i', [10, 20, 40, 60])

**Explanation:** In the above example, we have imported an array and defined a variable named as "number" which stores the values of an array. Here, by using del statement, we are removing the third element [3] of the given array.

Finding the length of an array

The length of an array is defined as the number of elements present in an array. It returns an integer value that is equal to the total number of the elements present in that array.

**Syntax**

1. len(array\_name)

Array Concatenation

We can easily concatenate any two arrays using the + symbol.

**Example**

1. a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
2. b=arr.array('d',[3.7,8.6])
3. c=arr.array('d')
4. c=a+b
5. print("Array c = ",c)

**Output:**

Array c= array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])

**Explanation**

In the above example, we have defined variables named as "a, b, c" that hold the values of an array.

**Example**

1. **import** array as arr
2. x = arr.array('i', [4, 7, 19, 22])
3. print("First element:", x[0])
4. print("Second element:", x[1])
5. print("Second last element:", x[-1])

**Output:**

First element: 4

Second element: 7

Second last element: 22

**Explanation:** In the above example, first, we have imported an array and defined a variable named as "x" which holds the value of an array and then, we have printed the elements using the indices of an array.

# Python Command line arguments

The Python supports the programs that can be run on the command line, complete with command line arguments. It is the input parameter that needs to be passed to the script when executing them.

It means to interact with a command-line interface for the scripts.

It provides a **getopt** module, in which command line arguments and options can be parsed.

## What is argument passing?

The command **ls** is often used to get a summary of files and folders present in a particular directory.

## Why to use argparse?

It means to communicate between the writer of a program and user which does not require going into the code and making changes to the script. It provides the ability to a user to enter into the command-line arguments.

## Access command line arguments

The Python sys module provides access to command-line arguments via sys.argv. It solves the two purposes:

## Python sys module

It is a basic module that was shipped with Python distribution from the early days on. It is a similar approach as C library using argc/argv to access the arguments. The sys module implements command-line arguments in a simple list structure named sys.argv.

Each list element represents a single argument. The first one -- sys.argv[0] -- is the name of Python script. The other list elements are sys.argv[1] to sys.argv[n]- are the command line arguments 2 to n. As a delimiter between arguments, space is used. Argument values that contain space in it have to be quoted, accordingly.

It stores command-line arguments into a list; we can access it using **sys.argv**. This is very useful and a simple way to read command-line arguments as String.

1. **import** sys
2. print(type(sys.argv))
3. print('The command line arguments are:')
4. **for** i in sys.argv:
5. print(i)

## Python getopt module

The Python getopt module extends the separation of the input string by parameter validation. Based on getopt C function, it allows both short and long options, including a value assignment.

It is very similar to C getopt() function for parsing command line parameters.

It is useful in parsing command line arguments where we want the user to enter some options.

**Code**

1. **import** getopt
2. **import** sys
3. argv = sys.argv[1:]
4. **try**:
5. opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my\_file='])
6. print(opts)
7. print(args)
8. except getopt.GetoptError:
9. # Print a message or **do** something useful
10. print('Something went wrong!')
11. sys.exit(2)

## Python argparse module

It offers a command-line interface with standardized output, whereas the former two solutions leave most of the work in your hands. argparse allows verification of fixed and optional arguments with a name checking as either UNIX or GNU style. It is the preferred way to parse command-line arguments. It provides a lot of option such as positional arguments, the default value for arguments, helps message, specifying the data type of argument etc.

It makes it easy to write the user-friendly command-line interfaces. It automatically generates help and usage messages and issues errors when a user gives invalid arguments to the program.

**getopt.getopt method**

This method is used for parsing the command line options and parameter list.

**Syntax:**

1. getopt.getopt(args, options, [long\_options])

**args**- It is an argument list that needs to be parsed.

**options**- A string of option letters that the script wants to recognize, with options that require an argument which should be followed by a colon(:).

**long\_options(optional)**- It must be a string with names of the long options, which should be supported.

* This method returns a value consisting of two elements, i.e. list of (**option, value**) pairs, list of program arguments left after option list was stripped.
* Each option-and-value pair are returned as an option as its first element, prefixed with a hyphen for short options (e.g.,'-x') or two hyphens for long options (e.g., '--long-option').

**Exception getopt.GetoptError**

This exception arises when an unrecognized option is found in the argument list or when any option requiring an argument is given none.

The argument to the exception is a string that indicates the cause of the error. The attributes **msg** and **opt** to give the error message and related option.

**Code**

1. #!/usr/bin/python
2. **import** sys, getopt
3. def main(argv):
4. inputfile = ''
5. outputfile = ''
6. **try**:
7. opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
8. except getopt.GetoptError:
9. print 'test.py -i <inputfile> -o <outputfile>'
10. sys.exit(2)
11. **for** opt, arg in opts:
12. **if** opt == '-h':
13. print 'test.py -i <inputfile> -o <outputfile>'
14. sys.exit()
15. elif opt in ("-i", "--ifile"):
16. inputfile = arg
17. elif opt in ("-o", "--ofile"):
18. outputfile = arg
19. print 'Input file is "', inputfile
20. print 'Output file is "', outputfile
22. **if** \_\_name\_\_ == "\_\_main\_\_":
23. main(sys.argv[1:])

**Output:**

$ test.py -h

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i BMP -o

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i inputfile

Input file is " inputfile

Output file is "

## How to use command line arguments in python?

<="" p="" style="color: rgb(51, 51, 51); font-family: inter-regular, system-ui, -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Helvetica, Arial, sans-serif; font-size: 16px; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; font-weight: 400; letter-spacing: normal; orphans: 2; text-align: justify; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: rgb(255, 255, 255); text-decoration-thickness: initial; text-decoration-style: initial; text-decoration-color: initial;">

|  |  |  |
| --- | --- | --- |
| **Module** | **Use** | **Python version** |
| sys | All arguments in sys.argv (basic) | All |
| argparse | Build a command line interface | >= 2.3 |
| docopt | Created command line interfaces | >= 2.5 |
| fire | Automatically generate command line interfaces (CLIs) | All |
| optparse | Deprecated | < 2.7 |

**Docopt**

Docopt is used to create command line interfaces.

from docopt import docopt

if \_\_name\_\_ == '\_\_main\_\_':

arguments = docopt(\_\_doc\_\_, version='Example 1')

print(arguments)

**Fire**

Python Fire automatically generates a command line interface; you only need one line of code. Unlike the other modules, it works instantly.

You don't need to define any arguments; all the methods are linked by default.

To install it type:

pip install fire

Define or use a class:

1. **import** fire
2. **class** Python(object):
3. def hello(self):
4. print("Hello")
5. def openfile(self, filename):
6. print("Open file '" + filename + "'")
8. **if** \_\_name\_\_ == '\_\_main\_\_':
9. fire.Fire(Python)

You have the options matching to the class methods:

1. python example.py hello
2. python example.py openfile filename.txt

Python Magic Methods

To add "magic" to the class we create, we can define special methods called "magic methods." For example, the magic methods \_\_init\_\_ and \_\_str\_\_are always wrapped by double underscores from both sides. By granting us accessibility to Python's built-in syntax tools, magic methods can improve the structure of our classes.

We can integrate Python's built-in classes with our classes. The class which has inherited from the built-in class is known as a child class. A child class has access to all of the attributes of the parent class, including its methods. By utilizing the essential built-in features, we can customize some of the tasks of our class by using magic methods.

\_\_init\_\_ Method

After we have constructed an instance of the class, but before that instance is returned to the caller of the class, the \_init\_ method is executed. When we create an instance of the class, it is called automatically, just like constructors in various programming languages like the popular ones C++, Java, C#, PHP, etc. These methods are invoked after \_new\_ and therefore are referred to as initialising. We should define the instance parameters here.

**Code**

1. # Python program to show how \_\_init\_\_ method works
3. # Creating a class
4. **class** methods():
5. **def** \_\_init\_\_(self, \*args):
6. **print** ("Now called \_\_init\_\_ magic method, after tha initialised parameters")
7. self.name = args[0]
8. self.std = args[1]
9. self.marks = args[2]
11. Student = methods("Itika", 11, 98)
12. **print**(Student)
13. **print**(f"Name, standard, and marks of the student is: \n", Student.name, "\n", Student.std, "\n", Student.marks)

**Output:**

Now called \_\_init\_\_ magic method, after tha initialised parameters

<\_\_main\_\_.methods object at 0x3701290>

Name, standard, and marks of the student is:

Itika

11

98

\_\_new\_\_() Method

The magic method \_\_new\_\_() is called implicitly by the \_\_init\_\_() method. The new instance returned by the \_\_new\_\_() method is initialised. To modify the creation of objects in a user-defined class, we must supply a modified implementation of the \_\_new\_\_() magic method. We need to provide the first argument as the reference to the class whose object is to be created for this static function.

**Code**

1. # Python program to show how \_\_new\_\_ method works
3. # Creating a class
4. **class** Method(object):
5. **def** \_\_new\_\_( cls ):
6. **print**( "Creating an instance by \_\_new\_\_ method")
7. **return** super(Method, cls).\_\_new\_\_(cls)
8. # Calling the init method
9. **def** \_\_init\_\_( self ):
10. **print**( "Init method is called here" )
12. Method()

**Output:**

Creating an instance by \_\_new\_\_ method

Init method is called here

<\_\_main\_\_.Method at 0x30dfb88>

\_\_add\_\_ Method

We use the magic method \_\_add\_\_to add the class instance's attributes. Consider the scenario where object1 belongs to class Method and object2 belongs to class Method 1, both of which have the same attribute called "attribute" that stores any value passed to the class while creating the instance. If specified to add the attributes, the \_\_add\_\_ function implicitly adds the instances' same attributes, such as object1.attribute + object2.attribute, when the action object1 + object2 is completed.

Below is the code to show how we add two attributes of two instances of different classes without using the \_\_add\_\_ magic method.

**Code**

1. # Python program to show how to add two attributes
3. # Creating a class
4. **class** Method:
5. **def** \_\_init\_\_(self, argument):
6. self.attribute = argument
8. # Creating a second class
9. **class** Method\_2:
10. **def** \_\_init\_\_(self, argument):
11. self.attribute = argument
12. # creating the instances
13. instance\_1 = Method(" Attribute")
14. **print**(instance\_1.attribute)
15. instance\_2 = Method\_2(" 27")
16. **print**(instance\_2.attribute)
18. # Adding two attributes of the instances
19. **print**(instance\_2.attribute + instance\_1.attribute)

**Output:**

Attribute

27

27 Attribute

By using \_\_add\_\_ magic method the code changes to this.

**Code**

1. # Python program to show how \_\_add\_\_ method works
3. # Creating a class
4. **class** Method:
5. **def** \_\_init\_\_(self, argument):
6. self.attribute = argument
7. **def** \_\_add\_\_(self, object1):
8. **return** self.attribute + object1.attribute
10. # Creating a second class
11. **class** Method\_2:
12. **def** \_\_init\_\_(self, argument):
13. self.attribute = argument
14. **def** \_\_add\_\_(self, object1):
15. **return** self.attribute + object1.attribute
16. instance\_1 = Method(" Attribute")
17. **print**(instance\_1)
18. instance\_2 = Method\_2(" 27")
19. **print**(instance\_2)
20. **print**(instance\_2 + instance\_1)

**Output:**

<\_\_main\_\_.Method object at 0x37470f0>

<\_\_main\_\_.Method\_2 object at 0x376beb8>

27 Attribute

Classes, Method and Method\_1 in the script above have a property called "attribute" that stores a string. We create two instances, instance\_1 and instances\_2, with corresponding attributes of " Attribute" and " 27" values. The \_\_add\_\_ method is used to translate the action instance\_1 + instance\_2 into instance\_1 + instance\_2.attribute, which produces output ( 27 Attribute).

\_\_repr\_\_ Method

The class instance is represented as a string using the magic method \_\_repr\_\_. The \_\_repr\_\_ method, which produces a string in the output, is automatically called whenever we attempt to print an object of that class.

**Code**

1. # Python program to show how \_\_repr\_\_ magic method works
3. # Creating a class
4. **class** Method:
5. # Calling \_\_init\_\_ method and initializing the attributes of the class
6. **def** \_\_init\_\_(self, x, y, z):
7. self.x = x
8. self.y = y
9. self.z = z
10. # Calling the \_\_repr\_\_ method and providing the string to be printed each time instance is printe
11. **def** \_\_repr\_\_(self):
12. **return** f"Following are the values of the attributes of the class Method:\nx = {self.x}\ny = {self.y}\nz = {self.z}"
13. instance = Method(4, 6, 2)
14. **print**(instance)

**Output:**

Following are the values of the attributes of the class Method:

x = 4

y = 6

z = 2

\_\_contains\_\_ Method

The 'in' membership operator of Python implicitly calls the \_\_contains\_\_ method. We can use the \_\_contains\_\_ method to determine if an element is contained in an object's attributes. We can use this method for attributes that are containers ( such as lists, tuples, etc.).

