

Notes on Python

Python is a widely used programming language that offers several unique features and advantages compared to languages like **Java** and **C++**.

In the late 1980s, **Guido van Rossum** dreamed of developing Python. The first version of **Python 0.9.0 was released in 1991**. Since its release, Python started gaining popularity. According to reports, Python is now the most popular programming language among developers because of its high demands in the tech realm.

What is Python

Python is a general-purpose, dynamically typed, high-level, compiled and interpreted, garbage-collected, and purely object-oriented programming language that supports procedural, object-oriented, and functional programming.

Python Features

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

1) Easy to Learn and Use

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

2) Expressive Language

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

3) Interpreted Language

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

4) Cross-platform Language

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

5) Free and Open Source

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

6) Object-Oriented Language

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

7) Extensible

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

8) Large Standard Library

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

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQt5, Tkinter, Kivy are the libraries which are used for developing the web application

10) Integrated

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

11. Embeddable

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

12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time.

Suppose we are assigned integer value 15 to **x**, then we don't need to write **int x = 15**. Just write **x = 15**.

Python History and Versions

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

Why the Name Python?

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

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

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

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

Python Version List

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

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

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

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-tac-toe game, an alarm clock app, a to-do list, student or customer management system, etc.

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

8. Teach Others

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

9. Explore Libraries and Frameworks

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

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

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

10. Contribute to Open Source

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

Usage of Python

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

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

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

How to Install Python

Python is a popular high-level, general-use programming language. Python is a programming language that enables rapid development as well as more effective system integration. Python has two main different versions: Python 2 and Python 3. Both are really different.

Python develops new versions with changes periodically and releases them according to version numbers. Python is currently at **version 3.11.3**.

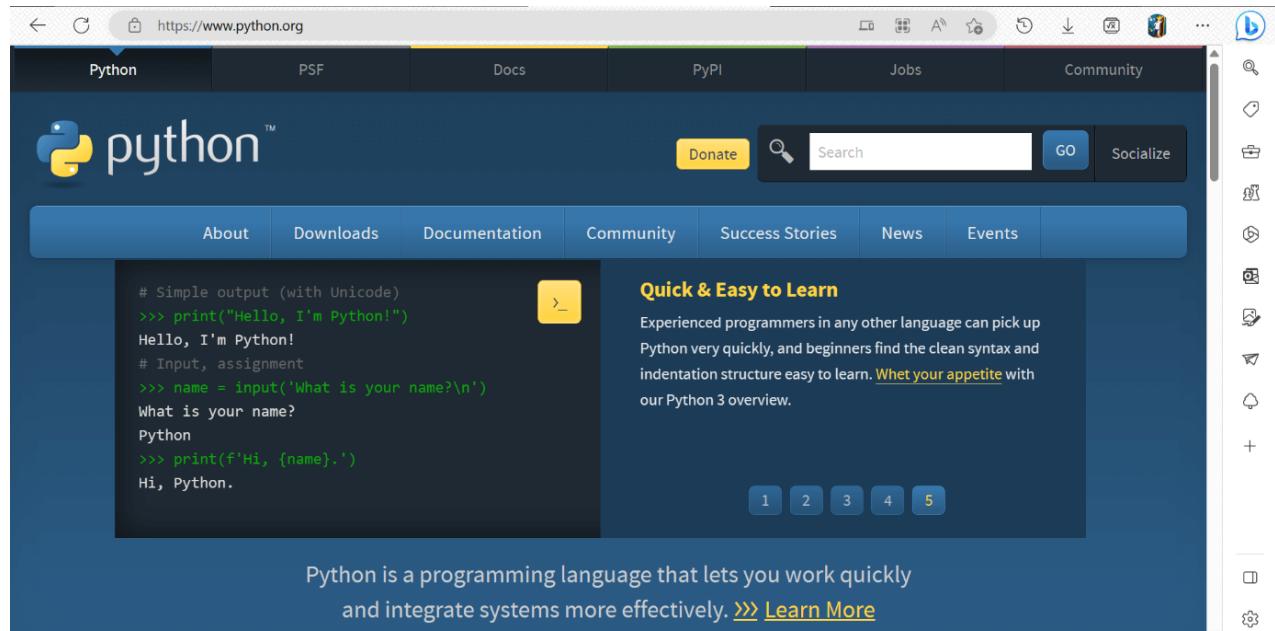
Python is much simpler to learn and programme in. Any plain text editor, such as notepad or notepad++, may be used to create Python programs. To make it easier to create these routines, one may also utilise an online IDE for Python or even install one on their machine. IDEs offer a variety of tools including a user-friendly code editor, the debugger, compiler, etc.

One has to have Python installed on their system in order to start creating Python code and carrying out many fascinating and helpful procedures. The first step in learning how to programming in Python is to install or update Python on your computer. There are several ways to install Python: you may use a package manager, get official versions from Python.org, or install specialised versions for embedded devices, scientific computing, and the Internet of Things.

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

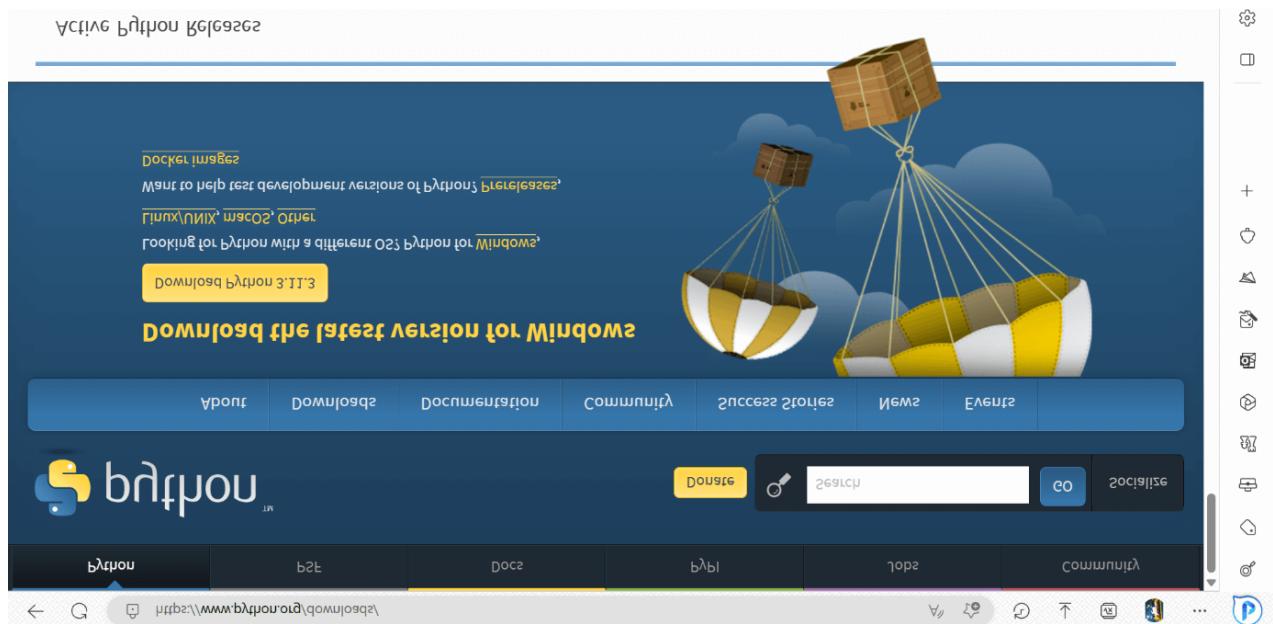
Installation on Windows

Visit the link <https://www.python.org> to download the latest release of Python. In this process, we will install Python **3.11.3** on our Windows operating system. When we click on the above link, it will bring us the following page.



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

Click on the download button to download the exe file of Python.



If in case you want to download the specific version of Python. Then, you can scroll down further below to see different versions from 2 and 3 respectively. Click on download button right next to the version number you want to download.

A screenshot of the Python releases page. The URL in the address bar is https://www.python.org/downloads/. The page title is "Looking for a specific release?". It shows a table of Python releases by version number. The table has columns for "Release version", "Release date", and "Click for more". Each row contains a link to "Download" and "Release Notes". The table includes versions from 3.10.11 down to 2.7.18. A "View older releases" link is at the bottom of the table. On the left, there's a "Sponsors" section with a note about visionary sponsors helping to host Python downloads.

Step - 2: Click on the Install Now

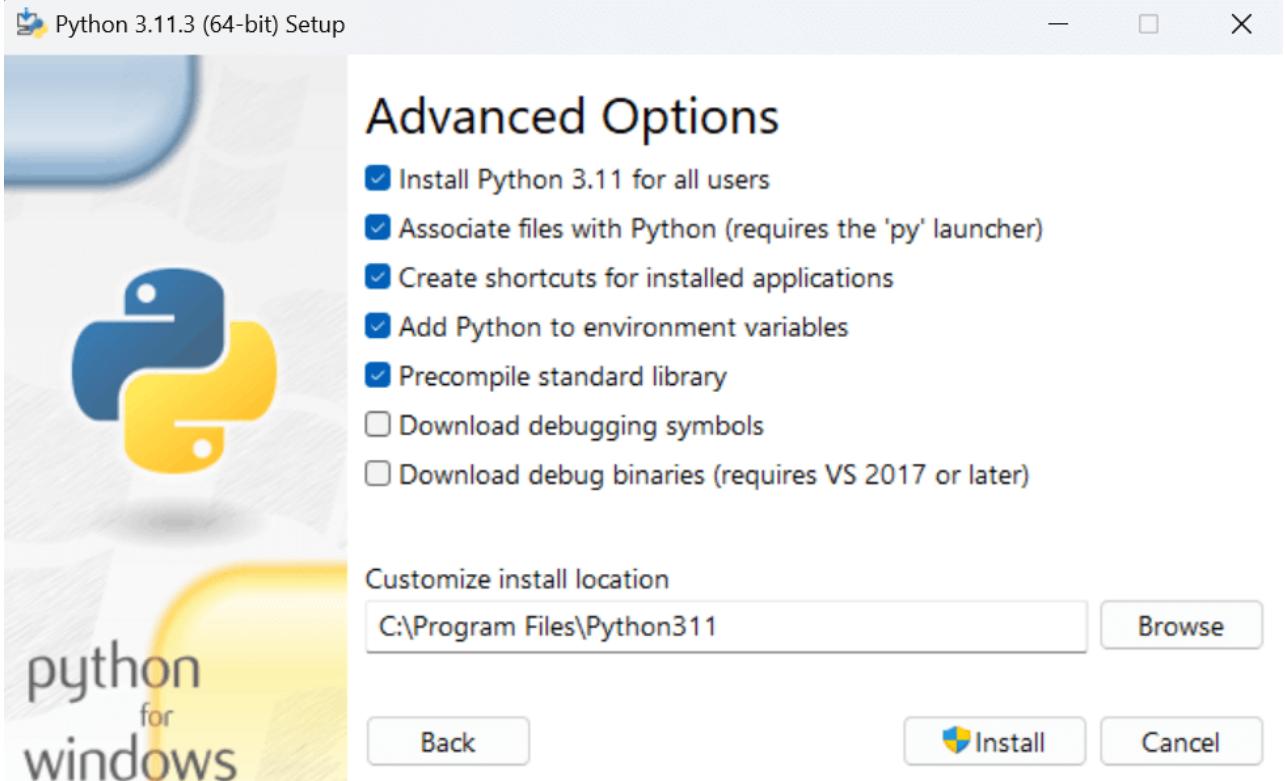
Double-click the executable file, which is downloaded.