**Code**

1. # Python code to show how the \_\_contains\_\_ magic method works
3. # Creating a class
4. **class** Method:
5. # Calling the \_\_init\_\_ method and initializing the attributes
6. **def** \_\_init\_\_(self, attribute):
7. self.attribute = attribute
9. # Calling the \_\_contains\_\_ method
10. **def** \_\_contains\_\_(self, attribute):
11. **return** attribute **in** self.attribute
12. # Creating an instance of the class
13. instance = Method([4, 6, 8, 9, 1, 6])
15. # Checking if a value is present in the container attribute
16. **print**("4 is contained in ""attribute"": ", 4 **in** instance)
17. **print**("5 is contained in ""attribute"": ", 5 **in** instance)

**Output:**

4 is contained in attribute: True

5 is contained in attribute: False

We have used the \_\_contanis\_\_ magic method in the program above to determine if a given integer is contained in the attribute "attribute". In this case, "attribute" is a list of integers. The integer 4 is present in the given list of integers passed to the class Method as an attribute. While 5 is not present in the list.

\_\_call\_\_ Method

When a class instance is called, the Python interpreter calls the magic method \_\_call\_\_. We can utilise the \_\_call\_\_ method to explicitly call an operation using the instance name rather than creating an additional method to carry out specific activities.

**Code**

1. # Python program to show how the \_\_call\_\_ magic method works
3. # Creating a class
4. **class** Method:
5. # Calling the \_\_init\_\_ method and initializing the attributes
6. **def** \_\_init\_\_(self, a):
7. self.a = a
8. # Calling the \_\_call\_\_ method to multiply a number to the attribute value
9. **def** \_\_call\_\_(self, number):
10. **return** self.a \* number
12. # Creating an instance and proving the value to the attribute a
13. instance = Method(7)
14. **print**(instance.a) # Printing the value of the attribute a
15. # Calling the instance while passing a value which will call the \_\_call\_\_ method
16. **print**(instance(5))

**Output:**

7

35

\_\_iter\_\_ Method

For the given instance, a generator object is supplied using the \_\_iter\_\_ method. To benefit from the \_\_iter\_\_ method, we can leverage the iter() and next() methods.

**Code**

1. # Python program to show how the \_\_iter\_\_ method works
3. # Creating a class
4. **class** Method:
5. **def** \_\_init\_\_(self, start\_value, stop\_value):
6. self.start = start\_value
7. self.stop = stop\_value
8. **def** \_\_iter\_\_(self):
9. **for** num **in** range(self.start, self.stop + 1):
10. **yield** num \*\* 2
11. # Creating an instance
12. instance = iter(Method(3, 8))
13. **print**( next(instance) )
14. **print**( next(instance) )
15. **print**( next(instance) )
16. **print**( next(instance) )
17. **print**( next(instance) )
18. **print**( next(instance) )

**Output:**

9

16

25

36

49

64

We have calculated the squares of the numbers in the code. For the numbers in the specified range, squares are calculated in the program above (start and stop). The \_\_iter\_\_ method, which generates squares of the numbers between the given range, is called when we call the function iter(Method(3, 8)). In our example, we're using the range of 3 to 8; therefore, calling the next() method will produce the results 9, 16, 25, 36, 49, 64.

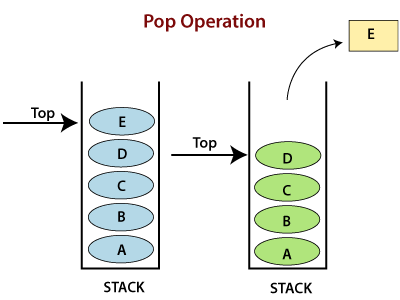
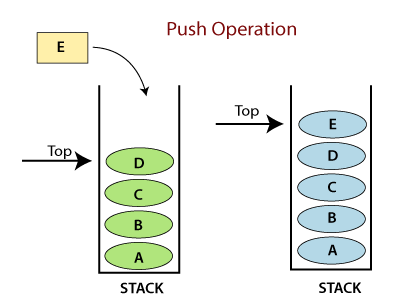
# Python Stack and Queue

Data structure organizes the storage in computers so that we can easily access and change data. Stacks and Queues are the earliest data structure defined in computer science. A simple Python list can act as a queue and stack as well. A queue follows FIFO rule (First In First Out) and used in programming for sorting. It is common for stacks and queues to be implemented with an array or linked list.

## Stack

A Stack is a data structure that follows the LIFO(Last In First Out) principle. To implement a stack, we need two simple operations:

* **push -** It adds an element to the top of the stack.
* **pop -** It removes an element from the top of the stack.



**Operations:**

* **Adding -** It adds the items in the stack and increases the stack size. The addition takes place at the top of the stack.
* **Deletion -** It consists of two conditions, first, if no element is present in the stack, then underflow occurs in the stack, and second, if a stack contains some elements, then the topmost element gets removed. It reduces the stack size.
* **Traversing -** It involves visiting each element of the stack.

**Characteristics:**

* Insertion order of the stack is preserved.
* Useful for parsing the operations.
* Duplicacy is allowed.

**Code**

1. # Code to demonstrate Implementation of
2. # stack using list
3. x = ["Python", "C", "Android"]
4. x.push("Java")
5. x.push("C++")
6. **print**(x)
7. **print**(x.pop())
8. **print**(x)
9. **print**(x.pop())
10. **print**(x)

**Output:**

['Python', 'C', 'Android', 'Java', 'C++']

C++

['Python', 'C', 'Android', 'Java']

Java

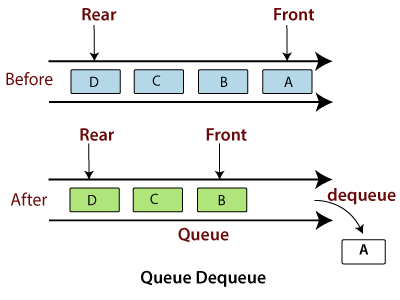
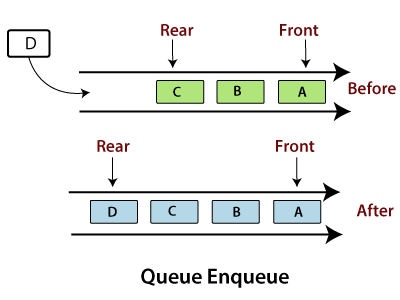
['Python', 'C', 'Android']

## Queue

A Queue follows the First-in-First-Out (FIFO) principle. It is opened from both the ends hence we can easily add elements to the back and can remove elements from the front.

To implement a queue, we need two simple operations:

* **enqueue -** It adds an element to the end of the queue.
* **dequeue -** It removes the element from the beginning of the queue.



**Operations on Queue**

* **Addition -** It adds the element in a queue and takes place at the rear end, i.e., at the back of the queue.
* **Deletion -** It consists of two conditions - If no element is present in the queue, Underflow occurs in the queue, or if a stack contains some elements then element present at the front gets deleted.
* **Traversing -** It involves to visit each element of the queue.

**Characteristics**

* Insertion order of the queue is preserved.
* Duplicacy is allowed.
* Useful for parsing CPU task operations.

#### Note: The implementation of a queue is a little bit different. A queue follows the "First-In-First-Out". Time plays an important factor here. The Stack is fast because we insert and pop the elements from the end of the list, whereas in the queue, the insertion and pops are made from the beginning of the list, so it becomes slow. The cause of this time difference is due to the properties of the list, which is fast in the end operation but slow at the beginning operations because all other elements have to be shifted one by one.

### Code

1. **import** queue
2. # Queue is created as an object 'L'
3. L = queue.Queue(maxsize=10)
5. # Data is inserted in 'L' at the end using put()
6. L.put(9)
7. L.put(6)
8. L.put(7)
9. L.put(4)
10. # get() takes data from
11. # from the head
12. # of the Queue
13. **print**(L.get())
14. **print**(L.get())
15. **print**(L.get())
16. **print**(L.get())

**Output:**

9

6

7

4

# PySpark MLlib

Machine Learning is a technique of data analysis that combines data with statistical tools to predict the output. This prediction is used by the various corporate industries to make a favorable decision.

PySpark provides an API to work with the Machine learning called as **mllib**. PySpark's mllib supports various machine learning algorithms like classification, regression clustering, collaborative filtering, and dimensionality reduction as well as underlying optimization primitives. Various machine learning concepts are given below:

* **classification**

The **pyspark.mllib** library supports several classification methods such as binary classification, multiclass classification, and regression analysis. The object may belong to a different class. The objective of classification is to differentiate the data based on the information. **Random Forest, Naive Bayes, Decision Tree** are the most useful algorithms in classification.

* **clustering**

Clustering is an unsupervised machine learning problem. It is used when you do not know how to classify the data; we require the algorithm to find patterns and classify the data accordingly. The popular clustering algorithms are the **K-means clustering, Gaussian mixture model, Hierarchical clustering.**

* **fpm**

The fpm means frequent pattern matching, which is used for mining various items, itemsets, subsequences, or other substructure. It is mostly used in large-scale datasets.

* **linalg**

The **mllib.linalg**utilities are used for linear algebra.

* **recommendation**

It is used to define the relevant data for making a recommendation. It is capable of predicting future preference and recommending the top items. For example, Online entertainment platform **Netflix** has a huge collection of movies, and sometimes people face difficulty in selecting the favorite items. This is the field where the recommendation plays an important role.

* **mllib regression**

The regression is used to find the relationship and dependencies between variables. It finds the correlation between each feature of data and predicts the future values.

The mllib package supports many other algorithms, classes, and functions. Here we will understand the basic concept of **pyspak.mllib**.

## MLlib Features

The **PySpark mllib** is useful for iterative algorithms. The features are the following:

* **Extraction:** It extracts features from "row" data.
* **Transformation:** It is used for scaling, converting, or modifying features.
* **Selection:** Selecting a useful subset from a larger set of features.
* **Locality Sensitive Hashing:** It combines aspects of feature transformation with other algorithms.

Let's have a look at the essential libraries of PySpark MLlib.

### MLlib Linear Regression

Linear regression is used to find the relationship and dependencies between variables. Consider the following code:

1. frompyspark.sql **import** SparkSession
2. spark = SparkSession.builder.appName('Customer').getOrCreate()
3. frompyspark.ml.regression **import** LinearRegression
4. dataset = spark.read.csv(r'C:\Users\DEVANSH SHARMA\Ecommerce-Customers.csv')
5. dataset.show(10)

**Output:**

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

| \_c0| \_c1| \_c2| \_c3| \_c4| \_c5| \_c6| \_c7|

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

| Email| Address| Avatar|Avg Session Length| Time on App| Time on Website|Length of Membership|Yearly Amount Spent|

|mstephenson@ferna...|835 Frank TunnelW...| Violet| 34.49726772511229| 12.65565114916675| 39.57766801952616| 4.0826206329529615| 587.9510539684005|

| hduke@hotmail.com|4547 Archer Commo...| DarkGreen| 31.92627202636016|11.109460728682564|37.268958868297744| 2.66403418213262| 392.2049334443264|

| pallen@yahoo.com|24645 Valerie Uni...| Bisque|33.000914755642675|11.330278057777512|37.110597442120856| 4.104543202376424| 487.54750486747207|

|riverarebecca@gma...|1414 David Throug...| SaddleBrown| 34.30555662975554|13.717513665142507| 36.72128267790313| 3.120178782748092| 581.8523440352177|

|mstephens@davidso...|14023 Rodriguez P...|MediumAquaMarine| 33.33067252364639|12.795188551078114| 37.53665330059473| 4.446308318351434| 599.4060920457634|

|alvareznancy@luca...|645 Martha Park A...| FloralWhite|33.871037879341976|12.026925339755056| 34.47687762925054| 5.493507201364199| 637.102447915074|

|katherine20@yahoo...|68388 Reyes Light...| DarkSlateBlue| 32.02159550138701|11.366348309710526| 36.68377615286961| 4.685017246570912| 521.5721747578274|

| awatkins@yahoo.com|Unit 6538 Box 898...| Aqua|32.739142938380326| 12.35195897300293| 37.37335885854755| 4.4342734348999375| 549.9041461052942|

|vchurch@walter-ma...|860 Lee KeyWest D...| Salmon| 33.98777289568564|13.386235275676436|37.534497341555735| 3.2734335777477144| 570.2004089636196|

+--------------------+--------------------+----------------+------------------+------------------+------------------+--------------------+-------------------+

only showing top 10 rows

In the following code, we are importing the **VectorAssembler** library to create a new column Independent feature:

1. frompyspark.ml.linalg **import** Vectors
2. frompyspark.ml.feature **import** VectorAssembler
3. featureassembler = VectorAssembler(inputCols = ["Avg Session Length","Time on App","Time on Website"],outputCol = "Independent Features")
4. output = featureassembler.transform(dataset)
5. output.show()

**Output:**

+------------------+

Independent Feature

+------------------+

|34.49726772511229 |

|31.92627202636016 |

|33.000914755642675|

|34.30555662975554 |

|33.33067252364639 |

|33.871037879341976|

|32.02159550138701 |

|32.739142938380326|

|33.98777289568564 |

+------------------+

1. z = featureassembler.transform(dataset)
2. finlized\_data = z.select("Indepenent feature", "Yearly Amount Spent",)
3. z.show()