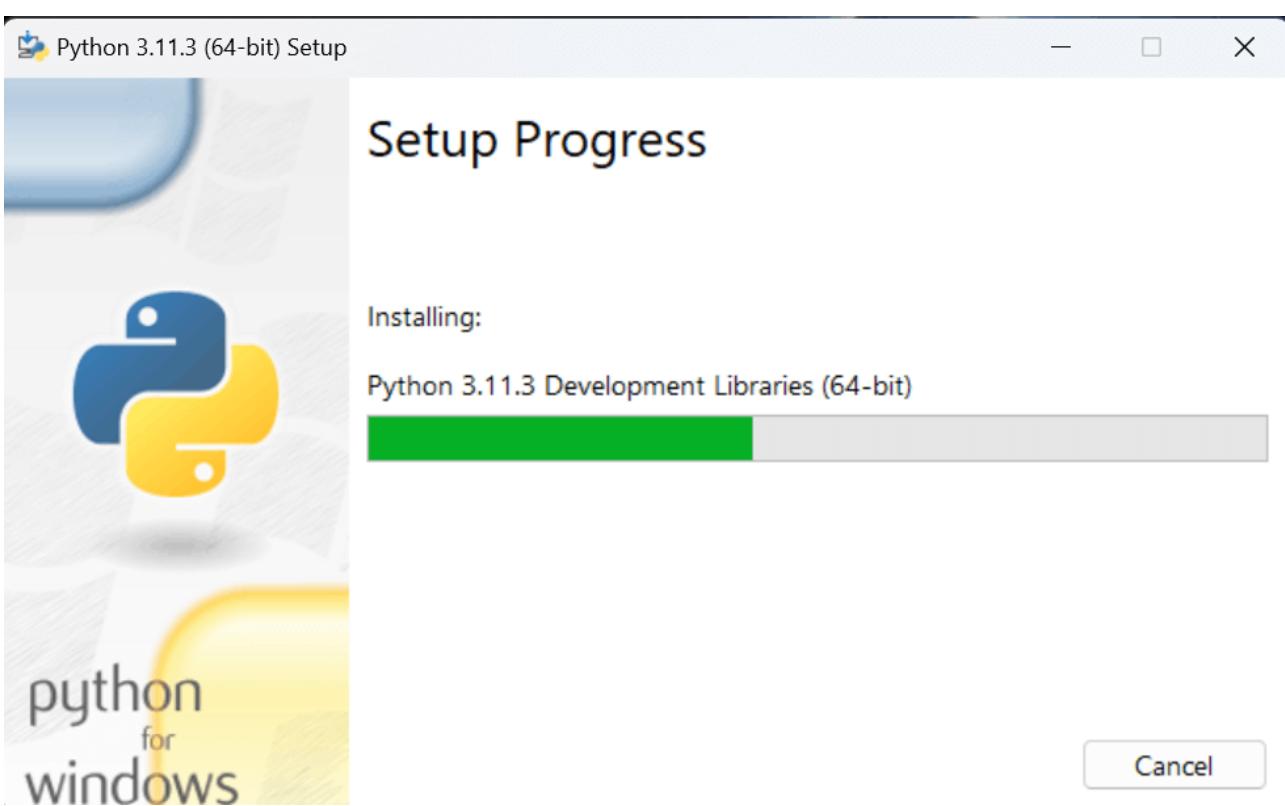
The following window will open. Click on the Add Path check box, it will set the Python path automatically.

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

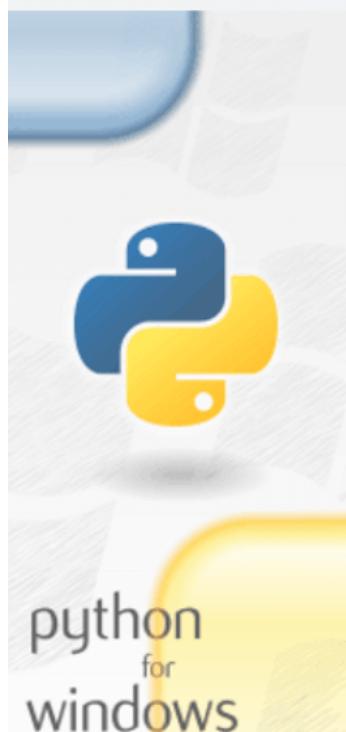
Here, under the advanced options, click on the checkboxes of " Install Python 3.11 for all users ", which is previously not checked in. This will checks the other option " Precompile standard library " automatically. And the location of the installation will also be changed. We can change it later, so we leave the install location default. Then, click on the install button to finally install.



Step - 3 Installation in Process



The set up is in progress. All the python libraries, packages, and other python default files will be installed in our system. Once the installation is successful, the following page will appear saying " Setup was successful ".



Setup was successful

New to Python? Start with the [online tutorial](#) and [documentation](#). At your terminal, type "py" to launch Python, or search for Python in your Start menu.

See [what's new](#) in this release, or find more info about [using Python on Windows](#).

Disable path length limit

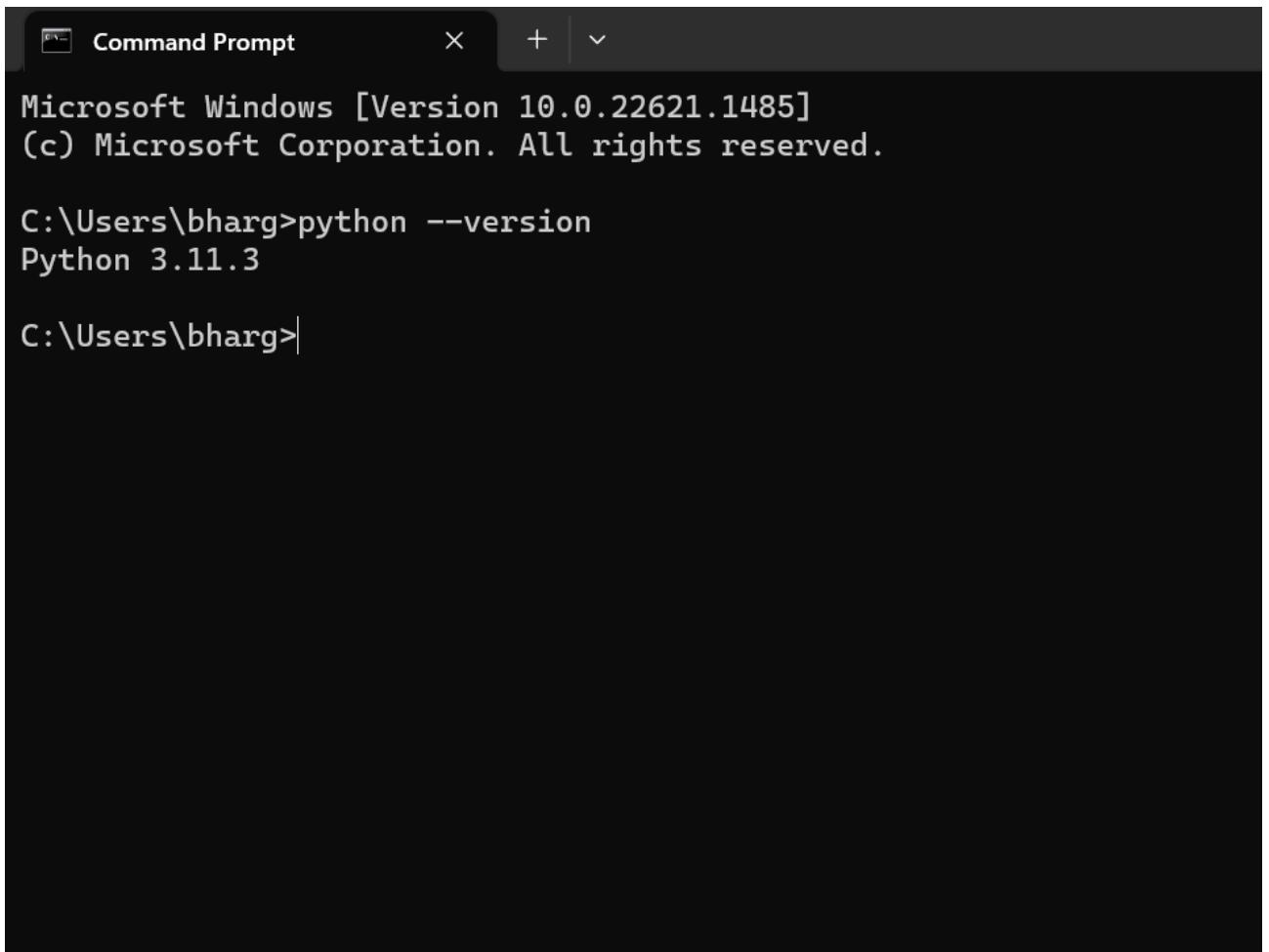
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

[Close](#)

Step - 4: Verifying the Python Installation

To verify whether the python is installed or not in our system, we have to do the following.

- Go to "Start" button, and search " cmd ".
- Then type, "**python --version**".
- If python is successfully installed, then we can see the version of the python installed.
- If not installed, then it will print the error as "**'python' is not recognized as an internal or external command, operable program or batch file.**".



A screenshot of a Microsoft Windows Command Prompt window. The title bar says "Command Prompt". The window content shows the following text:

```
Microsoft Windows [Version 10.0.22621.1485]
(c) Microsoft Corporation. All rights reserved.

C:\Users\bharg>python --version
Python 3.11.3

C:\Users\bharg>
```

We are ready to work with the Python.

Step - 5: Opening idle

Now, to work on our first python program, we will go to the interactive interpreter prompt(idle). To open this, go to "Start" and type idle. Then, click on open to start working on idle.



The screenshot shows the IDLE Shell 3.11.3 window. The title bar reads "IDLE Shell 3.11.3". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python 3.11.3 interpreter output:

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

First Python Program

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

Python provides us the two ways to run a program:

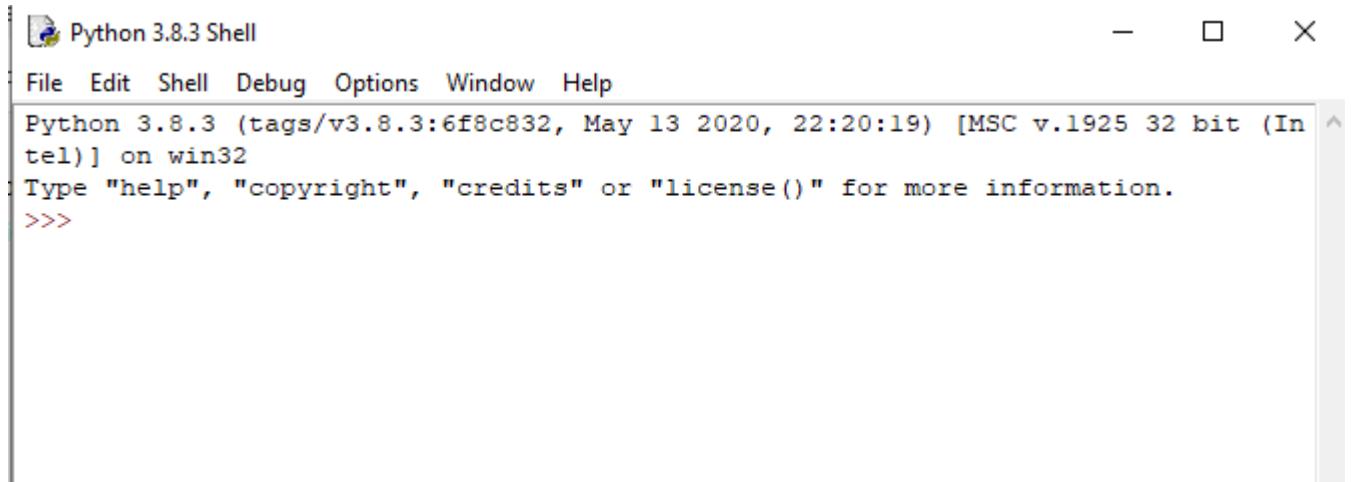
- Using Interactive interpreter prompt
- Using a script file

Interactive interpreter prompt

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

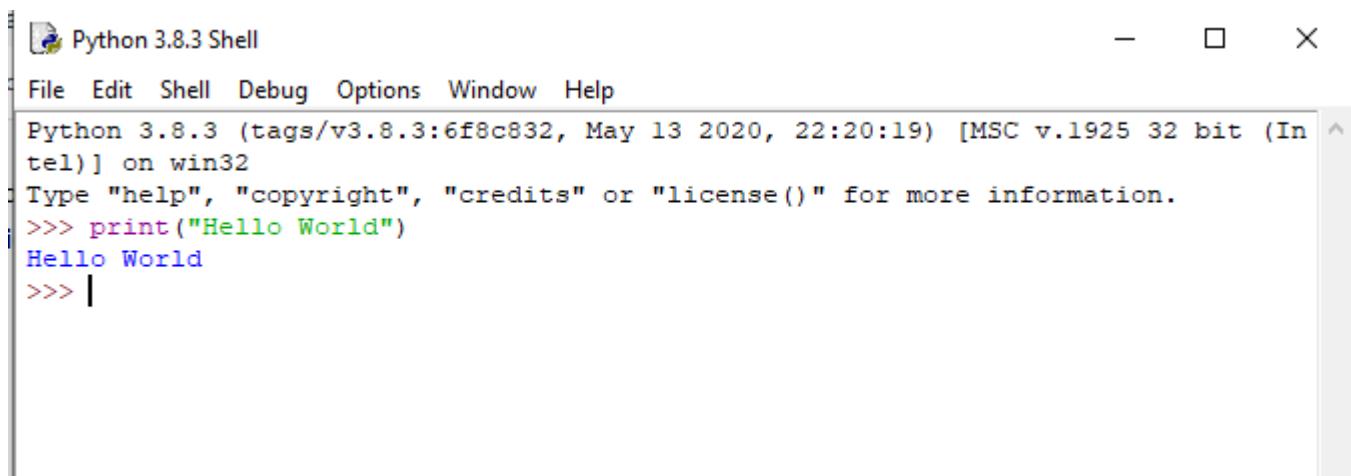
To open the interactive mode, open the terminal (or command prompt) and type python (python3 in case if you have Python2 and Python3 both installed on your system).

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



A screenshot of the Python 3.8.3 Shell window. The title bar says "Python 3.8.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the Python interpreter's welcome message: "Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32" and "Type "help", "copyright", "credits" or "license()" for more information.". Below this, the prompt ">>>" is visible, indicating the user can enter code.