**Output:**

+--------------------++-------------------+

|Independent Feature | Yearly Amount Spent|

+--------------------++-------------------+

|34.49726772511229 | 587.9510539684005 |

|31.92627202636016 | 392.2049334443264 |

|33.000914755642675 | 487.5475048674720 |

|34.30555662975554 | 581.8523440352177 |

|33.33067252364639 | 599.4060920457634 |

|33.871037879341976 | 637.102447915074 |

|32.02159550138701 | 521.5721747578274 |

|32.739142938380326 | 549.9041461052942 |

|33.98777289568564 | 570.2004089636196 |

+--------------------++-------------------+

PySpark provides the **LinearRegression()** function to find the prediction of any given dataset. The syntax is given below:

1. regressor = LinearRegression(featureCol = 'column\_name1', labelCol = 'column\_name2 ')

## MLlib K- Mean Cluster

The K- Mean cluster algorithm is one of the most popular and commonly used algorithms. It is used to cluster the data points into a predefined number of clusters. The below example is showing the use of MLlib K-Means Cluster library:

1. from pyspark.ml.clustering **import** KMeans
2. from pyspark.ml.evaluation **import** ClusteringEvaluator
3. # Loads data.
4. dataset = spark.read.format("libsvm").load(r"C:\Users\DEVANSH SHARMA\Iris.csv")
5. # Trains a k-means model.
6. kmeans = KMeans().setK(2).setSeed(1)
7. model = kmeans.fit(dataset)
8. # Make predictions
9. predictions = model.transform(dataset)
10. # Evaluate clustering by computing Silhouette score
11. evaluator = ClusteringEvaluator()
12. silhouette = evaluator.evaluate(predictions)
13. print("Silhouette with squared euclidean distance = " + str(silhouette))
14. # Shows the result.
15. centers = model.clusterCenters()
16. print("Cluster Centers: ")
17. **for** center in centers:
18. print(center)

### Parameters of PySpark MLlib

The few important parameters of **PySpark MLlib** are given below:

* **Ratings**

It is RDD of Ratings or (userID, productID, rating) tuple.

* **Rank**

It represents Rank of the computed feature matrices (number of features).

* **Iterations**

It represents the number of iterations of ALS. (default: 5)

* **Lambda**

It is the Regularization parameter. (default : 0.01)

* **Blocks**

It is used to parallelize the computation of some number of blocks.

### Collaborative Filtering (mllib.recommendation)

Collaborative filtering is a technique that is generally used for a recommender system. This technique is focused on filling the missing entries of a user-item. Association matrix **spark.ml** currently supports model-based collaborative filtering. In collaborative filtering, users and products are described by a small set of hidden factors that can be used to predict missing entries.

### Scaling of the regularization parameter

The regularization parameter **regParam** is scaled to solve least-squares problem. The least-square problem occurs when the number of ratings are user-generated in updating user factors, or the number of ratings the product received in updating product factors.

### Cold-start strategy

The **ALS Model (Alternative Least Square Model)** is used for prediction while making a common prediction problem. The problem encountered when user or items in the test dataset occurred that may not be present during training the model. It can occur in the two scenarios which are given below:

* In the prediction, the model is not trained for users and items that have no rating history (it is called a cold-start strategy).
* The data is splitted between training and evaluation sets during cross-validation. It is widespread to encounter users and items in the evaluation set that are not in the training set.

Let's consider the following example, where we load ratings data from the MovieLens dataset. Each row is containing a user, a movie, rating and a timestamp.

1. #importing the libraries
2. frompyspark.ml.evaluation **import** RegressionEvaluator
3. frompyspark.ml.recommendation **import** ALS
4. frompyspark.sql **import** Row
5. no\_of\_lines = spark.read.text(r"C:\Users\DEVANSH SHARMA\MovieLens.csv").rdd
6. no\_of\_parts = no\_of\_lines.map(lambda row: row.value.split("::"))
7. ratingsRDD = no\_of\_lines.map(lambda p: Row(userId=**int**(p[0]), movieId=**int**(p[1]),
8. rating=**float**(p[2]), timestamp=**long**(p[3])))
9. ratings = spark.createDataFrame(ratingsRDD)
10. (training, test) = ratings.randomSplit([0.8, 0.2])
12. # Develop the recommendation model using ALS on the training data
13. # Note we set cold start strategy to make sure that we don't get NaN evaluation metrics.
14. als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating",
15. coldStartStrategy="drop")
16. model = als.fit(training)
18. # Calculate the model by computing the RMSE on the test data
19. predictions = model.transform(test)
20. evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
21. predictionCol="prediction")
22. rmse = evaluator.evaluate(predictions)
23. print("Root-mean-square error = " + str(rmse))
25. # Evaluate top 10 movie recommendations **for** each user
26. userRecs = model.recommendForAllUsers(10)
27. # Evaluate top 10 user recommendations **for** each movie
28. movieRecs = model.recommendForAllItems(10)
29. # Evaluate top 10 movie recommendations **for** a specified set of users
30. users = ratings.select(als.getUserCol()).distinct().limit(3)
31. userSubsetRecs = model.recommendForUserSubset(users, 10)
32. # Evalute top 10 user recommendations **for** a specified set of movies
33. movies = ratings.select(als.getItemCol()).distinct().limit(3)
34. movieSubSetRecs = model.recommendForItemSubset(movies, 10)

# Python Decorator

Decorators are one of the most helpful and powerful tools of Python. 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.

In Decorators, functions are passed as an argument into another function and then called inside the wrapper function.

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

Before understanding the **Decorator**, we need to know some important concepts of Python.

## What are the functions in Python?

Python has the most interesting feature that everything is treated as an object even classes or any variable we define in Python is also assumed as an object. Functions are **first-class** objects in the Python because they can reference to, passed to a variable and returned from other functions as well. The example is given below:

1. def func1(msg):
2. print(msg)
3. func1("Hii")
4. func2 = func1
5. func2("Hii")

**Output:**

Hii

Hii

In the above program, when we run the code it give the same output for both functions. The **func2**referred to function **func1** and act as function. We need to understand the following concept of the function:

* The function can be referenced and passed to a variable and returned from other functions as well.
* The functions can be declared inside another function and passed as an argument to another function.

## Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions. Consider the following example:

1. def func():
2. print("We are in first function")
3. def func1():
4. print("This is first child function")
5. def func2():
6. print(" This is second child function")
7. func1()
8. func2()
9. func()

**Output:**

We are in first function

This is first child function

This is second child function

In the above program, it doesn't matter how the child functions are declared. The execution of the child function makes effect on the output. These child functions are locally bounded with the **func()** so they cannot be called separately.

A function that accepts other function as an argument is also called **higher order function**. Consider the following example:

1. def add(x):
2. **return** x+1
3. def sub(x):
4. **return** x-1
5. def operator(func, x):
6. temp = func(x)
7. **return** temp
8. print(operator(sub,10))
9. print(operator(add,20))

**Output:**

9

21

In the above program, we have passed the **sub()** function and **add()** function as argument in **operator()** function.

A function can return another function. Consider the below example:

1. def hello():
2. def hi():
3. print("Hello")
4. **return** hi
5. **new** = hello()
6. **new**()

**Output:**

Hello

In the above program, the **hi()** function is nested inside the **hello()** function. It will return each time we call **hi()**.

### Decorating functions with parameters

Let's have an example to understand the parameterized decorator function:

1. def divide(x,y):
2. print(x/y)
3. def outer\_div(func):
4. def inner(x,y):
5. **if**(x<y):
6. x,y = y,x
7. **return** func(x,y)
8. **return** inner
9. divide1 = outer\_div(divide)
10. divide1(2,4)

**Output:**

2.0

### Syntactic Decorator

In the above program, we have decorated **out\_div()** that is little bit bulky. Instead of using above method, Python allows to **use decorator in easy way with @symbol**. Sometimes it is called "pie" syntax.

1. def outer\_div(func):
2. def inner(x,y):
3. **if**(x<y):
4. x,y = y,x
5. **return** func(x,y)
6. **return** inner
7. # syntax of generator
8. @outer\_div
9. def divide(x,y):
10. print(x/y)

**Output:**

2.0

### Reusing Decorator

We can reuse the decorator as well by recalling that decorator function. Let's make the decorator to its own module that can be used in many other functions. Creating a file called **mod\_decorator.py** with the following code:

1. def do\_twice(func):
2. def wrapper\_do\_twice():
3. func()
4. func()
5. **return** wrapper\_do\_twice

We can import mod\_decorator.py in other file.

1. from decorator **import** do\_twice
2. @do\_twice
3. def say\_hello():
4. print("Hello There")
5. say\_hello()

**Output:**

Hello There

Hello There

### Python Decorator with Argument

We want to pass some arguments in function. Let's do it in following code:

1. from decorator **import** do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display()

**Output:**

TypeError: display() missing 1 required positional argument: 'name'

As we can see that, the function didn't accept the argument. Running this code raises an error. We can fix this error by using **\*args** and **\*\*kwargs**in the inner wrapper function. Modifying the **decorator.py**as follows:

1. def do\_twice(func):
2. def wrapper\_function(\*args,\*\*kwargs):
3. func(\*args,\*\*kwargs)
4. func(\*args,\*\*kwargs)
5. **return** wrapper\_function

Now **wrapper\_function()** can accept any number of argument and pass them on the function.

1. from decorator **import** do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display("John")

**Output:**

Hello John

Hello John

### Returning Values from Decorated Functions

We can control the return type of the decorated function. The example is given below:

1. from decorator **import** do\_twice
2. @do\_twice
3. def return\_greeting(name):
4. print("We are created greeting")
5. **return** f"Hi {name}"
6. hi\_adam = return\_greeting("Adam")

**Output:**

We are created greeting

We are created greeting

## Fancy Decorators

Let's understand the fancy decorators by the following topic:

### Class Decorators

Python provides two ways to decorate a class. Firstly, we can decorate the method inside a class; there are built-in decorators like **@classmethod, @staticmethod** and **@property** in Python. The **@classmethod** and **@staticmethod** define methods inside class that is not connected to any other instance of a class. The @property is generally used to modify the getters and setters of a class attributes. Let’s understand it by the following example:

Example: 1- **@property decorator** - By using it, we can use the class function as an attribute. Consider the following code:

1. **class** Student:
2. def \_\_init\_\_(self,name,grade):
3. self.name = name
4. self.grade = grade
5. @property
6. def display(self):
7. **return** self.name + " got grade " + self.grade
9. stu = Student("John","B")
10. print("Name:", stu.name)
11. print("Grade:", stu.grade)
12. print(stu.display)

**Output:**

Name: John

Grade: B

John got grade B

Example:2 - **@staticmethod decorator**- The @staticmethod is used to define a static method in the class. It is called by using the class name as well as instance of the class. Consider the following code:

1. **class** Person:
2. @staticmethod
3. def hello():
4. print("Hello Peter")
5. per = Person()
6. per.hello()
7. Person.hello()

**Output:**

Hello Peter

Hello Peter

### Singleton Class

A singleton class only has one instance. There are many singletons in Python including True, None, etc.

### Nesting Decorators

We can use multiple decorators by using them on top of each other. Let's consider the following example:

1. @function1
2. @function2
3. def function(name):
4. print(f "{name}")

In the above code, we have used the nested decorator by stacking them onto one another.

### Decorator with Arguments

It is always useful to pass arguments in a decorator. The decorator can be executed several times according to the given value of the argument. Let us consider the following example:

1. Import functools
3. def repeat(num):
5. #Creating and returning a wrapper function
6. def decorator\_repeat(func):
7. @functools.wraps(func)
8. def wrapper(\*args,\*\*kwargs):
9. **for** \_ in range(num):
10. value = func(\*args,\*\*kwargs)
11. **return** value
12. **return** wrapper
13. **return** decorator\_repeat
15. #Here we are passing num as an argument which repeats the print function
16. @repeat(num=5)
17. def function1(name):
18. print(f"{name}")

**Output:**

JavatPoint

JavatPoint

JavatPoint

JavatPoint

JavatPoint

In the above example, **@repeat**refers to a function object that can be called in another function. The **@repeat(num = 5)**will return a function which acts as a decorator.

The above code may look complex but it is the most commonly used decorator pattern where we have used one additional **def** that handles the arguments to the decorator.

#### Note: Decorator with argument is not frequently used in programming, but it provides flexibility. We can use it with or without argument.

### Stateful Decorators

Stateful decorators are used to keep track of the decorator state. Let us consider the example where we are creating a decorator that counts how many times the function has been called.

1. Import functools
3. def count\_function(func):
4. @functools.wraps(func)
5. def wrapper\_count\_calls(\*args, \*\*kwargs):
6. wrapper\_count\_calls.num\_calls += 1
8. print(f"Call{wrapper\_count\_calls.num\_calls} of {func.\_\_name\_\_!r}")
9. **return** func(\*args, \*\*kwargs)
11. wrapper\_count\_calls.num\_calls = 0
12. **return** wrapper\_count\_calls
14. @count\_function
15. def say\_hello():
16. print("Say Hello")
18. say\_hello()
19. say\_hello()

**Output:**

Call 1 of 'say\_hello'

Say Hello

Call 2 of 'say\_hello'

Say Hello

In the above program, the state represented the number of calls of the function stored in **.num\_calls**on the wrapper function. When we call **say\_hello()**it will display the number of the call of the function.