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



A screenshot of the Python 3.8.3 Shell window. The title bar says "Python 3.8.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area shows the interpreter's message and the user's input: ">>> print("Hello World")". The output "Hello World" is displayed below the input. A cursor is shown at the end of the input line.

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

Using a script file (Script Mode Programming)

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

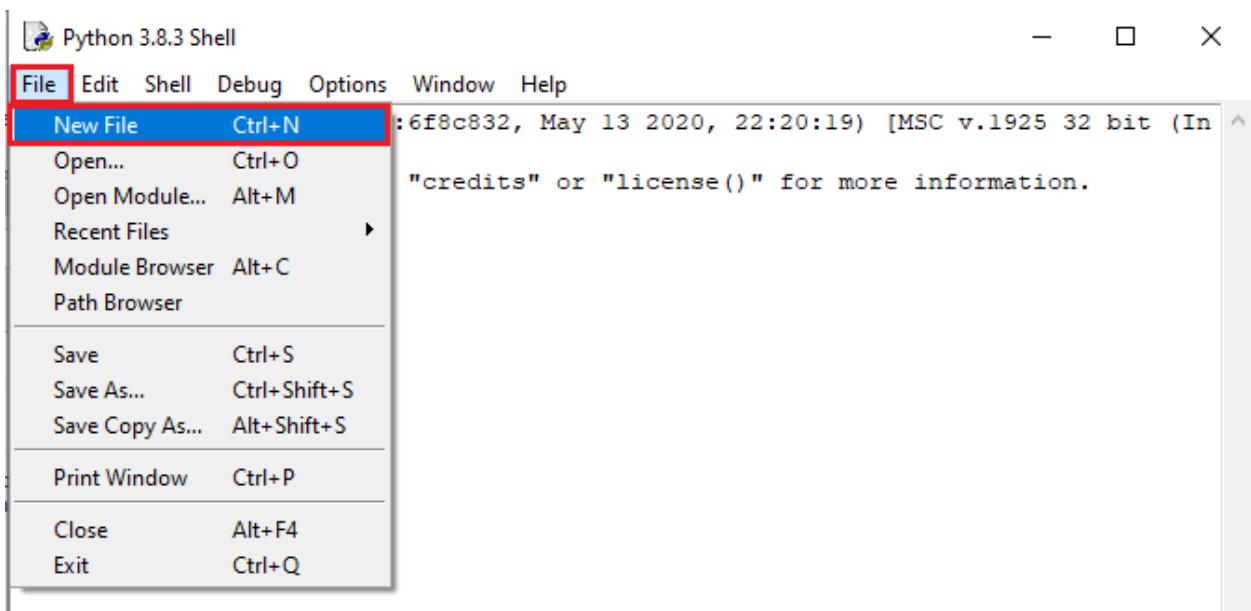
Using the script mode, we can write multiple lines code into a file which can be executed later. For this purpose, we need to open an editor like notepad, create a file named and save it with **.py** extension, which stands for "**Python**". Now, we will implement the above example using the script mode.

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

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

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (In tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

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

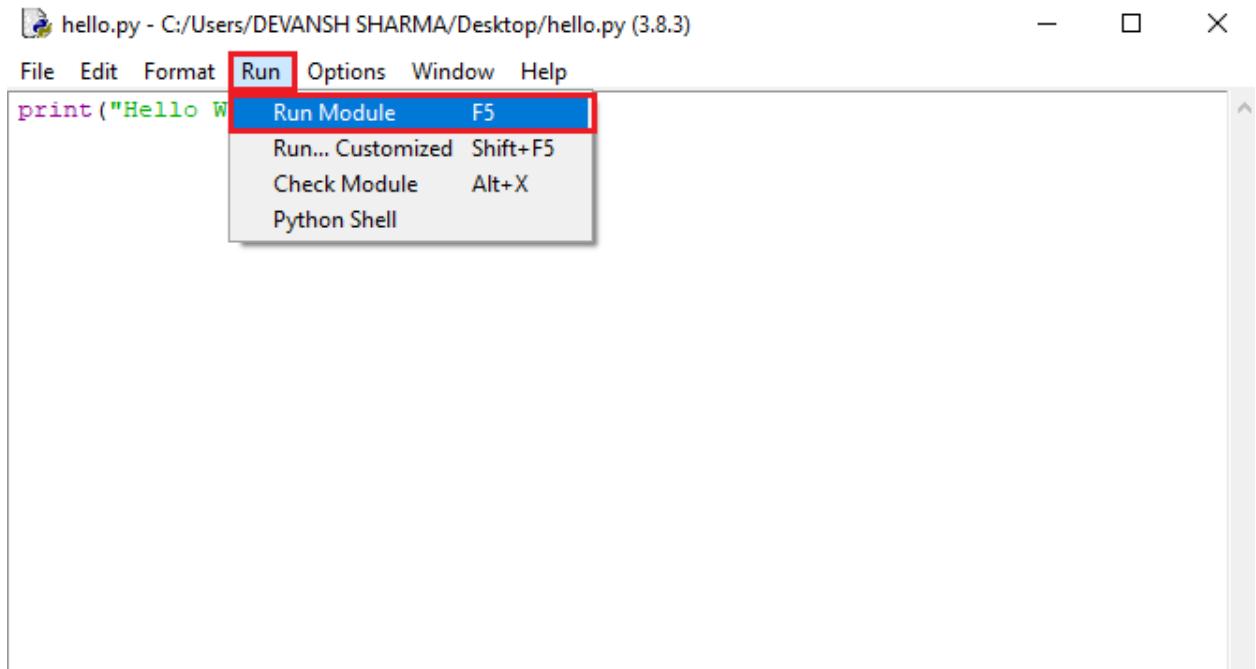


Step - 2: Now, write the code and press "**Ctrl+S**" to save the file.

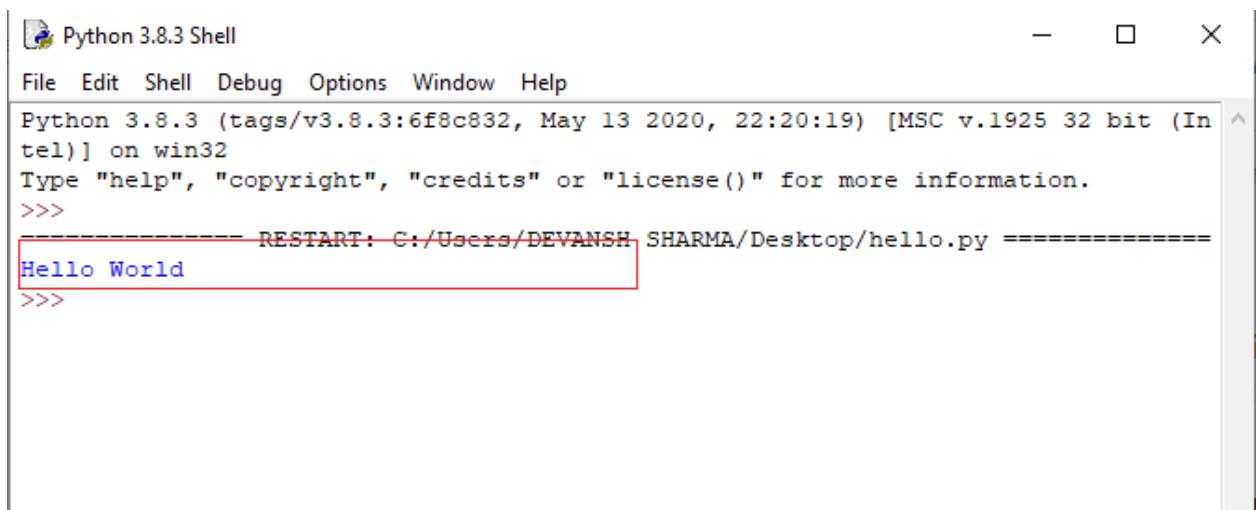
```
*untitled*
File Edit Format Run Options Window Help
print("Hello World")
```

ADVERTISEMENT

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

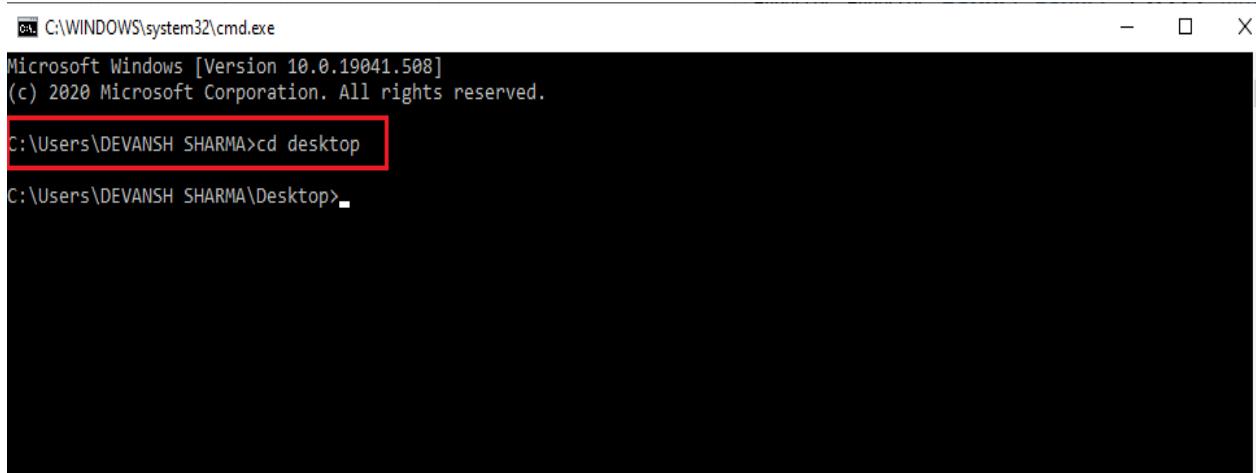


The output will be shown as follows.



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

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



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



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the command 'python hello.py' being run, which outputs 'Hello World'. The output line is highlighted with a red rectangle.

```
C:\Users\DEVANSH SHARMA\Desktop>python hello.py
Hello World
C:\Users\DEVANSH SHARMA\Desktop>
```

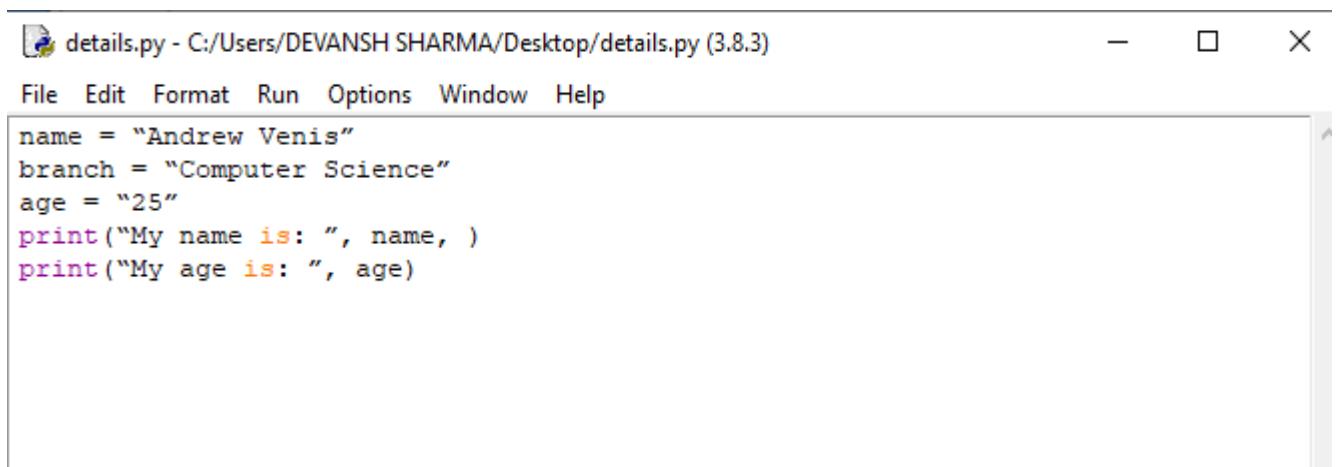
Multi-line Statements

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

Code:

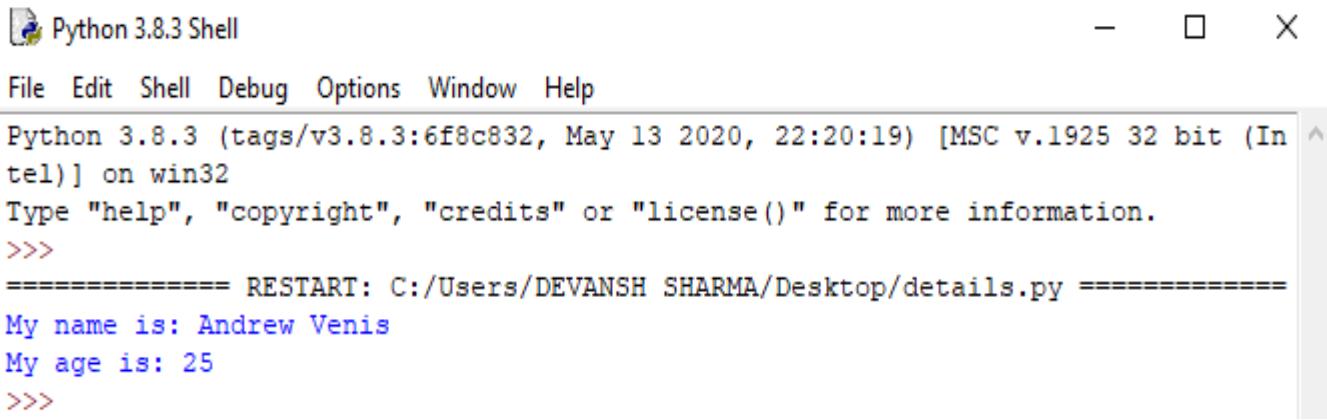
1. name = "Andrew Venis"
2. branch = "Computer Science"
3. age = "25"
4. **print("My name is: ", name,)**
5. **print("My age is: ", age)**

Script File:



A screenshot of a Notepad window titled 'details.py - C:/Users/DEVANSH SHARMA/Desktop/details.py (3.8.3)'. The window displays a Python script with variables and print statements.

```
File Edit Format Run Options Window Help
name = "Andrew Venis"
branch = "Computer Science"
age = "25"
print("My name is: ", name, )
print("My age is: ", age)
```



The screenshot shows a Python 3.8.3 Shell window. The title bar says "Python 3.8.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the Python interpreter's prompt and some sample code. The code runs a script named "details.py" which outputs "My name is: Andrew Venis" and "My age is: 25".