### Classes as Decorators

The classes are the best way to maintain state. In this section, we will learn how to use a class as a decorator. Here we will create a class that contains **\_\_init\_\_()** and take **func** as an argument. The class needs to be callable so that it can stand in for the decorated function.

To making a class callable, we implement the special **\_\_call\_\_()** method.

1. **import** functools
3. **class** Count\_Calls:
4. def \_\_init\_\_(self, func):
5. functools.update\_wrapper(self, func)
6. self.func = func
7. self.num\_calls = 0
9. def \_\_call\_\_(self, \*args, \*\*kwargs):
10. self.num\_calls += 1
11. print(f"Call{self.num\_calls} of {self.func.\_\_name\_\_!r}")
12. **return** self.func(\*args, \*\*kwargs)
14. @Count\_Calls
15. def say\_hello():
16. print("Say Hello")
18. say\_hello()
19. say\_hello()
20. say\_hello()

**Output:**

Call 1 of 'say\_hello'

Say Hello

Call 2 of 'say\_hello'

Say Hello

Call 3 of 'say\_hello'

Say Hello

The **\_\_init\_\_()** method stores a reference to the function and can do any other required initialization.

# Python Generators

## What is Python Generator?

Python Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once. The generator can also be an expression in which syntax is similar to the list comprehension in Python.

There is a lot of complexity in creating iteration in Python; we need to implement **\_\_iter\_\_()** and **\_\_next\_\_()** method to keep track of internal states.

It is a lengthy process to create iterators. That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

## How to Create Generator function in Python?

It is quite simple to create a generator in Python. It is similar to the normal function defined by the **def** keyword and uses a **yield** keyword instead of return. Or we can say that if the body of any function contains a **yield** statement, it automatically becomes a generator function. Consider the following example:

1. def simple():
2. **for** i in range(10):
3. **if**(i%2==0):
4. yield i
6. #Successive Function call using **for** loop
7. **for** i in simple():
8. print(i)

**Output:**

0

2

4

6

8

### yield vs. return

The **yield** statement is responsible for controlling the flow of the generator function. It pauses the function execution by saving all states and yielded to the caller. Later it resumes execution when a successive function is called. We can use the multiple yield statement in the generator function.

The return statement **returns** a value and terminates the whole function and only one return statement can be used in the function.

**Using multiple yield Statement**

We can use the multiple yield statement in the generator function. Consider the following example.

1. def multiple\_yield():
2. str1 = "First String"
3. yield str1
5. str2 = "Second string"
6. yield str2
8. str3 = "Third String"
9. yield str3
10. obj = multiple\_yield()
11. print(next(obj))
12. print(next(obj))
13. print(next(obj))

**Output:**

First String

Second string

Third String

### Difference between Generator function and Normal function

* Normal function contains only one L**return** statement whereas generator function can contain one or more **yield** statement.
* When the generator functions are called, the normal function is paused immediately and control transferred to the caller.
* Local variable and their states are remembered between successive calls.
* StopIteration exception is raised automatically when the function terminates.

### Generator Expression

We can easily create a generator expression without using user-defined function. It is the same as the lambda function which creates an anonymous function; the generator's expressions create an anonymous generator function.

The representation of generator expression is similar to the Python list comprehension. The only difference is that **square bracket is replaced by round parentheses**. The list comprehension calculates the entire list, whereas the generator expression calculates one item at a time.

Consider the following example:

1. list = [1,2,3,4,5,6,7]
3. # List Comprehension
4. z = [x\*\*3 **for** x in list]
6. # Generator expression
7. a = (x\*\*3 **for** x in list)
9. print(a)
10. print(z)

**Output:**

<generator object <genexpr> at 0x01BA3CD8>

[1, 8, 27, 64, 125, 216, 343]

In the above program, list comprehension has returned the list of cube of elements whereas generator expression has returned the reference of calculated value. Instead of applying a **for loop**, we can also call **next()** on the generator object. Let's consider another example:

1. list = [1,2,3,4,5,6]
3. z = (x\*\*3 **for** x in list)
5. print(next(z))
7. print(next(z))
9. print(next(z))
11. print(next(z))

**Output:**

1

8

27

64

#### Note:- When we call the next(), Python calls \_\_next\_\_() on the function in which we have passed it as a parameter.

In the above program, we have used the **next()** function, which returned the next item of the list.

**Example:** Write a program to print the table of the given number using the generator.

1. def table(n):
2. **for** i in range(1,11):
3. yield n\*i
4. i = i+1
6. **for** i in table(15):
7. print(i)

**Output:**

15

30

45

60

75

90

105

120

135

150

In the above example, a generator function is iterating using for loop.

## Advantages of Generators

There are various advantages of Generators. Few of them are given below:

### 1. Easy to implement

Generators are easy to implement as compared to the iterator. In iterator, we have to implement **\_\_iter\_\_()** and **\_\_next\_\_()** function.

### 2. Memory efficient

Generators are memory efficient for a large number of sequences. The normal function returns a sequence of the list which creates an entire sequence in memory before returning the result, but the generator function calculates the value and pause their execution. It resumes for successive call. An infinite sequence generator is a great example of memory optimization. Let's discuss it in the below example by using **sys.getsizeof()** function.

1. **import** sys
2. # List comprehension
3. nums\_squared\_list = [i \* 2 **for** i in range(1000)]
4. print(sys.getsizeof("Memory in Bytes:"nums\_squared\_list))
5. # Generator Expression
6. nums\_squared\_gc = (i \*\* 2 **for** i in range(1000))
7. print(sys.getsizeof("Memory in Bytes:", nums\_squared\_gc))

**Output:**

Memory in Bytes: 4508

Memory in Bytes: 56

We can observe from the above output that list comprehension is using 4508 bytes of memory, whereas generator expression is using 56 bytes of memory. It means that generator objects are much efficient than the list compression.

### 3. Pipelining with Generators

Data Pipeline provides the facility to process large datasets or stream of data without using extra computer memory.

Suppose we have a log file from a famous restaurant. The log file has a column (4th column) that keeps track of the number of burgers sold every hour and we want to sum it to find the total number of burgers sold in 4 years. In that scenario, the generator can generate a pipeline with a series of operations. Below is the code for it:

1. with open('sells.log') as file:
2. burger\_col = (line[3] **for** line in file)  per\_hour = (**int**(x) **for** x in burger\_col **if** x != 'N/A')
3. print("Total burgers sold = ",sum(per\_hour))

**4. Generate Infinite Sequence**

The generator can produce infinite items. Infinite sequences cannot be contained within the memory and since generators produce only one item at a time, consider the following example:

1. def infinite\_sequence():
2. num = 0
3. **while** True:
4. yield num
5. num += 1
7. **for** i in infinite\_sequence():
8. print(i)

**Output:**

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

.........

..........

315

316

317

Traceback (most recent call last):

File "C:\Users\DEVANSH SHARMA\Desktop\generator.py", line 33, in <module>

print(i)

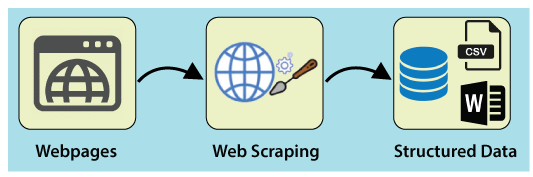
KeyboardInterrupt

In this tutorial, we have learned about the Python Generators.

# Web Scraping Using Python

## What is Web Scraping?

Web Scraping is a technique to extract a large amount of data from several websites. The term **"scraping"** refers to obtaining the information from another source (webpages) and saving it into a local file. For example: Suppose you are working on a project called **"Phone comparing website,"** where you require the price of mobile phones, ratings, and model names to make comparisons between the different mobile phones. If you collect these details by checking various sites, it will take much time. In that case, web scrapping plays an important role where by writing a few lines of code you can get the desired results.



Web Scrapping extracts the data from websites in the unstructured format. It helps to collect these unstructured data and convert it in a structured form.

Startups prefer web scrapping because it is a cheap and effective way to get a large amount of data without any partnership with the data selling company.

## Is Web Scrapping legal?

Here the question arises **whether the web scrapping is legal or not**. The answer is that some sites allow it when used legally. Web scraping is just a tool you can use it in the right way or wrong way.

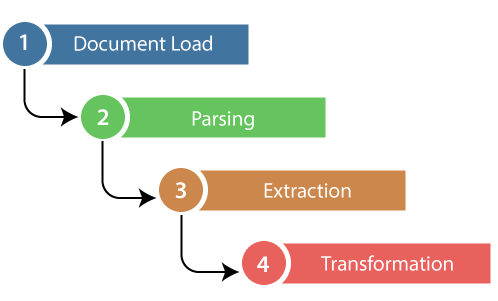
Play Videox

Web scrapping is illegal if someone tries to scrap the nonpublic data. Nonpublic data is not reachable to everyone; if you try to extract such data then it is a violation of the legal term.

There are several tools available to scrap data from websites, such as:

* Scrapping-bot
* Scrapper API
* Octoparse
* Import.io
* Webhose.io
* Dexi.io
* Outwit
* Diffbot
* Content Grabber
* Mozenda
* Web Scrapper Chrome Extension

## Why Web Scrapping?



As we have discussed above, web scrapping is used to extract the data from websites. But we should know how to use that raw data. That raw data can be used in various fields. Let's have a look at the usage of web scrapping:

* **Dynamic Price Monitoring**

It is widely used to collect data from several online shopping sites and compare the prices of products and make profitable pricing decisions. Price monitoring using web scrapped data gives the ability to the companies to know the market condition and facilitate dynamic pricing. It ensures the companies they always outrank others.

* **Market Research**

eb Scrapping is perfectly appropriate for market trend analysis. It is gaining insights into a particular market. The large organization requires a great deal of data, and web scrapping provides the data with a guaranteed level of reliability and accuracy.

* **Email Gathering**

Many companies use personals e-mail data for email marketing. They can target the specific audience for their marketing.

* **News and Content Monitoring**

A single news cycle can create an outstanding effect or a genuine threat to your business. If your company depends on the news analysis of an organization, it frequently appears in the news. So web scraping provides the ultimate solution to monitoring and parsing the most critical stories. News articles and social media platform can directly influence the stock market.

* **Social Media Scrapping**

Web Scrapping plays an essential role in extracting data from social media websites such as **Twitter, Facebook,** and **Instagram,** to find the trending topics.

* **Research and Development**

The large set of data such as **general information, statistics, and temperature** is scrapped from websites, which is analyzed and used to carry out surveys or research and development.

## Why use Python for Web Scrapping?

There are other popular programming languages, but why we choose the [Python](https://www.javatpoint.com/python-tutorial) over other programming languages for web scraping? Below we are describing a list of Python's features that make the most useful programming language for web scrapping.

* **Dynamically Typed**

In Python, we don't need to define data types for variables; we can directly use the variable wherever it requires. It saves time and makes a task faster. Python defines its classes to identify the data type of variable.

* **Vast collection of libraries**

Python comes with an extensive range of libraries such as **NumPy, Matplotlib, Pandas, Scipy, etc**., that provide flexibility to work with various purposes. It is suited for almost every emerging field and also for web scrapping for extracting data and do manipulation.

* **Less Code**

The purpose of the web scrapping is to save time. But what if you spend more time in writing the code? That's why we use Python, as it can perform a task in a few lines of code.

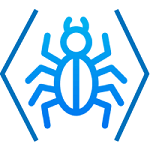
* **Open-Source Community**

Python is open-source, which means it is freely available for everyone. It has one of the biggest communities across the world where you can seek help if you get stuck anywhere in Python code.

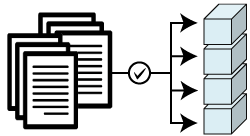
### The basics of web scraping

The web scrapping consists of two parts: **a web crawler and a web scraper**. In simple words, the web crawler is a horse, and the scrapper is the chariot. The crawler leads the scrapper and extracts the requested data. Let's understand about these two components of web scrapping:

* **The crawler**

A web crawler is generally called a **"spider."** It is an artificial intelligence technology that browses the internet to index and searches for the content by given links. It searches for the relevant information asked by the programmer.

 **The scrapper**

A web scraper is a dedicated tool that is designed to extract the data from several websites quickly and effectively. Web scrappers vary widely in design and complexity, depending on the projects.

### How does Web Scrapping work?

These are the following steps to perform web scraping. Let's understand the working of web scraping.

**Step -1: Find the URL that you want to scrape**

First, you should understand the requirement of data according to your project. A webpage or website contains a large amount of information. That's why scrap only relevant information. In simple words, the developer should be familiar with the data requirement.

**Step - 2: Inspecting the Page**

The data is extracted in raw [HTML](https://www.javatpoint.com/html-tutorial) format, which must be carefully parsed and reduce the noise from the raw data. In some cases, data can be simple as name and address or as complex as high dimensional weather and stock market data.

**Step - 3: Write the code**

Write a code to extract the information, provide relevant information, and run the code.

**Step - 4: Store the data in the file**

Store that information in required csv, [xml](https://www.javatpoint.com/xml-tutorial), [JSON](https://www.javatpoint.com/json-tutorial) file format.

### Getting Started with Web Scrapping

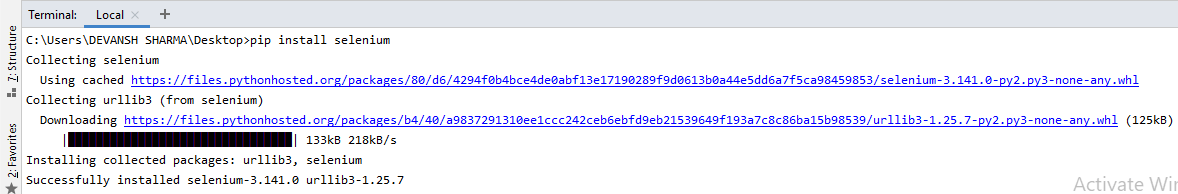
Python has a vast collection of libraries and also provides a very useful library for web scrapping. Let's understand the required library for Python.

**Library used for web scrapping**

* **Selenium-** Selenium is an open-source automated testing library. It is used to check browser activities. To install this library, type the following command in your terminal.

1. pip install selenium

#### Note - It is good to use the PyCharm IDE.



* **Pandas**

Pandas library is used for **data manipulation and analysis**. It is used to extract the data and store it in the desired format.

* **BeautifulSoup**

BeautifulSoup is a Python library that is used to pull data of HTML and XML files. It is mainly designed for web scrapping. It works with the parser to provide a natural way of navigating, searching, and modifying the parse tree. The latest version of BeautifulSoup is 4.8.1.

Let's understand the **BeautifulSoup** library in detail.

**Installation of BeautifulSoup**

You can install BeautifulSoup by typing the following command:

1. pip install bs4

**Installing a parser**

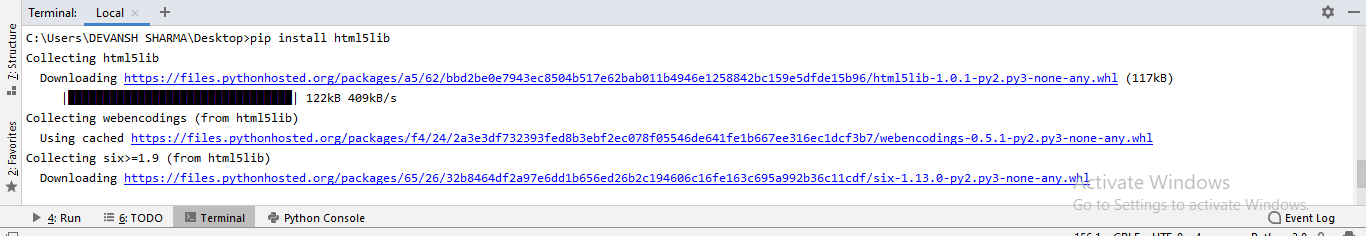
BeautifulSoup supports HTML parser and several third-party Python parsers. You can install any of them according to your dependency. The list of BeautifulSoup's parsers is the following:

|  |  |
| --- | --- |
| **Parser** | **Typical usage** |
| Python's html.parser | BeautifulSoup(markup,"html.parser") |
| lxml's HTML parser | BeautifulSoup(markup,"lxml") |
| lxml's XML parser | BeautifulSoup(markup,"lxml-xml") |
| Html5lib | BeautifulSoup(markup,"html5lib") |

We recommend you to install **html5lib** parser because it is much suitable for the newer version of Python, or you can install **lxml** parser.

Type the following command in your terminal:

1. pip install html5lib



BeautifulSoup is used to transform a complex HTML document into a complex tree of Python objects. But there are a few essential types object which are mostly used:

* **Tag**

A **Tag** object corresponds to an XML or HTML original document.

1. soup = bs4.BeautifulSoup("<b class = "boldest">Extremely bold</b>)
2. tag = soup.b
3. type(tag)

**Output:**

<class "bs4.element.Tag">

Tag contains lot of attributes and methods, but most important features of a tag are name and attribute.

* **Name**

Every tag has a name, accessible as **.name:**

1. tag.name

* **Attributes**

A tag may have any number of attributes. The tag <b id = "boldest"> has an attribute "id" whose value is "boldest". We can access a tag's attributes by treating the tag as dictionary.

1. tag[id]

We can add, remove, and modify a tag's attributes. It can be done by using tag as dictionary.

1. # add the element
2. tag['id'] = 'verybold'
3. tag['another-attribute'] = 1
4. tag
5. # delete the tag
6. del tag['id']

* **Multi-valued Attributes**

In HTML5, there are some attributes that can have multiple values. The class (consists more than one css) is the most common multivalued attributes. Other attributes are **rel, rev, accept-charset, headers,** and **accesskey**.

1. class\_is\_multi= { '\*' : 'class'}
2. xml\_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml', multi\_valued\_attributes=class\_is\_multi)
3. xml\_soup.p['class']
4. # [u'body', u'strikeout']

* **NavigableString**

A string in BeautifulSoup refers text within a tag. BeautifulSoup uses the **NavigableString** class to contain these bits of text.

1. tag.string
2. # u'Extremely bold'
3. type(tag.string)
4. # <**class** 'bs4.element.NavigableString'>

A string is immutable means it can't be edited. But it can be replaced with another string using **replace\_with()**.

1. tag.string.replace\_with("No longer bold")
2. tag

In some cases, if you want to use a **NavigableString** outside the BeautifulSoup, the **unicode()** helps it to turn into normal Python Unicode string.

* **BeautifulSoup object**

The BeautifulSoup object represents the complete parsed document as a whole. In many cases, we can use it as a Tag object. It means it supports most of the methods described in navigating the tree and searching the tree.

1. doc=BeautifulSoup("**<document><content/>**INSERT FOOTER HERE**</document**","xml")
2. footer=BeautifulSoup("**<footer>**Here's the footer**</footer>**","xml")
3. doc.find(text="INSERT FOOTER HERE").replace\_with(footer)
4. print(doc)

**Output:**

?xml version="1.0" encoding="utf-8"?>

# <document><content/><footer>Here's the footer</footer></document>

### Web Scrapping Example:

Let's take an example to understand the scrapping practically by extracting the data from the webpage and inspecting the whole page.

First, open your favorite page on Wikipedia and inspect the whole page, and before extracting data from the webpage, you should ensure your requirement. Consider the following code:

1. #importing the BeautifulSoup Library
3. importbs4
4. **import** requests
6. #Creating the requests
8. res = requests.get("https://en.wikipedia.org/wiki/Machine\_learning")
9. print("The object type:",type(res))
11. # Convert the request object to the Beautiful Soup Object
12. soup = bs4.BeautifulSoup(res.text,'html5lib')
13. print("The object type:",type(soup)

**Output:**

The object type <class 'requests.models.Response'>

Convert the object into: <class 'bs4.BeautifulSoup'>

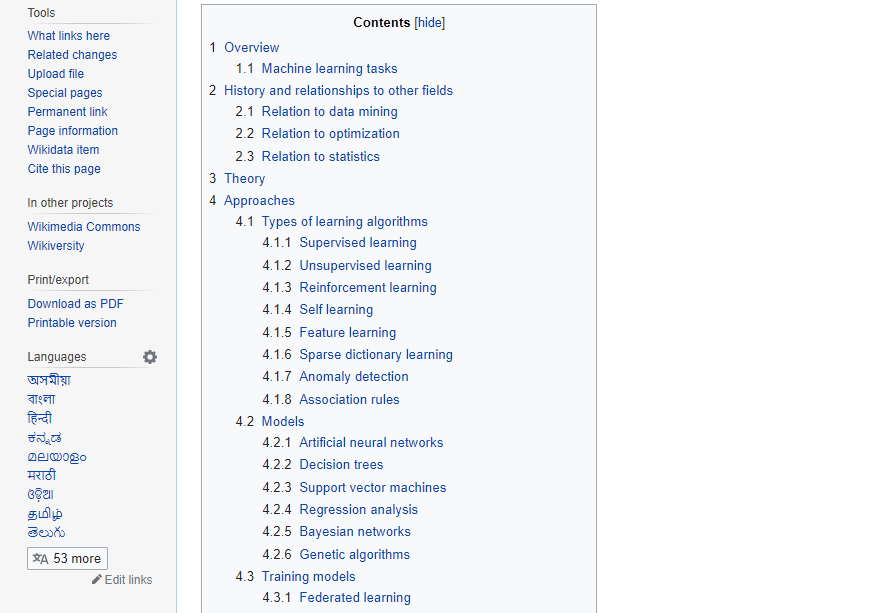
In the following lines of code, we are extracting all headings of a webpage by class name. Here front-end knowledge plays an essential role in inspecting the webpage.

1. soup.select('.mw-headline')
2. **for** i in soup.select('.mw-headline'):
3. print(i.text,end = ',')

**Output:**

Overview,Machine learning tasks,History and relationships to other fields,Relation to data mining,Relation to optimization,Relation to statistics, Theory,Approaches,Types of learning algorithms,Supervised learning,Unsupervised learning,Reinforcement learning,Self-learning,Feature learning,Sparse dictionary learning,Anomaly detection,Association rules,Models,Artificial neural networks,Decision trees,Support vector machines,Regression analysis,Bayesian networks,Genetic algorithms,Training models,Federated learning,Applications,Limitations,Bias,Model assessments,Ethics,Software,Free and open-source software,Proprietary software with free and open-source editions,Proprietary software,Journals,Conferences,See also,References,Further reading,External links,

In the above code, we imported the **bs4** and **requested** the library. In the third line, we created a **res** object to send a request to the webpage. As you can observe that we have extracted all heading from the webpage.



**Webpage of Wikipedia Learning**

Let's understand another example; we will make a GET request to the URL and create a parse Tree object (soup) with the use of BeautifulSoup and Python built-in **"html5lib"** parser.

Here we will scrap the webpage of given link [(https://www.javatpoint.com/).](https://www.javatpoint.com/) Consider the following code:

1. following code:
2. # importing the libraries
3. from bs4 **import** BeautifulSoup
4. **import** requests
6. url="https://www.javatpoint.com/"
8. # Make a GET request to fetch the raw HTML content
9. html\_content = requests.get(url).text
11. # Parse the html content
12. soup = BeautifulSoup(html\_content, "html5lib")
13. print(soup.prettify()) # print the parsed data of html

The above code will display the all html code of javatpoint homepage.

Using the **BeautifulSoup** object, i.e. **soup**, we can collect the required data table. Let's print some interesting information using the **soup** object:

* Let's print the title of the web page.

1. print(soup.title)

**Output:** It will give an output as follow:

<title>Tutorials List - Javatpoint</title>

* In the above output, the HTML tag is included with the title. If you want text without tag, you can use the following code:

1. print(soup.title.text)

**Output:** It will give an output as follow:

Tutorials List - Javatpoint

* We can get the entire link on the page along with its attributes, such as href, title, and its inner Text. Consider the following code:

1. **for** link in soup.find\_all("a"):
2. print("Inner Text is: {}".format(link.text))
3. print("Title is: {}".format(link.get("title")))
4. print("href is: {}".format(link.get("href")))

**Output:** It will print all links along with its attributes. Here we display a few of them:

href is: https://www.facebook.com/javatpoint

Inner Text is:

The title is: None

href is: https://twitter.com/pagejavatpoint

Inner Text is:

The title is: None

href is: https://www.youtube.com/channel/UCUnYvQVCrJoFWZhKK3O2xLg

Inner Text is:

The title is: None

href is: https://javatpoint.blogspot.com

Inner Text is: Learn Java

Title is: None

href is: https://www.javatpoint.com/java-tutorial

Inner Text is: Learn Data Structures

Title is: None

href is: https://www.javatpoint.com/data-structure-tutorial

Inner Text is: Learn C Programming

Title is: None

href is: https://www.javatpoint.com/c-programming-language-tutorial

Inner Text is: Learn C++ Tutorial

### Demo: Scraping Data from Flipkart Website

In this example, we will scrap the mobile phone prices, ratings, and model name from Flipkart, which is one of the popular e-commerce websites. Following are the prerequisites to accomplish this task:

**Prerequisites:**

* Python 2.x or Python 3.x with **Selenium, BeautifulSoup, Pandas** libraries installed.
* Google - chrome browser
* Scrapping Parser such as html.parser, xlml, etc.

**Step - 1: Find the desired URL to scrap**

The initial step is to find the URL that you want to scrap. Here we are extracting mobile phone details from the flipkart. The URL of this page is https://www.flipkart.com/search?q=iphones&otracker=search&otracker1=search&marketplace=FLIPKART&as-show=on&as=off.

**Step -2: Inspecting the page**

It is necessary to inspect the page carefully because the data is usually contained within the tags. So we need to inspect to select the desired tag. To inspect the page, right-click on the element and click **"inspect"**.