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (In tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:/Users/DEVANSH SHARMA/Desktop/details.py =====
My name is: Andrew Venis
My age is: 25
>>>
```

Pros and Cons of Script Mode

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

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

Let's see the disadvantages of the script mode.

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

Get Started with PyCharm

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

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

ADVERTISEMENT

PyCharm installation

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

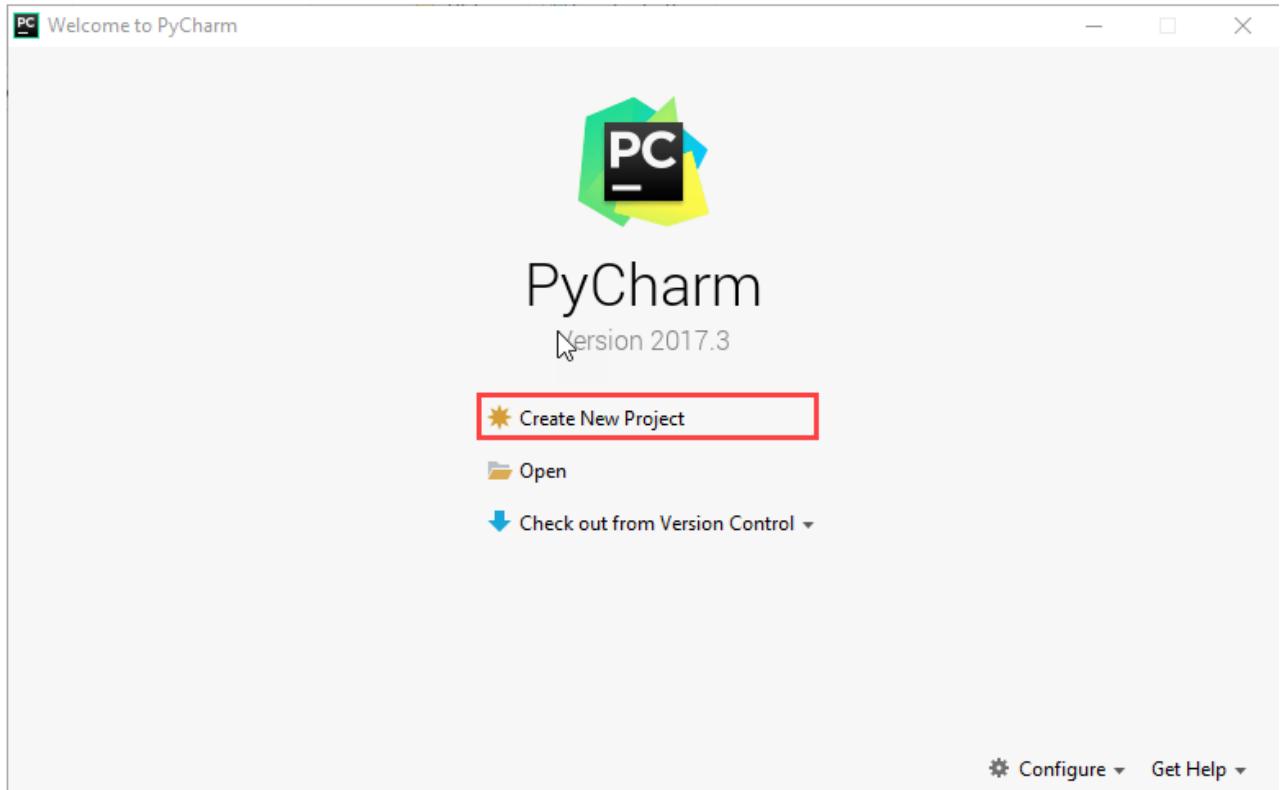
Windows

Installing PyCharm on Windows is very simple. To install PyCharm on Windows operating system, visit the link <https://www.jetbrains.com/pycharm/download/>

[thanks.html?platform=windows](#) to download the executable installer. **Double click** the installer (.exe) file and install PyCharm by clicking next at each step.

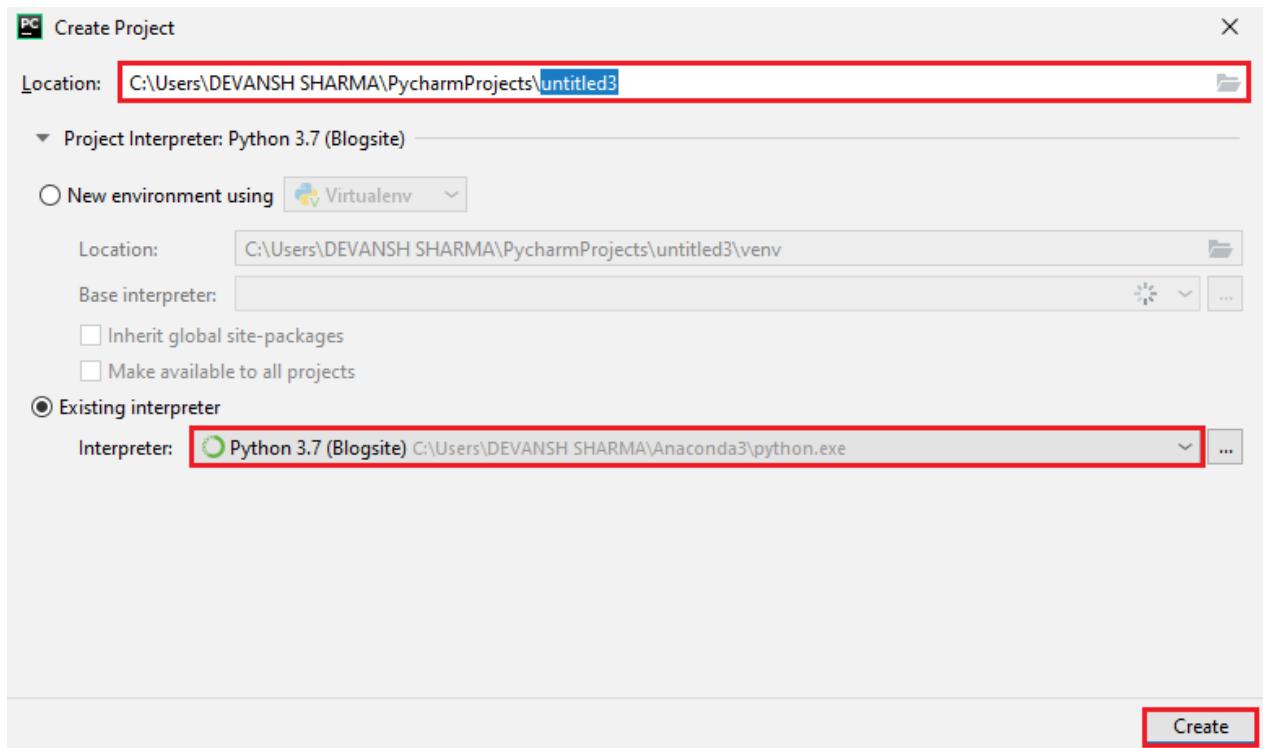
To create a first program to Pycharm follows the following step.

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

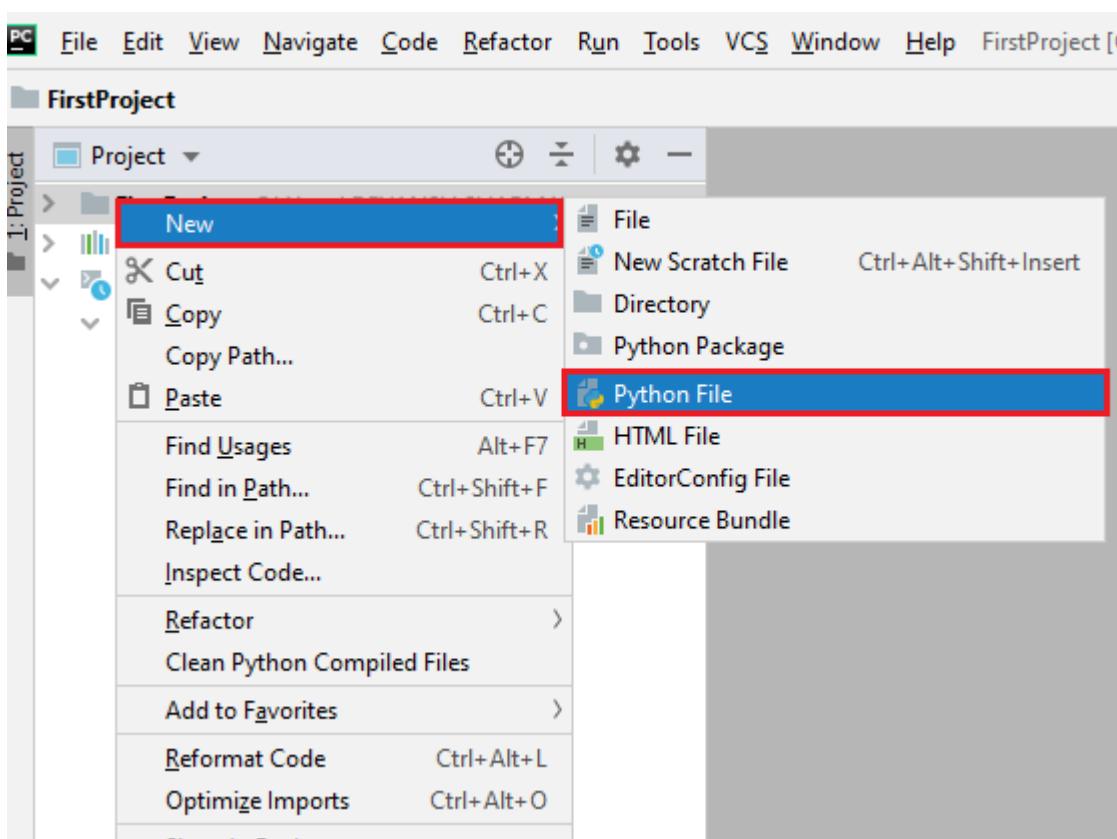


Step - 2. Select a location to save the project.

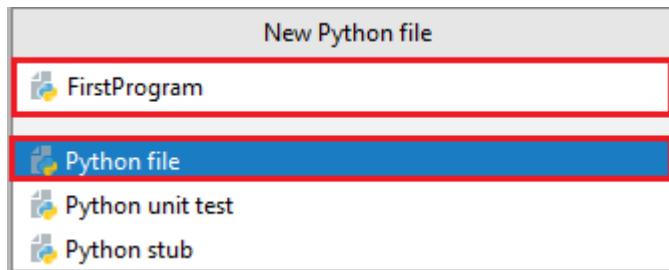
- a. We can save the newly created project at desired memory location or can keep file location as it is but atleast change the project default name **untitled** to "**FirstProject**" or something meaningful.
- b. Pycharm automatically found the installed Python interpreter.
- c. After change the name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



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



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



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

The screenshot shows the PyCharm code editor with a single file named 'FirstProgram.py'. The code contains a single line: `print("Hello World")`. The line is highlighted in yellow, and there is a small yellow lightbulb icon next to it, indicating a code suggestion or warning.

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

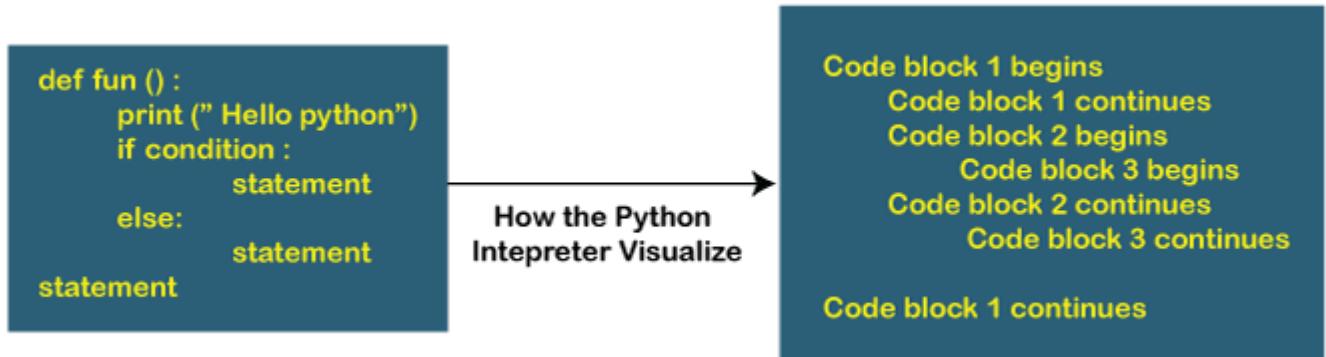
The screenshot shows the PyCharm 'Run' tab. It displays the output of the 'FirstProgram' run. The output window shows the command used: `"C:\Users\DEVANSH SHARMA\Anaconda3\python.exe" "C:/Users/DEVANSH SHARMA/PycharmProjects/FirstProject/FirstProgram.py"`, followed by the printed output: `Hello World`, and finally the message: `Process finished with exit code 0`.

Basic Syntax of Python

Indentation and Comment in Python

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

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



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

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

Example -

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

Output:

```
1
2
3
4
End of for loop
```

Explanation:

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

Comments in Python

Comments are essential for defining the code and help us and other to understand the code. By looking the comment, we can easily understand the intention of every line that

we have written in code. We can also find the error very easily, fix them, and use in other applications.

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

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

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

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

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

Types of Comment

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

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

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

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

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

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

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

We can also use another way.

'''

This is an example
Of multi-line comment
Using triple-quotes
'''

Python Identifiers

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

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

Let's understand the following example.

Example -

1. number = 10
2. **print**(num)
- 3.
4. _a = 100
5. **print**(_a)
- 6.
7. x_y = 1000
8. **print**(x_y)

Output:

```
10
100
1000
```

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

Python Variables

A variable is the name given to a memory location. A value-holding Python variable is also known as an identifier.

Since Python is an infer language that is smart enough to determine the type of a variable, we do not need to specify its type in Python.

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

The name of the variable should be written in lowercase. Both Rahul and rahul are distinct variables.

Identifier Naming

Identifiers are things like variables. An Identifier is utilized to recognize the literals utilized in the program. The standards to name an identifier are given underneath.

- The variable's first character must be an underscore or alphabet (_).
- Every one of the characters with the exception of the main person might be a letter set of lower-case(a-z), capitalized (A-Z), highlight, or digit (0-9).
- White space and special characters (!, @, #, %, etc.) are not allowed in the identifier name.
^, &, *).
- Identifier name should not be like any watchword characterized in the language.
- Names of identifiers are case-sensitive; for instance, my name, and MyName isn't something very similar.
- 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 doesn't tie us to pronounce a variable prior to involving it in the application. It permits us to make a variable at the necessary time.
- In Python, we don't have to explicitly declare variables. The variable is declared automatically whenever a value is added to it.
- The equal (=) operator is utilized to assign worth to a variable.

Object References

When we declare a variable, it is necessary to comprehend how the Python interpreter works. Compared to a lot of other programming languages, the procedure for dealing with variables is a little different.

Python is the exceptionally object-arranged programming language; Because of this, every data item is a part of a particular class. Think about the accompanying model.

1. `print("John")`

Output:

John

The Python object makes a integer object and shows it to the control center. We have created a string object in the print statement above. Make use of the built-in type() function in Python to determine its type.

1. `type("John")`

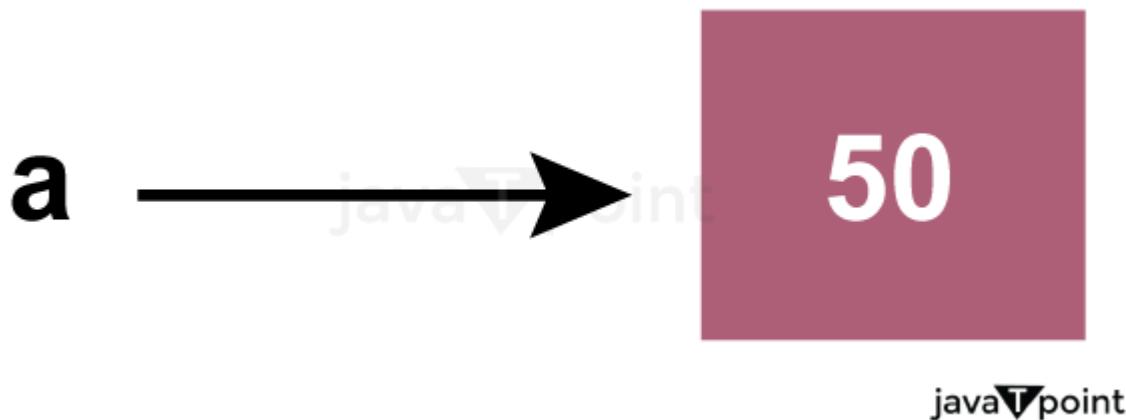
Output:

```
<class 'str'>
```

In Python, factors are symbolic names that are references or pointers to an item. The factors are utilized to indicate objects by that name.

Let's understand the following example

1. `a = 50`



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

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

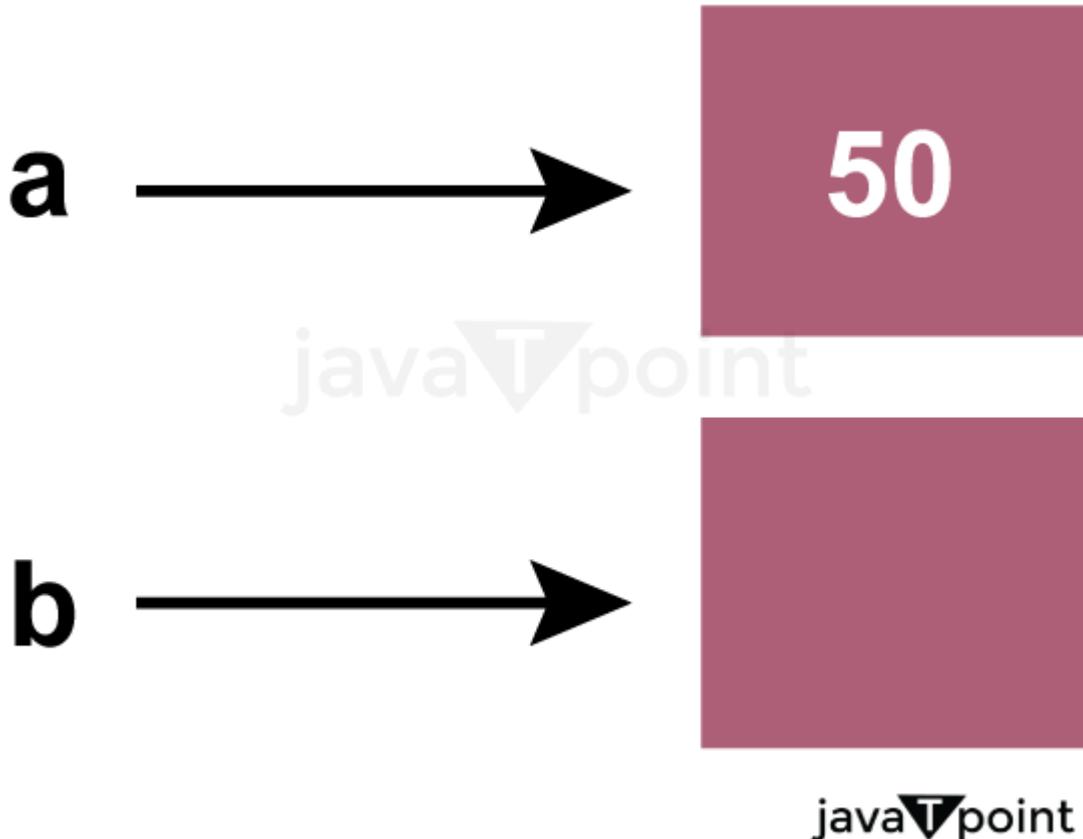
1. `a = 50`
2. `b = a`



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

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

1. `a = 50`
2. `b = 100`



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

Object Identity

Every object created in Python has a unique identifier. Python gives the dependable that no two items will have a similar identifier. The object identifier is identified using the built-in `id()` function. consider about the accompanying model.

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 highlight a similar item. The `id()` function that we used to check returned the same number. We reassign `a` to 500; The new object identifier was then mentioned.

Variable Names

The process for declaring the valid variable has already been discussed. Variable names can be any length can have capitalized, lowercase (start to finish, a to z), the digit (0-9), and highlight character(_). Take a look at the names of valid variables in the following example.

1. `name = "Devansh"`
2. `age = 20`
3. `marks = 80.50`
- 4.
5. `print(name)`
6. `print(age)`
7. `print(marks)`

Output:

```
Devansh  
20  
80.5
```

Consider the following valid variables name.

1. `name = "A"`
2. `Name = "B"`
3. `naMe = "C"`
4. `NAME = "D"`
5. `n_a_m_e = "E"`
6. `_name = "F"`
7. `name_ = "G"`
8. `_name_ = "H"`
9. `na56me = "I"`
- 10.
11. `print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)`

Output:

```
A B C D E D E F G F I
```

We have declared a few valid variable names in the preceding example, such as `name`, `_name_`, and so on. However, this is not recommended because it may cause confusion

when we attempt to read code. To make the code easier to read, the name of the variable ought to be descriptive.

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

Multiple assignments, also known as assigning values to multiple variables in a single statement, is a feature of Python.

We can apply different tasks in two ways, either by relegating a solitary worth to various factors or doling out numerous qualities to different factors. Take a look at 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

The variables that are declared within the function and have scope within the function are known as local variables. Let's examine the following illustration.

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.
9. `# Calling a function`
10. `add()`

Output:

```
The sum is: 50
```

Explanation:

We declared the function `add()` and assigned a few variables to it in the code above. These factors will be alluded to as the neighborhood factors which have scope just inside the capability. We get the error that follows if we attempt to use them outside of the function.

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 utilized all through the program, and its extension is in the whole program. Global variables can be used inside or outside the function.

By default, a variable declared outside of the function serves as the global variable. Python gives the worldwide catchphrase to utilize worldwide variable inside the capability. The function treats it as a local variable if we don't use the global keyword. Let's examine the following illustration.

Example -

```
1. # Declare a variable and initialize it
2. x = 101
3.
4. # Global variable in function
5. def mainFunction():
6.     # printing a global variable
7.     global x
8.     print(x)
9.     # modifying a global variable
10.    x = 'Welcome To Javatpoint'
11.    print(x)
12.
13. mainFunction()
14. print(x)
```

Output:

```
101
Welcome To Javatpoint
Welcome To Javatpoint
```

Explanation:

In the above code, we declare a global variable x and give out a value to it. We then created a function and used the global keyword to access the declared variable within the function. We can now alter its value. After that, we gave the variable x a new string value and then called the function and printed x, which displayed the new value.

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

Python, to the other programming languages, does not support long int or float data types. It uses the int data type to handle all integer values. The query arises here. In Python, what is the maximum value that the variable can hold? Take a look at the following example.

Example -

Output:

As we can find in the above model, we assigned a large whole number worth to variable x and really look at its sort. It printed class `<int>` not long int. As a result, the number of bits is not limited, and we are free to use all of our memory.

There is no special data type for storing larger numbers in Python.

Print Single and Numerous Factors in Python

We can print numerous factors inside the single print explanation. The examples of single and multiple printing values are provided below.

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:

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

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

Python Data Types

Every value has a datatype, and variables can hold values. Python is a powerfully composed language; consequently, we don't have to characterize the sort of variable while announcing it. The interpreter binds the value implicitly to its type.

1. `a = 5`

We did not specify the type of the variable `a`, which has the value five from an integer. The Python interpreter will automatically interpret the variable as an integer.

We can verify the type of the program-used variable thanks to Python. The `type()` function in Python returns the type of the passed variable.

Consider the following illustration when defining and verifying the values of various data types.

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'>
```