**Step - 3: Find the data for extracting**

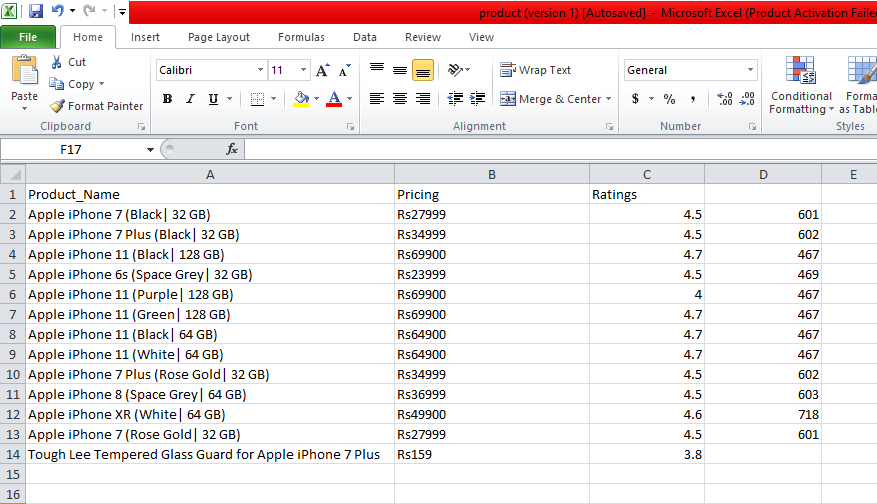
Extract the Price, Name, and Rating, which are contained in the "div" tag, respectively.

**Step - 4: Write the Code**

1. from bs4 **import** BeautifulSoupas soup
2. from urllib.request **import** urlopen as uReq
4. # Request from the webpage
5. myurl = "https://www.flipkart.com/search?q=iphones&otracker=search&otracker1=search&marketplace=FLIPKART&as-show=on&as=off"

8. uClient  = uReq(myurl)
9. page\_html = uClient.read()
10. uClient.close()
12. page\_soup = soup(page\_html, features="html.parser")
14. # print(soup.prettify(containers[0]))
16. # This variable held all html of webpage
17. containers = page\_soup.find\_all("div",{"class": "\_3O0U0u"})
18. # container = containers[0]
19. # # print(soup.prettify(container))
20. #
21. # price = container.find\_all("div",{"class": "col col-5-12 \_2o7WAb"})
22. # print(price[0].text)
23. #
24. # ratings = container.find\_all("div",{"class": "niH0FQ"})
25. # print(ratings[0].text)
26. #
27. # #
28. # # print(len(containers))
29. # print(container.div.img["alt"])
31. # Creating CSV File that will store all data
32. filename = "product1.csv"
33. f = open(filename,"w")
35. headers = "Product\_Name,Pricing,Ratings\n"
36. f.write(headers)
38. **for** container in containers:
39. product\_name = container.div.img["alt"]
41. price\_container = container.find\_all("div", {"class": "col col-5-12 \_2o7WAb"})
42. price = price\_container[0].text.strip()
44. rating\_container = container.find\_all("div",{"class":"niH0FQ"})
45. ratings = rating\_container[0].text
47. # print("product\_name:"+product\_name)
48. # print("price:"+price)
49. # print("ratings:"+ str(ratings))
51. edit\_price = ''.join(price.split(','))
52. sym\_rupee = edit\_price.split("?")
53. add\_rs\_price = "Rs"+sym\_rupee[1]
54. split\_price = add\_rs\_price.split("E")
55. final\_price = split\_price[0]
57. split\_rating = str(ratings).split(" ")
58. final\_rating = split\_rating[0]
60. print(product\_name.replace(",", "|")+","+final\_price+","+final\_rating+"\n")
61. f.write(product\_name.replace(",", "|")+","+final\_price+","+final\_rating+"\n")
63. f.close()

**Output:**



We scrapped the details of the iPhone and saved those details in the CSV file as you can see in the output. In the above code, we put a comment on the few lines of code for testing purpose. You can remove those comments and observe the output.

In this tutorial, we have discussed all basic concepts of web scrapping and described the sample scrapping from the leading online ecommerce site flipkart.

# Python JSON

JSON stands for **JavaScript Object Notation**, which is a widely used data format for data interchange on the web. JSON is the ideal format for organizing data between a client and a server. Its syntax is similar to the JavaScript programming language. The main objective of JSON is to transmit the data between the client and the web server. It is easy to learn and the most effective way to interchange the data. It can be used with various programming languages such as [**Python**](https://www.javatpoint.com/python-tutorial)**,**[**Perl**](https://www.javatpoint.com/perl-tutorial)**,**[**Java**](https://www.javatpoint.com/java-tutorial)**,** etc.

[JSON](https://www.javatpoint.com/json-tutorial) mainly supports 6 types of data type In [JavaScript](https://www.javatpoint.com/javascript-tutorial):

* **String**
* **Number**
* **Boolean**
* **Null**
* **Object**
* **Array**

JSON is built on the two structures:

* It stores data in the name/value pairs. It is treated as an **object, record, dictionary, hash table, keyed list**.
* The ordered list of values is treated as an array, vector, list, or sequence.

JSON data representation is similar to the Python dictionary. Below is an example of JSON data:

Play Videox

1. {
2. "book": [
3. {
4. "id": 01,
5. "language": "English",
6. "edition": "Second",
7. "author": "Derrick Mwiti"
8. ],
9. {
10. {
11. "id": 02,
12. "language": "French",
13. "edition": "Third",
14. "author": "Vladimir"
15. }
16. }

## Working with Python JSON

Python provides a module called **json**. Python supports standard library marshal and pickle module, and JSON API behaves similarly as these library. Python natively supports JSON features.

The encoding of JSON data is called **Serialization**. Serialization is a technique where data transforms in the **series of bytes** and transmitted across the network.

The deserialization is the reverse process of decoding the data that is converted into the JSON format.

This module includes many built-in functions.

Let's have a look at these functions:

1. **import** json
2. print(dir(json))

**Output:**

['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '\_\_all\_\_', '\_\_author\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_path\_\_', '\_\_spec\_\_', '\_\_version\_\_', '\_default\_decoder', '\_default\_encoder', 'codecs', 'decoder', 'detect\_encoding', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']

In this section, we will learn the following methods:

* **load()**
* **loads()**
* **dump()**
* **dumps()**

## Serializing JSON

Serialization is the technique to convert the Python objects to JSON. Sometimes, computer need to process lots of information so it is good to store that information into the file. We can store JSON data into file using JSON function. The json module provides the **dump()** and **dumps()** method that are used to transform Python object.

Python objects are converted into the following JSON objects. The list is given below:

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Python Objects** | **JSON** |
| **1.** | Dict | Object |
| **2.** | list, tuple | Array |
| **3.** | Str | String |
| **4.** | int, float | Number |
| **5.** | True | true |
| **6.** | False | false |
| **7.** | None | null |

* **The dump() function**

**Writing JSON Data into File**

Python provides a **dump()** function to transmit(encode) data in JSON format. It accepts two positional arguments, first is the data object to be serialized and second is the file-like object to which the bytes needs to be written.

Let's consider the simple serialization example:

1. Import json
2. # Key:value mapping
3. student  = {
4. "Name" : "Peter",
5. "Roll\_no" : "0090014",
6. "Grade" : "A",
7. "Age": 20,
8. "Subject": ["Computer Graphics", "Discrete Mathematics", "Data Structure"]
9. }
11. with open("data.json","w") as write\_file:
12. json.dump(student,write\_file)

**Output:**

{"Name" : "Peter", "Roll\_no" : "0090014" , "Grade" : "A", "Age" : 20, "Subject" : ["Computer Graphics", "Discrete Mathematics", "Data Structure"] }

In the above program, we have opened a file named **data.json** in writing mode. We opened this file in write mode because if the file doesn't exist, it will be created. The **json.dump()** method transforms dictionary into JSON string.

* **The dumps () function**

The **dumps()** function is used to store serialized data in the Python file. It accepts only one argument that is Python data for serialization. The file-like argument is not used because we aren't not writing data to disk. Let's consider the following example:

1. **import** json
2. # Key:value mapping
3. student  = {
4. "Name" : "Peter",
5. "Roll\_no" : "0090014",
6. "Grade" : "A",
7. "Age": 20
8. }
9. b = json.dumps(student)
11. print(b)

**Output:**

{"Name": "Peter", "Roll\_no": "0090014", "Grade": "A", "Age": 20}

JSON supports primitive data types, such as strings and numbers, as well as nested list, tuples and objects.

1. **import** json
3. #Python  list conversion to JSON  Array
4. print(json.dumps(['Welcome', "to", "javaTpoint"]))
6. #Python  tuple conversion to JSON Array
7. print(json.dumps(("Welcome", "to", "javaTpoint")))
9. # Python string conversion to JSON String
10. print(json.dumps("Hello"))
12. # Python **int** conversion to JSON Number
13. print(json.dumps(1234))
15. # Python **float** conversion to JSON Number
16. print(json.dumps(23.572))
18. # Boolean conversion to their respective values
19. print(json.dumps(True))
20. print(json.dumps(False))
22. # None value to **null**
23. print(json.dumps(None))

**Output:**

["Welcome", "to", "javaTpoint"]

["Welcome", "to", "javaTpoint"]

"Hello"

1234

23.572

true

false

null

### Deserializing JSON

Deserialization is the process to decode the JSON data into the Python objects. The json module provides two methods **load()** and **loads()**, which are used to convert JSON data in actual Python object form. The list is given below:

|  |  |  |
| --- | --- | --- |
| **SR.** | **JSON** | **Python** |
| **1.** | Object | dict |
| **2.** | Array | list |
| **3.** | String | str |
| **4.** | number(int) | int |
| **5.** | true | True |
| **6.** | false | False |
| **7.** | null | None |

The above table shows the inverse of the serialized table but technically it is not a perfect conversion of the JSON data. It means that if we encode the object and decode it again after sometime; we may not get the same object back.

Let's take real-life example, one person translates something into Chinese and another person translates back into English, and that may not be exactly translated. Consider the simple example:

1. **import** json
2. a = (10,20,30,40,50,60,70)
3. print(type(a))
4. b = json.dumps(a)
5. print(type(json.loads(b)))

**Output:**

<class 'tuple'>

<class 'list'>

* **The load() function**

The **load()** function is used to deserialize the JSON data to Python object from the file. Consider the following example:

1. **import** json
2. # Key:value mapping
3. student  = {
4. "Name" : "Peter",
5. "Roll\_no" : "0090014",
6. "Grade" : "A",
7. "Age": 20,
8. }
10. with open("data.json","w") as write\_file:
11. json.dump(student,write\_file)
13. with open("data.json", "r") as read\_file:
14. b = json.load(read\_file)
15. print(b)

**Output:**

{'Name': 'Peter', 'Roll\_no': '0090014', 'Grade': 'A', 'Age': 20}

In the above program, we have encoded Python object in the file using **dump()** function. After that we read JSON file using **load()** function, where we have passed **read\_file** as an argument.

The json module also provides **loads()** function, which is used to convert JSON data to Python object. It is quite similar to the **load()** function. Consider the following example:

1. Import json
2. a = ["Mathew","Peter",(10,32.9,80),{"Name" : "Tokyo"}]
4. # Python object into JSON
5. b = json.dumps(a)
7. # JSON into Python Object
8. c = json.loads(b)
9. print(c)

**Output:**

['Mathew', 'Peter', [10, 32.9, 80], {'Name': 'Tokyo'}]

### json.load() vs json.loads()

The **json.load()** function is used to load JSON file, whereas **json.loads()** function is used to load string.

### json.dump() vs json.dumps()

The **json.dump()** function is used when we want to serialize the Python objects into JSON file and **json.dumps()** function is used to convert JSON data as a string for parsing and printing.

## Python Pretty Print JSON

Sometimes we need to analyze and debug a large amount of JSON data. It can be done by passing additional arguments indent and sort\_keys in json.dumps() and json.dump() methods.

#### Note: Both dump() and dumps() functions accept indent and short\_keys arguments.

Consider the following example:

1. **import** json
3. person = '{"Name": "Andrew","City":"English", "Number":90014, "Age": 23,"Subject": ["Data Structure","Computer Graphics", "Discrete mathematics"]}'
5. per\_dict = json.loads(person)
7. print(json.dumps(per\_dict, indent = 5, sort\_keys= True))

**Output:**

{

"Age": 23,

"City": "English",

"Name": "Andrew",

"Number": 90014,

"Subject": [

"Data Structure",

"Computer Graphics",

"Discrete mathematics"

]

}

In the above code, we have provided the 5 spaces to the indent argument and the keys are sorted in ascending order. The default value of indent is **None** and the default value of **sort\_key** is **False**.