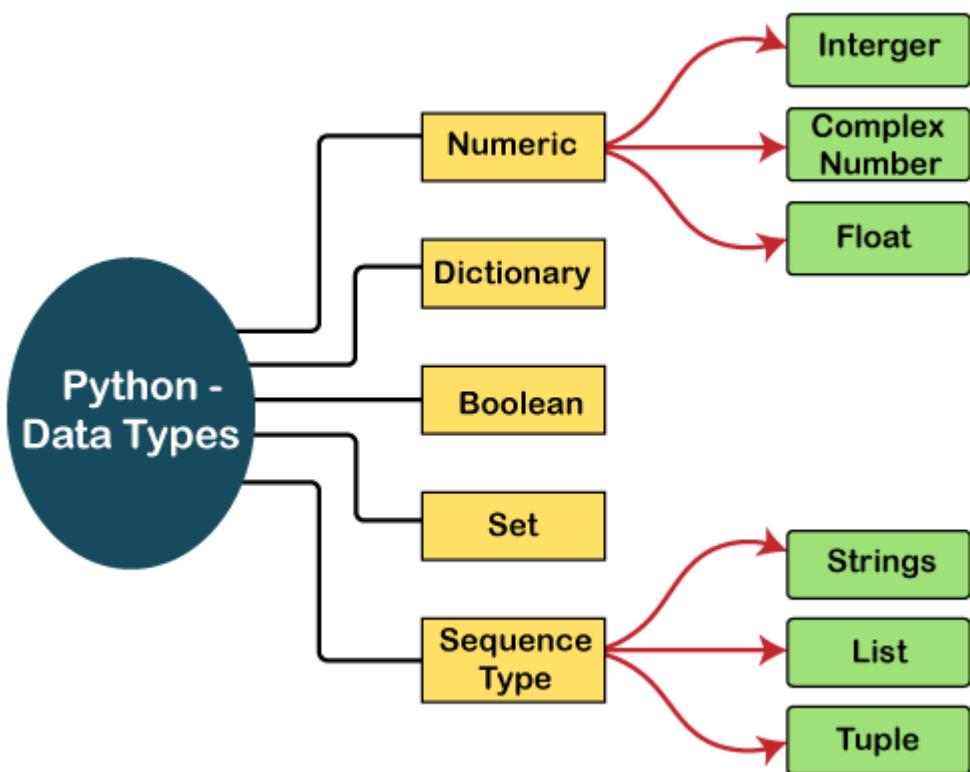
Standard data types

A variable can contain a variety of values. On the other hand, a person's id must be stored as an integer, while their name must be stored as a string.

The storage method for each of the standard data types that Python provides is specified by Python. The following is a list of the Python-defined data types.

1. [Numbers](#)
2. [Sequence Type](#)
3. [Boolean](#)

4. [Set](#)
5. [Dictionary](#)



The data types will be briefly discussed in this tutorial section. We will talk about every single one of them exhaustively later in this instructional exercise.

Numbers

Numeric values are stored in numbers. The whole number, float, and complex qualities have a place with a Python Numbers datatype. Python offers the `type()` function to determine a variable's data type. The `instance()` capability is utilized to check whether an item has a place with a specific class.

When a number is assigned to a variable, Python generates Number objects. For instance,

1. `a = 5`
2. `print("The type of a", type(a))`
- 3.
4. `b = 40.5`
5. `print("The type of b", type(b))`
- 6.
7. `c = 1+3j`
8. `print("The type of c", type(c))`
9. `print(" c is a complex number", isinstance(1+3j,complex))`

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

Python supports three kinds of numerical data.

- **Int:** Whole number worth can be any length, like numbers 10, 2, 29, - 20, - 150, and so on. An integer can be any length you want in Python. Its worth has a place with int.
- **Float:** Float stores drifting point numbers like 1.9, 9.902, 15.2, etc. It can be accurate to within 15 decimal places.
- **Complex:** An intricate number contains an arranged pair, i.e., $x + iy$, where x and y signify the genuine and non-existent parts separately. The complex numbers like $2.14j$, $2.0 + 2.3j$, etc.

Sequence Type

String

The sequence of characters in the quotation marks can be used to describe the string. A string can be defined in Python using single, double, or triple quotes.

String dealing with Python is a direct undertaking since Python gives worked-in capabilities and administrators to perform tasks in the string.

When dealing with strings, the operation "hello" + " python" returns "hello python," and the operator + is used to combine two strings.

Because the operation "Python" *2 returns "Python," the operator * is referred to as a repetition operator.

The Python string is demonstrated in the following example.

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
```

Look at the following illustration 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

Lists in Python are like arrays in C, but lists can contain data of different types. The things put away in the rundown are isolated with a comma (,) and encased inside square sections [].

To gain access to the list's data, we can use slice [:] operators. Like how they worked with strings, the list is handled by the concatenation operator (+) and the repetition operator (*).

Look at the following example.

Example:

1. list1 = [1, "hi", "Python", 2]
2. #Checking type of given list
3. **print**(type(list1))
- 4.
5. #Printing the list1
6. **print** (list1)
- 7.
8. # List slicing
9. **print** (list1[3:])
- 10.
11. # List slicing
12. **print** (list1[0:2])
- 13.
14. # List Concatenation using + operator
15. **print** (list1 + list1)
- 16.
17. # List repetition using * operator
18. **print** (list1 * 3)

Output:

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

Tuple

In many ways, a tuple is like a list. Tuples, like lists, also contain a collection of items from various data types. A parenthetical space () separates the tuple's components from one another.

Because we cannot alter the size or value of the items in a tuple, it is a read-only data structure.

Let's look at a straightforward tuple in action.

Example:

1. `tup = ("hi", "Python", 2)`
2. `# Checking type of tup`
3. `print(type(tup))`
- 4.
5. `#Printing the tuple`
6. `print(tup)`
- 7.
8. `# Tuple slicing`
9. `print(tup[1:])`
10. `print(tup[0:1])`
- 11.
12. `# Tuple concatenation using + operator`
13. `print(tup + tup)`
- 14.
15. `# Tuple repatation using * operator`
16. `print(tup * 3)`
- 17.
18. `# Adding value to tup. It will throw an error.`
19. `t[2] = "hi"`

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)
```

```
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is a key-value pair set arranged in any order. It stores a specific value for each key, like an associative array or a hash table. Value is any Python object, while the key can hold any primitive data type.

The comma (,) and the curly braces are used to separate the items in the dictionary.

Look at the following example.

1. `d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}`
- 2.
3. `# Printing dictionary`
4. `print (d)`
- 5.
6. `# Accesing value using keys`
7. `print("1st name is "+d[1])`
8. `print("2nd name is "+ d[4])`
- 9.
10. `print (d.keys())`
11. `print (d.values())`

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

True and False are the two default values for the Boolean type. These qualities are utilized to decide the given assertion valid or misleading. The class book indicates this. False can be represented by the 0 or the letter "F," while true can be represented by any value that is not zero.

Look at 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

The data type's unordered collection is Python Set. It is iterable, mutable(can change after creation), and has remarkable components. The elements of a set have no set order; It might return the element's altered sequence. Either a sequence of elements is passed through the curly braces and separated by a comma to create the set or the built-in function set() is used to create the set. It can contain different kinds of values.

Look at the following example.

```
1. # Creating Empty set
2. set1 = set()
3.
4. set2 = {'James', 2, 3,'Python'}
5.
6. #Printing Set value
7. print(set2)
8.
9. # Adding element to the set
10.
11.set2.add(10)
12. print(set2)
13.
14. #Removing element from the set
15.set2.remove(2)
16. print(set2)
```

Output:

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

Python Keywords

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

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

Introducing Python Keywords

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

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

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

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

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

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

Code

1. # Python program to demonstrate the application of iskeyword()
2. # importing keyword library which has lists
3. **import** keyword
- 4.
5. # displaying the complete list using "kwlist()."
6. **print**("The set of keywords in this version is: ")
7. **print**(keyword.kwlist)

Output:

```
The set of keywords in this version is :
```

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

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

Code

1. `help("keywords")`

How to Identify Python Keywords

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

Write Code on a Syntax Highlighting IDE

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

Verify Keywords with Script in a REPL

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

Look for a SyntaxError

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

Python Keywords and Their Usage

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

- o 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.
6. `number = no_return_function()`
7. `print(number)`

Output:

None

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

Code

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

Output:

None

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

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

| Mathematical Operations | Operations in Other Languages | Python Keyword |
|-------------------------|-------------------------------|----------------|
| AND, \wedge | $\&\&$ | and |
| OR, \vee | \parallel | or |
| NOT, \neg | ! | not |
| CONTAINS, \in | | 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.
10. `the_outer_function()`

Output:

```
The value inside the inner function: 14  
The value inside the outer function: 14
```

`the_inner_function()` is placed inside `the_outer_function` in this case.

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

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

Code

```
1. def the_outer_function():
2.     var = 10
3.     def the_inner_function():
4.         var = 14
5.         print("Value inside the inner function: ", var)
6.     the_inner_function()
7.     print("Value inside the outer function: ", var)
8.
9. the_outer_function()
```

Output:

```
Value inside the inner function:  14
Value inside the outer function:  10
```

Iteration Keywords: for, while, break, continue

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

The for Keyword

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

The while Keyword

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

The break Keyword

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

The continue Keyword

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

Code

```
1. # Program to show the use of keywords for, while, break, continue
2. for i in range(15):
3.
4.     print(i + 4, end = " ")
5.
6.     # breaking the loop when i = 9
7.     if i == 9:
8.         break
9.     print()
10.
11. # looping from 1 to 15
12. i = 0 # initial condition
13. while i < 15:
14.
15.     # When i has value 9, loop will jump to next iteration using continue. It will not print
16.     if i == 9:
17.         i += 3
18.         continue
19.     else:
20.         # when i is not equal to 9, adding 2 and printing the value
21.         print(i + 2, end = " ")
22.
23.     i += 1
```

Output:

```
4 5 6 7 8 9 10 11 12 13
2 3 4 5 6 7 8 9 10 14 15 16
```

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

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

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

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

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

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

Code

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

Output:

```
We cannot divide by zero
This is inside finally block
The value of var1 / var2 is :
-----
AssertionError                                                 Traceback (most recent call last)
Input In [44], in ()
    15 # by using assert keyword we will check if var2 is 0
    16 print ("The value of var1 / var2 is : ")
--> 17 assert var2 != 0, "Divide by 0 error"
    18 print (var1 / var2)

AssertionError: Divide by 0 error
```

The pass Keyword

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

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

Code

1. `def function_pass(arguments):`

Output:

```
def function_pass( arguments ):  
    ^  
IndentationError: expected an indented block after function definition on line  
1
```

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

Code

1. `def function_pass(arguments):`

2. `pass`

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

Code

1. `class passed_class:`

2. `pass`

The return Keyword

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

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

Code

1. `def func_with_return():`

2. `var = 13`

3. `return var`

4.

5. `def func_with_no_return():`

6. `var = 10`

7.

8. `print(func_with_return())`

9. `print(func_with_no_return())`

Output:

```
13  
None
```

The del Keyword

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

Code

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

Output:

```
5
-----
NameError                                 Traceback (most recent call last)
Input In [42], in ()
      2 del var1
      3 print(var2)
----> 4 print(var1)

NameError: name 'var1' is not defined
```

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

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

Code

1. `list_ = ['A','B','C']`
2. `del list_[2]`
3. `print(list_)`

Output:

```
['A', 'B']
```

Python Literals

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

Python supports the following literals:

1. String literals:

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

Example:

1. "Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python:

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

Example:

1. text1='hello'

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

There are two ways to create multiline strings:

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

Example:

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

2) Using triple quotation marks:-

Example:

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

Output:

```
welcome
to
SSSIT
```

II. Numeric literals:

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

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

Example - Numeric Literals

1. `x = 0b10100 #Binary Literal`
2. `y = 100 #Decimal Literal`
3. `z = 0o215 #Octal Literal`
4. `u = 0x12d #Hexadecimal Literal`
- 5.
6. `#Float Literal`
7. `float_1 = 100.5`
8. `float_2 = 1.5e2`
- 9.
10. `#Complex Literal`
11. `a = 5+3.14j`
- 12.
13. `print(x, y, z, u)`
14. `print(float_1, float_2)`
15. `print(a, a.imag, a.real)`

Output:

```
20 100 141 301
100.5 150.0
(5+3.14j) 3.14 5.0
```

III. Boolean literals:

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

Example - Boolean Literals

1. `x = (1 == True)`
2. `y = (2 == False)`
3. `z = (3 == True)`
4. `a = True + 10`
5. `b = False + 10`

- 6.
7. **print**("x is", x)
8. **print**("y is", y)
9. **print**("z is", z)
10. **print**("a:", a)
11. **print**("b:", b)

Output:

```
x is True
y is False
z is False
a: 11
b: 10
```

IV. Special literals.

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

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

Example - Special Literals

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

Output:

```
10
None
```

V. Literal Collections.

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

List:

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

Example - List literals

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

4. **print**(list + list1)

Output:

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

Dictionary:

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

Example

1. dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
2. **print**(dict)

Output:

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

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

Example

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

Output:

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

Set:

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

Example: - Set Literals

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

Output:

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

Python Operators

Introduction:

In this article, we are discussing Python Operators. The operator is a symbol that performs a specific operation between two operands, according to one definition. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks. Same as other languages, Python also has some operators, and these are given below -

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

Arithmetic Operators

Arithmetic operators used between two operands for a particular operation. There are many arithmetic operators. It includes the exponent (**), addition (+), subtraction (-), multiplication (*), division (/), remainder (%), and floor division (//) 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 $a = 10, b = 10 \Rightarrow a+b = 20$ |
| - (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 $a = 20, b = 5 \Rightarrow a - b = 15$ |
| / (divide) | It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20, b = 10 \Rightarrow a/b = 2.0$ |
| * (Multiplication) | It is used to multiply one operand with the other. For example, if $a = 20, b = 4 \Rightarrow a * b = 80$ |

| | |
|----------------------------|---|
| % (remainder) | It returns the remainder after dividing the first operand by the second operand. For example, if $a = 20$, $b = 10 \Rightarrow a\%b = 0$ |
| ** (Exponent) | As it calculates the first operand's power to the second operand, it is an exponent operator. |
| // (Floor division) | It provides the quotient's floor value, which is obtained by dividing the two operands. |

Program Code:

Now we give code examples of arithmetic operators in Python. The code is given below -

1. `a = 32 # Initialize the value of a`
2. `b = 6 # Initialize the value of b`
3. `print('Addition of two numbers:',a+b)`
4. `print('Subtraction of two numbers:',a-b)`
5. `print('Multiplication of two numbers:',a*b)`
6. `print('Division of two numbers:',a/b)`
7. `print('Reminder of two numbers:',a%b)`
8. `print('Exponent of two numbers:',a**b)`
9. `print('Floor division of two numbers:',a//b)`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Addition of two numbers: 38
Subtraction of two numbers: 26
Multiplication of two numbers: 192
Division of two numbers: 5.333333333333333
Reminder of two numbers: 2
Exponent of two numbers: 1073741824
Floor division of two numbers: 5
```

Comparison operator

Comparison operators mainly use for comparison purposes. Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The example of comparison operators are `==`, `!=`, `<=`, `>=`, `>`, `<`. In the below table, we explain the works of the operators.

| Operator | Description |
|----------|-------------|
| | |

| | |
|--------------------|---|
| <code>==</code> | If the value of two operands is equal, then the condition becomes true. |
| <code>!=</code> | If the value of two operands is not equal, then the condition becomes true. |
| <code><=</code> | The condition is met if the first operand is smaller than or equal to the second operand. |
| <code>>=</code> | The condition is met if the first operand is greater than or equal to the second operand. |
| <code>></code> | If the first operand is greater than the second operand, then the condition becomes true. |
| <code><</code> | If the first operand is less than the second operand, then the condition becomes true. |

Program Code:

Now we give code examples of Comparison operators in Python. The code is given below -

1. `a = 32 # Initialize the value of a`
2. `b = 6 # Initialize the value of b`
3. `print('Two numbers are equal or not:',a==b)`
4. `print('Two numbers are not equal or not:',a!=b)`
5. `print('a is less than or equal to b:',a<=b)`
6. `print('a is greater than or equal to b:',a>=b)`
7. `print('a is greater b:',a>b)`
8. `print('a is less than b:',a<b)`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Two numbers are equal or not: False
Two numbers are not equal or not: True
a is less than or equal to b: False
a is greater than or equal to b: True
a is greater b: True
a is less than b: False
```

Assignment Operators

Using the assignment operators, the right expression's value is assigned to the left operand. There are some examples of assignment operators like `=`, `+=`, `-=`, `*=`, `%=`, `**=`, `//=`. In the below table, we explain the works of the operators.

| Operator | Description |
|----------|---|
| = | It assigns the value of the right expression to the left operand. |
| += | By multiplying the value of the right operand by the value of the left operand, the left operand receives a changed value. For example, if $a = 10, b = 20 \Rightarrow a+ = b$ 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 $a = 20, b = 10 \Rightarrow a- = b$ 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 $a = 10, b = 20 \Rightarrow a* = b$ 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 remainder back to the left operand. For example, if $a = 20, b = 10 \Rightarrow a \% = b$ 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 . |

Program Code:

Now we give code examples of Assignment operators in Python. The code is given below -

1. `a = 32 # Initialize the value of a`
2. `b = 6 # Initialize the value of b`
3. `print('a=b:', a==b)`
4. `print('a+=b:', a+b)`
5. `print('a-=b:', a-b)`
6. `print('a*=b:', a*b)`
7. `print('a%=b:', a%b)`
8. `print('a**=b:', a**b)`
9. `print('a//=b:', a//b)`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a=b: False
a+=b: 38
a-=b: 26
a*=b: 192
a%=b: 2
a**=b: 1073741824
a/=b: 5
```

Bitwise Operators

The two operands' values are processed bit by bit by the bitwise operators. The examples of Bitwise operators are bitwise OR (|), bitwise AND (&), bitwise XOR (^), negation (~), Left shift (<<), and Right shift (>>). Consider the case below.

For example,

1. **if** a = 7
2. b = 6
3. then, binary (a) = 0111
4. binary (b) = 0110
- 5.
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000
10. Let, Binary of x = 0101
11. Binary of y = 1000
12. Bitwise OR = 1101
13. 8 4 2 1
14. 1 1 0 1 = 8 + 4 + 1 = 13
- 15.
16. Bitwise AND = 0000
17. 0000 = 0
- 18.
19. Bitwise XOR = 1101
20. 8 4 2 1
21. 1 1 0 1 = 8 + 4 + 1 = 13
22. Negation of x = ~x = (-x) - 1 = (-5) - 1 = -6
23. ~x = -6

In the below table, we are explaining the works of the bitwise operators.

| Operator | Description |
|----------|-------------|
|----------|-------------|

| | |
|------------------|---|
| & (binary and) | A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 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) | If the two bits are different, the outcome bit will be 1, else it will be 0. |
| ~ (negation) | The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa. |
| << (left shift) | The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand. |
| >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |

Program Code:

Now we give code examples of Bitwise operators in Python. The code is given below -

1. a = 5 # initialize the value of a
2. b = 6 # initialize the value of b
3. **print('a&b:', a&b)**
4. **print('a|b:', a|b)**
5. **print('a^b:', a^b)**
6. **print('~a:', ~a)**
7. **print('a<<b:', a<<b)**
8. **print('a>>b:', a>>b)**

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
a&b: 4
a|b: 7
a^b: 3
~a: -6
a<>b: 0
```

Logical Operators

The assessment of expressions to make decisions typically uses logical operators. The examples of logical operators are and, or, and not. In the case of logical AND, if the first one is 0, it does not depend upon the second one. In the case of logical OR, if the first one is 1, it does not depend on the second one. Python supports the following logical operators. In the below table, we explain the works of the logical operators.

| Operator | Description |
|----------|--|
| and | The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true. |
| or | The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false. |
| not | If an expression a is true, then not (a) will be false and vice versa. |

Program Code:

Now we give code examples of arithmetic operators in Python. The code is given below -

1. `a = 5 # initialize the value of a`
2. `print('Is this statement true?:', a > 3 and a < 5)`
3. `print('Any one statement is true?:', a > 3 or a < 5)`
4. `print('Each statement is true then return False and vice-versa:', (not(a > 3 and a < 5)))`

Output:

Now we give code examples of Bitwise operators in Python. The code is given below -

```
Is this statement true?: False
Any one statement is true?: True
Each statement is true then return False and vice-versa: True
```

Membership Operators

The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.

| Operator | Description |
|----------|--|
| in | If the first operand cannot be found in the second operand, it is evaluated to be true (list, tuple, or dictionary). |
| not in | If the first operand is not present in the second operand, the evaluation is true (list, tuple, or dictionary). |

Program Code:

Now we give code examples of Membership operators in Python. The code is given below -

1. `x = ["Rose", "Lotus"]`
2. `print(' Is value Present?', "Rose" in x)`
3. `print(' Is value not Present?', "Riya" not in x)`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Is value Present? True  
Is value not Present? True
```

Identity Operators

| Operator | Description |
|---------------------|---|
| <code>is</code> | If the references on both sides point to the same object, it is determined to be true. |
| <code>is not</code> | If the references on both sides do not point at the same object, it is determined to be true. |

Program Code:

Now we give code examples of Identity operators in Python. The code is given below -

1. `a = ["Rose", "Lotus"]`
2. `b = ["Rose", "Lotus"]`
3. `c = a`
4. `print(a is c)`
5. `print(a is not c)`
6. `print(a is b)`
7. `print(a is not b)`
8. `print(a == b)`
9. `print(a != b)`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
True  
False  
False  
True
```

True
False

Operator Precedence

The order in which the operators are examined is crucial to understand since it tells us which operator needs to be considered first. Below is a list of the Python operators' precedence tables.

| Operator | Description |
|---------------------|--|
| ** | Overall other operators employed in the expression, the exponent operator is given precedence. |
| ~ + - | the minus, unary plus, and negation. |
| * / % // | the division of the floor, the modules, the division, and the multiplication. |
| + - | 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 |

Conclusion:

So, in this article, we are discussing all the Python Operators. We briefly discuss how they work and share the program code using each operator in Python.

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
- 2.
3. **def** add(x, y):
4. """This function adds the values of x and y"""
5. **return** x + y
- 6.
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

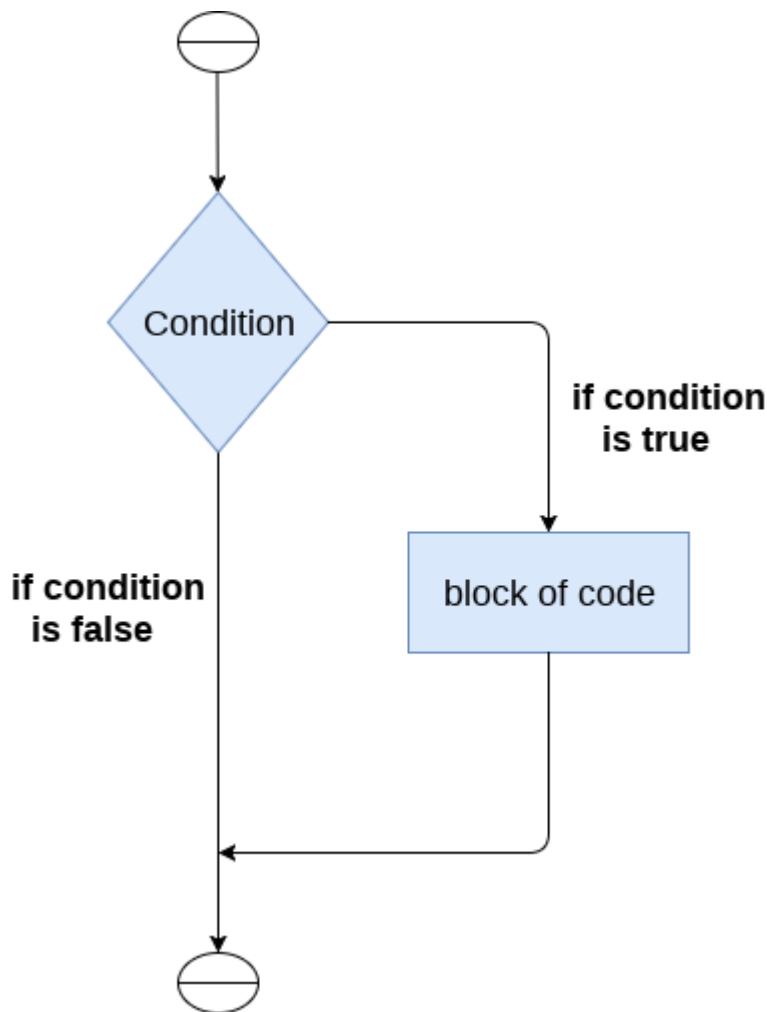
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

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. # Simple Python program to understand the if statement
2. num = int(input("enter the number:"))
3. # Here, we are taking an integer num and taking input dynamically
4. if num%2 == 0:
5. # Here, we are checking the condition. If the condition is true, we will enter the block
6. print("The Given number is an even number")

Output:

```

enter the number: 10
The Given number is an even number

```

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

1. # Simple Python Program to print the largest of the three numbers.
2. a = int (input("Enter a: "));
3. b = int (input("Enter b: "));
4. c = int (input("Enter c: "));