## Encoding and Decoding

Encoding is the technique for transforming the text or values into an encrypted form. Encrypted data can only be used by the preferred user by decoding it. Encoding is also known as **serialization** and decoding is also called **deserialization**. Encoding and decoding are done for JSON(object) format. Python provides a popular package for such operations. We can install it on Windows by the following command:

1. pip install demjson

**Encoding** - The demjson package provides **encode()** function that is used to convert the Python object into a JSON string representation. The syntax is given below:

1. demjson.encode(self,obj,nest\_level = 0)

**Example:1 - Encoding using demjson package**

1. **import** demjson
2. a = [{"Name": 'Peter',"Age":20, "Subject":"Electronics"}]
3. print(demjson.encode(a))

**Output:**

[{"Age":20,"Name":"Peter","Subject":"Electronics"}]

**Decoding**-The **demjson** module provides **decode()** function, which is used to convert JSON object into Python format type. The syntax is given below:

1. Import demjson
2. a = "['Peter', 'Smith', 'Ricky', 'Hayden']"
3. print(demjson.decode(a))

**Output:**

['Peter', 'Smith', 'Ricky', 'Hayden']

In this tutorial, we have learned about the Python JSON. JSON is the most effective way to transmit data between the client and the web server.

# Python Itertools

Itertool is one of the most amazing Python 3 standard libraries. This library has pretty much coolest functions and nothing wrong to say that it is the gem of the Python programing language. Python provides excellent documentation of the itertools but in this tutorial, we will discuss few important and useful functions or iterators of itertools.

The key thing about itertools is that the functions of this library are used to make memory-efficient and precise code.

Before learning the Python itertools, you should have knowledge of the Python iterator and generators. In this article, we will describe itertools for beginners are well as for professionals.

## Introduction

According to the official definition of itertools, "**this module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML**." In simple words, the number of iterators can together create 'iterator algebra' which makes it possible to complete the complex task. The functions in itertools are used to produce more complex iterators. Let's take an example: [Python built-in zip() function](https://www.javatpoint.com/python-zip-function)

accepts any number of arguments as iterable. It iterates over tuples and return their corresponding elements.

1. a = [1,2,3]
2. b= ['a', 'b', 'c']
3. c = zip(a,b)
4. print(c)

**Output:**

[(1, 'a'), (2, 'b'), (3, 'c')]

In the above code, we have passed two lists [1,2,3] and ['a', 'b', 'c'] as iterable in **zip()**function**.**These lists return one element at a time. In [Python](https://www.javatpoint.com/python-tutorial)

, an element that implement **.\_\_iter\_\_()** or **.\_\_getitem\_\_()** method called iterable.

The [Python iter() function](https://www.javatpoint.com/python-iter-function)

is used to call on the iterable and return iterator object of the iterable.

1. a = iter('Hello')
2. print(a)

**Output:**

<str\_iterator object at 0x01505FA0>

The [Python zip() function](https://www.javatpoint.com/python-zip-function)

calls **iter()** on each of its argument and then calls **next()** by combining the result into tuple.

#### Note: If you are using the zip() function and map() function that means you are already using itertools. You don't need to import it distinctly.

## Types of Iterator

There are various types of iterator in itertools module. The list is given below:

* Infinite iterators
* Combinatoric iterators
* Terminating iterators

### Infinite Iterators

In Python, any object that can implement **for loop** is called iterators. Lists, tuples, set, dictionaries, strings are the example of iterators but iterator can also be infinite and this type of iterator is called **infinite iterator**.

|  |  |  |
| --- | --- | --- |
| **Iterator** | **Argument** | **Results** |
| count(start,step) | start, [step] | start, start+step, step+2\*step |
| cycle() | P | p0,p1,….plast |
| repeat() | elem [,n] | elem, elem, elem,….endlessly or upto n times |

* **count(start, stop)**: It prints from the start value to infinite. The step argument is optional, if the value is provided to the**step** then the number of steps will be skipped. Consider the following example:

1. **import** itertools
3. **for** i in itertools.count(10,5):
4. **if** i == 50:
5. **break**
6. **else**:
7. print(i,end=" ")

**Output:**

10 15 20 25 30 35 40 45

* **cycle(iterable)**: This iterator prints all value in sequence from the passed argument. It prints the values in a cyclic manner. Consider the following example:

1. **import** itertools
2. temp = 0
3. **for** i in itertools.cycle("123"):
4. **if** temp > 7:
5. **break**
6. **else**:
7. print(i,end=' ')
8. temp = temp+1

**Output:**

1 2 3 1 2 3 1 2 3 1 2

**Example - 2: Using next() function**

1. **import** itertools
3. val = ['Java', 'T', 'Point']
5. iter = itertools.cycle(val)
7. **for** i in range(6):
8. # Using next function
9. print(next(iter), end = " ")

**Output:**

Java T Point Java T Point

* **repeat(val,num)**: As the name suggests, it repeatedly prints the passed value for infinite time. The **num**argument is optional. Consider the following example:

1. **import** itertools
2. print("Printing the number repeadtly:")
3. print(list(itertools.repeat(40,15)))

**Output:**

[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40]

**Combinatoric iterators:** The complex combinatorial constructs are simplified by the recursive generators. The permutations, combinations, and Cartesian products are the example of the combinatoric construct.

In Python, there are four types of combinatoric iterators:

* **Product() -**It is used to calculate the cartesian product of input iterable. In this function, we use the optional **repeat** keyword argument for computation of the product of an iterable with itself. The **repeat** keyword represents the number of repetitions. It returns output in the form of sorted tuples. Consider the following example:

1. from itertools **import** product
3. print("We are computing cartesian product using repeat Keyword Argument:")
4. print(list(product([1, 2], repeat=2)))
5. print()
7. print("We are computing cartesian product of the containers:")
8. print(list(product(['Java', 'T', 'point'], '5')))
9. print()
11. print("We are computing product of the containers:")
12. print(list(product('CD', [4, 5])))

**Output:**

Computing cartesian product using repeat Keyword Argument:

[(1, 1), (1, 2), (2, 1), (2, 2)]

Computing cartesian product of the containers:

[('Java', '5'), ('T', '5'), ('point', '5')]

Computing product of the containers:

[('C', 4), ('C', 5), ('D', 4), ('D', 5)]

* **Permutations()**: It is used to generate all possible permutation of an iterable. The uniqueness of each element depends upon their position instead of values. It accepts two argument **iterable** and **group\_size**. If the value of group\_size is **none** or not specified then group\_size turns into length of the iterable.

1. from itertools **import** permutations
3. print("Computing all permutation of the following list")
4. print(list(permutations([3,"Python"],2)))
5. print()
7. print("Permutations of following string")
8. print(list(permutations('AB')))
9. print()
11. print("Permutation of the given container is:")
12. print(list(permutations(range(4),2)))

**Output:**

Computing all permutation of the following list

[(3, 'Python'), ('Python', 3)]

Permutations of following string

[('A', 'B'), ('B', 'A')]

Permutation of the given container is:

[(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]

* **Combinations()**: It is used to print all the possible combinations (without replacement) of the container which is passed as argument in the specified group size in sorted order.

1. from itertools **import** combinations
2. print("Combination of list in sorted order(without replacement)",list(combinations(['B',3],2)))
3. print()
5. print("Combination of string in sorted order",list(combinations("ZX",2)))
6. print()
8. print("Combination of list in sorted order",list(combinations(range(20),1)))

**Output:**

Combination of list in sorted order(without replacement) [('B', 3)]

Combination of string in sorted order [('Z', 'X')]

Combination of list in sorted order [(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]

* **Combination\_with\_replacement()**: It accepts two arguments, first argument is a r-length tuple and the second argument is repetition. It returns a subsequence of length n from the elements of the iterable and repeat the same process. Separate elements may repeat itself in **combination\_with\_replacement()**

1. from itertools **import** combinations\_with\_replacement
3. print("Combination of string in sorted order(with replacement) is:")
4. print(list(combinations\_with\_replacement("XY", 3)))
5. print()
7. print("Combination of list in sorted order(with replacement) is:")
8. print(list(combinations\_with\_replacement([4, 2], 3)))
9. print()
11. print("Combination of container in sorted order(with replacement) is:")
12. print(list(combinations\_with\_replacement(range(3), 2)))

**Output:**

Combination of string in sorted order(with replacement) is:

[('X', 'X', 'X'), ('X', 'X', 'Y'), ('X', 'Y', 'Y'), ('Y', 'Y', 'Y')]

Combination of list in sorted order(with replacement) is:

[(4, 4, 4), (4, 4, 2), (4, 2, 2), (2, 2, 2)]

Combination of container in sorted order(with replacement) is:

[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]

### Terminating Iterator

Terminating iterators are generally used to work on the small input sequence and generate the output based on the functionality of the method used in iterator.

There are different types of terminating iterator:

* **accumulate(iter, func)**: It takes two arguments, the first argument is iterable and the second is a function which would be followed at each iteration of value in iterable. If the function is not defined in **accumulate()** iterator, addition takes place by default. The output iterable depends on the input iterable; if input iterable contains no value then the output iterable will also be empty.

1. **import** itertools
2. **import** operator
4. # initializing list 1
5. list1 = [1, 4, 5, 7, 9, 11]
7. # using accumulate() that will prints the successive summation of elements
8. print("The sum is : ", end="")
9. print(list(itertools.accumulate(list1)))
11. # using accumulate() that will prints the successive multiplication of elements
12. print("The product is : ", end="")
13. print(list(itertools.accumulate(list1, operator.mul)))

16. # using accumulate() that will prints the successive summation of elements
17. print("The sum is : ", end="")
18. print(list(itertools.accumulate(list1)))
20. # using accumulate() that will prints the successive multiplication of elements
21. print("The product is : ", end="")
22. print(list(itertools.accumulate(list1, operator.mul)))

**Output:**

The sum is : [1, 5, 10, 17, 26, 37]

The product is : [1, 4, 20, 140, 1260, 13860]

The sum is : [1, 5, 10, 17, 26, 37]

The product is : [1, 4, 20, 140, 1260, 13860]

* **chain(iter1, iter2)** - It is used to print all the values in iterable passed in the form of chain and declared in arguments. Consider the following example:

1. **import** itertools
3. # declaring list 1
4. list1 = [1, 2, 3, 4]
6. # declaring list 2
7. list2 = [1, 5, 6, 8]
9. # declaring list 3
10. list3 = [9, 10, 11, 12]
12. # using chain() function that will to print all elements of lists
13. print("The output is : ", end="")
14. print(list(itertools.chain(list1, list2, list3)))

**Output:**

The output is: [1, 2, 3, 4, 1, 5, 6, 8, 9, 10, 11, 12]

* **dropwhile(func, seq)** - It starts printing the character only after the **func**. Consider the following argument:

1. **import** itertools
2. # initializing list
3. list1 = [2, 4, 5, 7, 8]
4. # using dropwhile() iterator that will print start displaying after condition is **false**
5. print("The output is : ", end="")
6. print(list(itertools.dropwhile(lambda x: x % 2 == 0, list1)))

**Output:**

The output is : [5, 7, 8]

* **filterfalse(func,seq)** - We can assume it by its name, as this iterator prints only those values that return false for the passed function. Consider the following example:

1. **import** itertools
3. # declaring list
4. list1 = [12, 14, 15, 27, 28]
6. # using filterfalse() iterator that will print **false** values
7. print("The Output is: ", end="")
8. print(list(itertools.filterfalse(lambda x: x % 2 == 0, list1)))

**Output:**

The Output is : [15, 27]

* **islice(iterable,start,stop,step)** - It slices the given iterable according to given position. It accepts four arguments respectively and these are iterable, container, starting pos., ending position and step(optional).

1. **import** itertools
2. # Declaring list
3. list1 = [12, 34, 65, 73, 80, 19, 20]
4. # using islice() iterator that will slice the list acc. to given argument
5. # starts printing from 3nd index till 8th skipping 2
6. print("The sliced list values are : ", end="")
7. print(list(itertools.islice(list1, 2, 8, 2)))

**Output:**

The sliced list values are : [34, 73, 19]

* **starmap(func, tuple list)** - It takes two arguments; first argument is function and second argument is list which consists element in the form of tuple. Consider the following example.

1. **import** itertools
3. # Declaring list that contain tuple as element
4. list1 = [(10, 20, 15), (18, 40, 19), (53, 42, 90), (16, 12, 27)]
6. # using starmap() iterator **for** selection value acc. to function
7. # selects max of all tuple values
8. print("The values acc. to function are : ", end="")
9. print(list(itertools.starmap(max, list1)))

**Output:**

The values acc. to function are : [20, 40, 90, 27]

* **takewhile(func, iterable)** - It is visa-versa of **dropwhile().** It will print values until it returns false condition. Consider the following example:

1. **import** itertools
3. # Defining a list
4. list1 = [20, 42, 64, 77, 8, 10, 20]
6. # takewhile() iterator is used  to print values till condition **return** **false**.
7. print("Print until 1st false value returned : ", end="")
8. print(list(itertools.takewhile(lambda x: x % 2 == 0, list1)))

**Output:**

The list values until false value return : [20, 42, 64]

* **tee(iterator, count)** - It divides the container into a number of iterators which is defined in the argument. Consider the following example:

1. **import** itertools
3. # Declaring list
4. li = [1, 2, 3, 4, 5, 6, 7]
6. # storing list in iterator
7. iti = iter(li)
8. # using tee() iterator to create a list of iterators
9. # Creating list of 3 iterators having similar values.
10. it = itertools.tee(iti, 3)
11. # It will print object of iterator
12. print(it)
13. print("The iterators are : ")
14. **for** i in range(0, 2):
15. print(list(it[i]))

**Output:**

(<itertools.\_tee object at 0x01B88D88>, <itertools.\_tee object at 0x01B88DA8>, <itertools.\_tee object at 0x01B88BA8>)

The iterators are :

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

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

* **zip\_longest(iterable1, iterable2, fillval)** - It prints the values of iterable alternatively in sequence. If one of the iterable prints all values, remaining values are filled by the values assigned to fill value.

1. **import** itertools
2. print(" The combined value of iterrables is :")
3. print(\*(itertools.zip\_longest('Java', 'Tpoint', fillvalue='\_')))

**Output:**

The combined value of iterables is :

('J', 'T') ('a', 'p') ('v', 'o') ('a', 'i') ('\_', 'n') ('\_', 't')

In this tutorial, we have discussed several useful iterators along with itertools.

# Python Multiprocessing

In this article, we will learn how we can achieve multiprocessing using Python. We also discuss its advanced concepts.

### What is Multiprocessing?

Multiprocessing is the ability of the system to run one or more processes in parallel. In simple words, multiprocessing uses the two or more [CPU](https://www.javatpoint.com/cpu-full-form) within the single computer system. This method is also capable to allocate the tasks between more than one process.

Processing units share the main memory and peripherals to process programs simultaneously. Multiprocessing Application breaks into smaller parts and runs independently. Each process is allocated to the processor by the operating system.

[Python](https://www.javatpoint.com/python-tutorial) provides the built-in package called multiprocessing which supports swapping processes. Before working with the multiprocessing, we must aware with the process object.

### Why Multiprocessing?

Multiprocessing is essential to perform the multiple tasks within the Computer system. Suppose a computer without multiprocessing or single processor. We assign various processes to that system at the same time.

It will then have to interrupt the previous task and move to another to keep all processes going. It is as simple as a chef is working alone in the kitchen. He has to do several tasks to cook food such as cutting, cleaning, cooking, kneading dough, baking, etc.

Therefore, multiprocessing is essential to perform several task at the same time without interruption. It also makes easy to track all the tasks. That is why the concept of multiprocessing is to arise.

* Multiprocessing can be represented as a computer with more than one central processor.
* A Multi-core processor refers to single computing component with two or more independent units.

In the multiprocessing, the [CPU](https://www.javatpoint.com/central-processing-unit) can assign multiple tasks at one each task has its own processor.

## Multiprocessing In Python

Python provides the multiprocessing module to perform multiple tasks within the single system. It offers a user-friendly and intuitive API to work with the multiprocessing.

Let's understand the simple example of multiple processing.

Example -

1. from multiprocessing import Process
2. def disp():
3. print ('Hello !! Welcome to Python Tutorial')
4. if \_\_name\_\_ == '\_\_main\_\_':
5. p = Process(target=disp)
6. p.start()
7. p.join()

**Output:**

'Hello !! Welcome to Python Tutorial'

**Explanation:**

In the above code, we have imported the Process class then create the Process object within the **disp()** function. Then we started the process using the **start()** method and completed the process with the **join()** method. We can also pass the arguments in the declared function using the **args** keywords.

Let's understand the following example of the multiprocessing with arguments.

**Example - 2**

1. # Python multiprocessing example
2. # importing the multiprocessing module
4. import multiprocessing
5. def cube(n):
6. # This function will print the cube of the given number
7. print("The Cube is: {}".format(n \* n \* n))
9. def square(n):
10. # This function will print the square of the given number
11. print("The Square is: {}".format(n \* n))
13. if \_\_name\_\_ == "\_\_main\_\_":
14. # creating two processes
15. process1 = multiprocessing.Process(target= square, args=(5, ))
16. process2 = multiprocessing.Process(target= cube, args=(5, ))
18. # Here we start the process 1
19. process1.start()
20. # Here we start process 2
21. process2.start()
23. # The join() method is used to wait for process 1 to complete
24. process1.join()
25. # It is used to wait for process 1 to complete
26. process2.join()
28. # Print if both processes are completed
29. print("Both processes are finished")

**Output:**

The Cube is: 125

The Square is: 25

Both processes are finished

**Explanation -**

In the above example, We created the two functions - the **cube()** function calculates the given number's cube, and the **square()** function calculates the square of the given number.

Next, we defined the process object of the Process class that has two arguments. The first argument is a **target** that represents the function to be executed, and the second argument is **args that** represents the argument to be passed within the function.

1. process1 = multiprocessing.Process(target= square, args=(5, ))
2. process2 = multiprocessing.Process(target= cube, args=(5, ))

We have used the **start()** method to start the process.

1. process1.start()
2. process2.start()

As we can see in the output, it waits to completion of **process one** and then **process 2**. The last statement is executed after both processes are finished.

## Python Multiprocessing Classes

Python multiprocessing module provides many classes which are commonly used for building parallel program. We will discuss its main classes - Process, Queue and Lock. We have already discussed the Process class in the previous example. Now we will discuss the Queue and Lock classes.

Let's see the simple example of a get number of CPUs currently in the system.

**Example -**

1. import multiprocessing
2. print("The number of CPU currently working in system : ", multiprocessing.cpu\_count())

**Output:**

('The number of CPU currently woking in system : ', 32)

The above number of CPUs can vary for your pc. For us, the number of cores is 32.

## Python Multiprocessing Using Queue Class

We know that Queue is important part of the data structure. Python multiprocessing is precisely the same as the data structure queue, which based on the "First-In-First-Out" concept. Queue generally stores the Python object and plays an essential role in sharing data between processes.

Queues are passed as a parameter in the Process' target function to allow the process to consume data. The Queue provides the **put()** function to insert the data and **get()** function to get data from the queues. Let's understand the following example.

**Example -**

1. # Importing Queue Class
3. from multiprocessing import Queue
5. fruits = ['Apple', 'Orange', 'Guava', 'Papaya', 'Banana']
6. count = 1
7. # creating a queue object
8. queue = Queue()
9. print('pushing items to the queue:')
10. for fr in fruits:
11. print('item no: ', count, ' ', fr)
12. queue.put(fr)
13. count += 1
15. print('\npopping items from the queue:')
16. count = 0
17. while not queue.empty():
18. print('item no: ', count, ' ', queue.get())
19. count += 1

**Output:**

pushing items to the queue:

('item no: ', 1, ' ', 'Apple')

('item no: ', 2, ' ', 'Orange')

('item no: ', 3, ' ', 'Guava')

('item no: ', 4, ' ', 'Papaya')

('item no: ', 5, ' ', 'Banana')

popping items from the queue:

('item no: ', 0, ' ', 'Apple')

('item no: ', 1, ' ', 'Orange')

('item no: ', 2, ' ', 'Guava')

('item no: ', 3, ' ', 'Papaya')

('item no: ', 4, ' ', 'Banana')

**Explanation -**

In the above code, we have imported the **Queue** class and initialized the list named fruits. Next, we assigned a **count** to 1. The count variable will count the total number of elements. Then, we created the queue object by calling the **Queue()** method. This object will used to perform operations in the Queue. In for loop, we inserted the elements one by one in the queue using the **put()** function and increased the count by 1 with each iteration of loop.

## Python Multiprocessing Lock Class

The multiprocessing Lock class is used to acquire a lock on the process so that we can hold the other process to execute a similar code until the lock has been released. The Lock class performs mainly two tasks. The first is to acquire a lock using the **acquire()** function and the second is to release the lock using the **release()** function.

## Python Multiprocessing Example

Suppose we have multiple tasks. So, we create two queues: the first queue will maintain the tasks, and the other will store the complete task log. The next step is to instantiate the processes to complete the task. As discussed previously, the Queue class is already synchronized, so we don't need to acquire a lock using the Lock class.

In the following example, we will merge all the multiprocessing classes together. Let's see the below example.

**Example -**

1. from multiprocessing import Lock, Process, Queue, current\_process
2. import time
3. import queue

6. def jobTodo(tasks\_to\_perform, complete\_tasks):
7. while True:
8. try:
10. # The try block to catch task from the queue.
11. # The get\_nowait() function is used to
12. # raise queue.Empty exception if the queue is empty.
14. task = tasks\_to\_perform.get\_nowait()
16. except queue.Empty:
18. break
19. else:
21. # if no exception has been raised, the else block will execute
22. # add the task completion

25. print(task)
26. complete\_tasks.put(task + ' is done by ' + current\_process().name)
27. time.sleep(.5)
28. return True

31. def main():
32. total\_task = 8
33. total\_number\_of\_processes = 3
34. tasks\_to\_perform = Queue()
35. complete\_tasks = Queue()
36. number\_of\_processes = []
38. for i in range(total\_task):
39. tasks\_to\_perform.put("Task no " + str(i))
41. # defining number of processes
42. for w in range(total\_number\_of\_processes):
43. p = Process(target=jobTodo, args=(tasks\_to\_perform, complete\_tasks))
44. number\_of\_processes.append(p)
45. p.start()
47. # completing process
48. for p in number\_of\_processes:
49. p.join()
51. # print the output
52. while not complete\_tasks.empty():
53. print(complete\_tasks.get())
55. return True

58. if \_\_name\_\_ == '\_\_main\_\_':
59. main()

**Output:**

Task no 2

Task no 5

Task no 0

Task no 3

Task no 6

Task no 1

Task no 4

Task no 7

Task no 0 is done by Process-1

Task no 1 is done by Process-3

Task no 2 is done by Process-2

Task no 3 is done by Process-1

Task no 4 is done by Process-3

Task no 5 is done by Process-2

Task no 6 is done by Process-1

Task no 7 is done by Process-3

## Python Multiprocessing Pool

Python multiprocessing pool is essential for parallel execution of a function across multiple input values. It is also used to distribute the input data across processes **(data parallelism)**. Consider the following example of a multiprocessing Pool.

**Example -**

1. from multiprocessing import Pool
2. import time
4. w = (["V", 5], ["X", 2], ["Y", 1], ["Z", 3])

7. def work\_log(data\_for\_work):
8. print(" Process name is %s waiting time is %s seconds" % (data\_for\_work[0], data\_for\_work[1]))
9. time.sleep(int(data\_for\_work[1]))
10. print(" Process %s Executed." % data\_for\_work[0])

13. def handler():
14. p = Pool(2)
15. p.map(work\_log, w)
17. if \_\_name\_\_ == '\_\_main\_\_':
18. handler()

**Output:**

Process name is V waiting time is 5 seconds

Process V Executed.

Process name is X waiting time is 2 seconds

Process X Executed.

Process name is Y waiting time is 1 seconds

Process Y Executed.

Process name is Z waiting time is 3 seconds

Process Z Executed.

Let's understand another example of the multiprocessing Pool.

**Example - 2**

1. from multiprocessing import Pool
2. def fun(x):
3. return x\*x
5. if \_\_name\_\_ == '\_\_main\_\_':
6. with Pool(5) as p:
7. print(p.map(fun, [1, 2, 3]))

**Output:**

[1, 8, 27]

## Proxy Objects

The proxy objects are referred to as shared objects which reside in a different process. This object is also called as a proxy. Multiple proxy objects might have a similar referent. A proxy object consists of various methods which are used to invoked corresponding methods of its referent. Below is the example of proxy objects.

**Example -**

1. from multiprocessing import Manager
2. manager = Manager()
3. l = manager.list([i\*i for i in range(10)])
4. print(l)
5. print(repr(l))
6. print(l[4])
7. print(l[2:5])

**Output:**

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

<ListProxy object, typeid 'list' at 0x7f063621ea10>

16

[4, 9, 16]

The proxy objects are picklable so we can pass them between processes. These objects are also used for level of control over the synchronization.

## Commonly Used Functions of Multiprocessing

So far, we have discussed the basic concepts of multiprocessing using Python. Multiprocessing is a broad topic itself and essential for performing various tasks within a single system. We are defining a few essential functions that are commonly used to achieve multiprocessing.

|  |  |
| --- | --- |
| **Method** | **Description** |
| pipe() | The pipe() function returns a pair of connection objects. |
| run() | The run() method is used to represent the process activities. |
| start() | The start()method is used to start the process. |
| join([timeout]) | The join() method is used to block the process until the process whose join() method is called terminates. The timeout is optional argument. |
| is\_alive() | It returns if process is alive. |
| terminate() | As the name suggests, it is used to terminate the process. Always remember - the **terminate()** method is used in Linux, for Windows, we use **TerminateProcess()** method. |
| kill() | This method is similar to the **terminate()** but using the SIGKILL signal on Unix. |
| close() | This method is used to close the **Process** object and releases all resources associated with it. |
| qsize() | It returns the approximate size of the queue. |
| empty() | If queue is empty, it returns **True**. |
| full() | It returns **True**, if queue is full. |
| get\_await() | This method is equivalent **get(False)**. |
| get() | This method is used to get elements from the queue. It removes and returns an element from queue. |
| put() | This method is used to insert an element into the queue. |
| cpu\_count() | It returns the number of working CPU within the system. |
| current\_process() | It returns the Process object corresponding to the current process. |
| parent\_process() | It returns the parent Process object corresponding to the current process. |
| task\_done() | This function is used indicate that an enqueued task is completed. |
| join\_thread() | This method is used to join the background thread |