```
5. if a>b and a>c:  
6. # Here, we are checking the condition. If the condition is true, we will enter the block  
7. print ("From the above three numbers given a is largest");  
8. if b>a and b>c:  
9. # Here, we are checking the condition. If the condition is true, we will enter the block  
10. print ("From the above three numbers given b is largest");  
11. if c>a and c>b:  
12. # Here, we are checking the condition. If the condition is true, we will enter the block  
13. print ("From the above three numbers given c is largest");
```

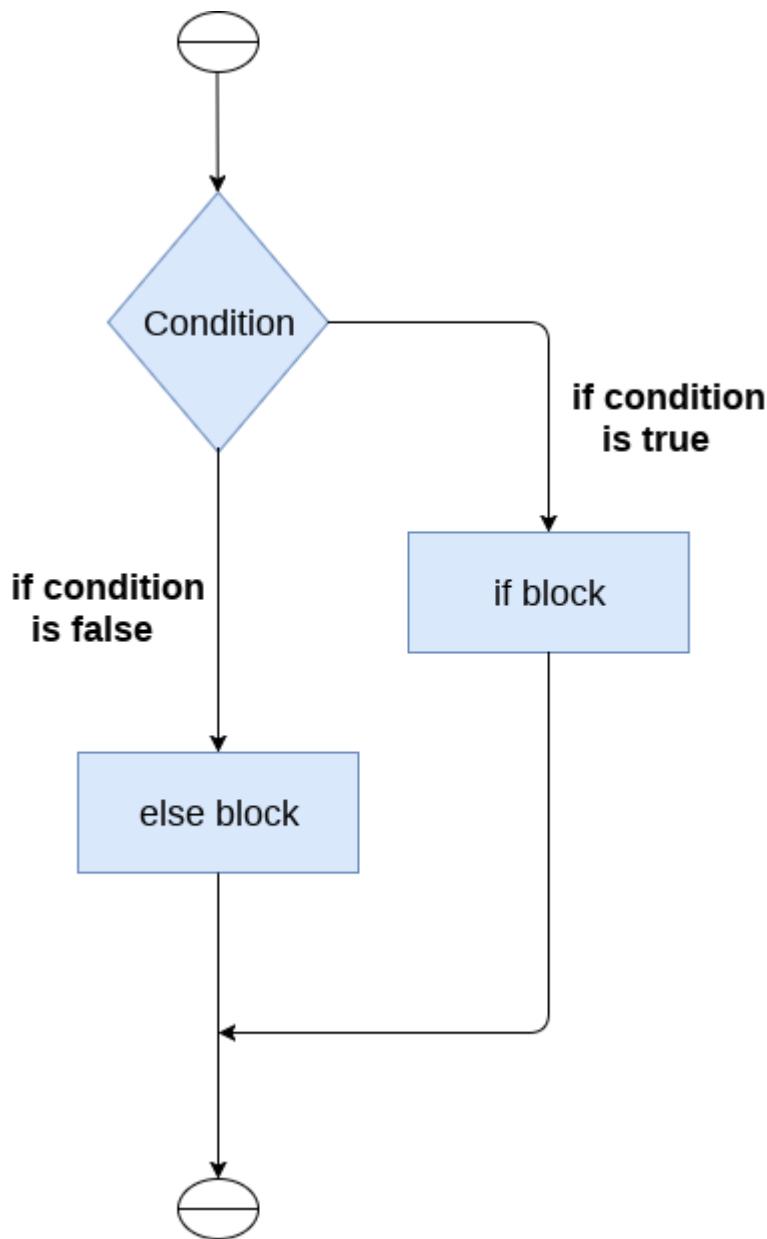
Output:

```
Enter a: 100  
Enter b: 120  
Enter c: 130  
From the above three numbers given 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. **# Simple Python Program to check whether a person is eligible to vote or not.**
2. **age = int(input("Enter your age: "))**
3. **# Here, we are taking an integer num and taking input dynamically**
4. **if age>=18:**
5. **# Here, we are checking the condition. If the condition is true, we will enter the block**
6. **print("You are eligible to vote !!");**
7. **else:**

```
8.     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. # Simple Python Program to check whether a number is even or not.
2. num = int(input("enter the number:"))
3. # Here, we are taking an integer num and taking input dynamically
4. if num%2 == 0:
5. # Here, we are checking the condition. If the condition is true, we will enter the block
6.     print("The Given number is an even number")
7. else:
8.     print("The Given Number is an odd number")
```

Output:

```
enter the number: 10
The Given number is even number
```

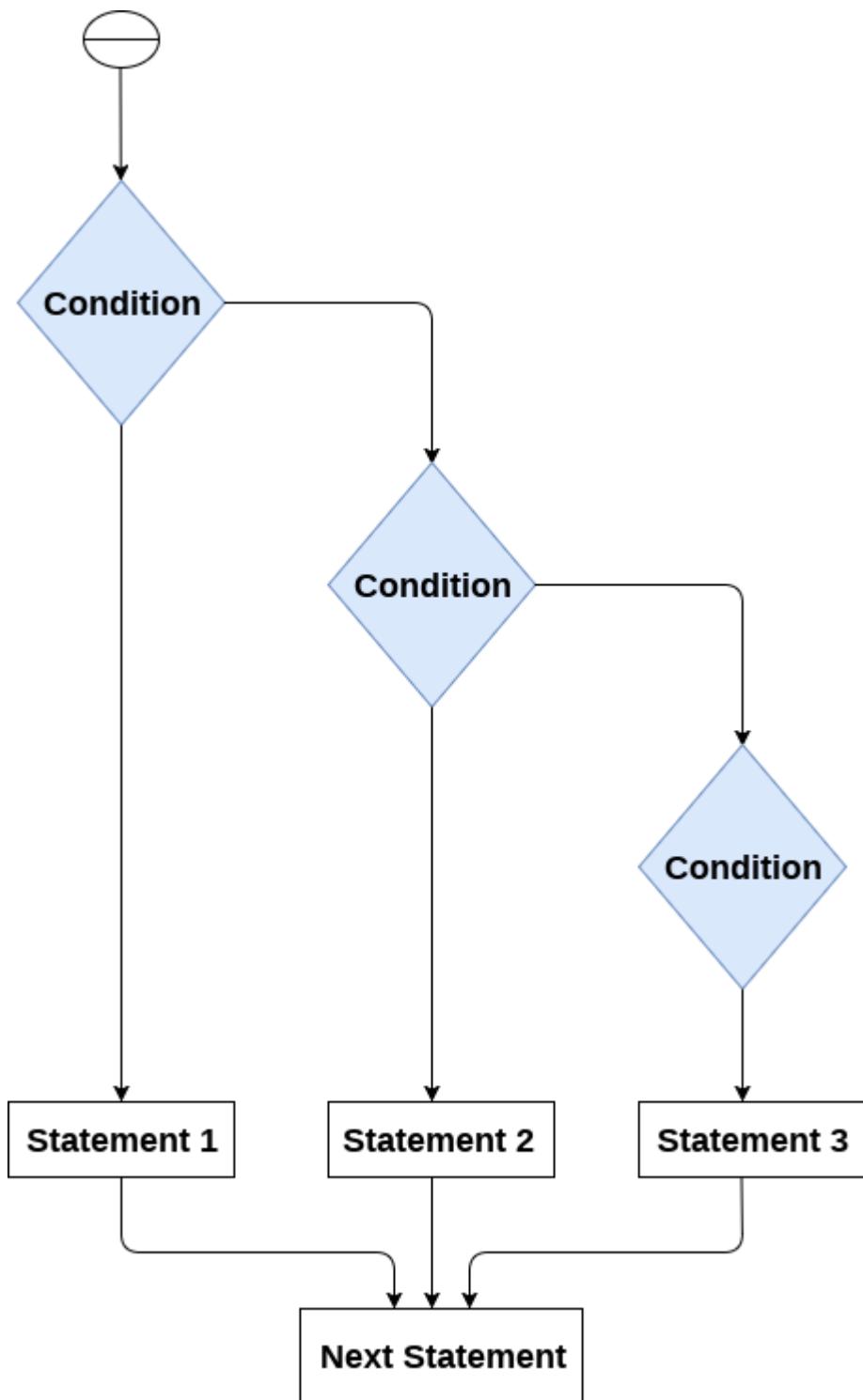
The elif statement

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.
4. elif expression 2:
5.     # block of statements
6.
7. elif expression 3:
8.     # block of statements
9.
10. else:
11.     # block of statements
```



Example 1

1. # Simple Python program to understand elif statement
2. number = int(input("Enter the number?"))
3. # Here, we are taking an integer number and taking input dynamically
4. if number==10:
5. # Here, we are checking the condition. If the condition is true, we will enter the block
6. print("The given number is equals to 10")
7. elif number==50:
8. # Here, we are checking the condition. If the condition is true, we will enter the block
9. print("The given number is equal to 50");

```
10. elif number==100:  
11. # Here, we are checking the condition. If the condition is true, we will enter the block  
12.     print("The given number is equal to 100");  
13. else:  
14.     print("The given number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15  
The given number is not equal to 10, 50 or 100
```

Example 2

```
1. # Simple Python program to understand elif statement  
2. marks = int(input("Enter the marks? "))  
3. # Here, we are taking an integer marks and taking input dynamically  
4. if marks > 85 and marks <= 100:  
5. # Here, we are checking the condition. If the condition is true, we will enter the block  
6.     print("Congrats ! you scored grade A ...")  
7. elif marks > 60 and marks <= 85:  
8. # Here, we are checking the condition. If the condition is true, we will enter the block  
9.     print("You scored grade B + ...")  
10. elif marks > 40 and marks <= 60:  
11. # Here, we are checking the condition. If the condition is true, we will enter the block  
12.     print("You scored grade B ...")  
13. elif (marks > 30 and marks <= 40):  
14. # Here, we are checking the condition. If the condition is true, we will enter the block  
15.     print("You scored grade C ...")  
16. else:  
17.     print("Sorry you are fail ?")
```

Output:

```
Enter the marks? 89  
Congrats ! you scored grade A ...
```

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.

3. # Creating a sequence which is a tuple of numbers

4. numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]

5.

6. # variable to store the square of the number

7. square = 0

8.

9. # Creating an empty list

10. squares = []

11.

12. # Creating a for loop

13. **for** value **in** numbers:

14. square = value ** 2

15. squares.append(square)

16. **print**("The list of squares is", squares)

Output:

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

Using else Statement with for Loop

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

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

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

Code

1. # Python program to show how if-else statements work

2.

```
3. string = "Python Loop"
4.
5. # Initiating a loop
6. for s in a string:
7.     # giving a condition in if block
8.     if s == "o":
9.         print("If block")
10.    # if condition is not satisfied then else block will be executed
11.   else:
12.       print(s)
```

Output:

```
P
Y
t
h
If block
n

L
If block
If block
p
```

Now similarly, using else with for loop.

Syntax:

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

Code

```
1. # Python program to show how to use else statement with for loop
2.
3. # Creating a sequence
4. tuple_ = (3, 4, 6, 8, 9, 2, 3, 8, 9, 7)
5.
6. # Initiating the loop
7. for value in tuple_:
8.     if value % 2 != 0:
9.         print(value)
10.    # giving an else statement
11. else:
```

12. `print("These are the odd numbers present in the tuple")`

Output:

```
3  
9  
3  
9  
7  
These are the odd numbers present in the tuple
```

The range() Function

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

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

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

The example that follows will make this clear.

Code

1. `# Python program to show the working of range() function`
- 2.
3. `print(range(15))`
- 4.
5. `print(list(range(15)))`
- 6.
7. `print(list(range(4, 9)))`
- 8.
9. `print(list(range(5, 25, 4)))`

Output:

```
range(0, 15)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]  
[4, 5, 6, 7, 8]  
[5, 9, 13, 17, 21]
```

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

Code

1. `# Python program to iterate over a sequence with the help of indexing`

```
2.  
3. tuple_ = ("Python", "Loops", "Sequence", "Condition", "Range")  
4.  
5. # iterating over tuple_ using range() function  
6. for iterator in range(len(tuple_)):  
7.     print(tuple_[iterator].upper())
```

Output:

```
PYTHON  
LOOPS  
SEQUENCE  
CONDITION  
RANGE
```

While Loop

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

Syntax of the while loop is:

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

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

Code

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

Output:

```
Python Loops  
Python Loops  
Python Loops  
Python Loops
```

Using else Statement with while Loops

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

Code

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

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
Code block inside the else statement
```

Single statement while Block

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

Code

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

Loop Control Statements

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

Continue Statement

It returns the control to the beginning of the loop.

Code

```
1. # Python program to show how the continue statement works
2.
3. # Initiating the loop
4. for string in "Python Loops":
5.     if string == "o" or string == "p" or string == "t":
```

6. **continue**
7. **print('Current Letter:', string)**

Output:

```
Current Letter: P
Current Letter: Y
Current Letter: h
Current Letter: n
Current Letter:
Current Letter: L
Current Letter: s
```

Break Statement

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

Code

1. **# Python program to show how the break statement works**
- 2.
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 strong, universally applicable prearranging language planned to be easy to comprehend and carry out. It is allowed to get to because it is open-source. In this tutorial, we will learn how to use Python for loops, one of the most fundamental looping instructions in Python programming.

Introduction to for Loop in Python

Python frequently uses the Loop to iterate over iterable objects like lists, tuples, and strings. Crossing is the most common way of emphasizing across a series, for loops are used when a section of code needs to be repeated a certain number of times. The for-circle is typically utilized on an iterable item, for example, a rundown or the in-fabricated range capability. In Python, the for Statement runs the code block each time it traverses a series of elements. On the other hand, the "while" Loop is used when a condition needs to be verified after each repetition or when a piece of code needs to be repeated indefinitely. The for Statement is opposed to this Loop.

Syntax of for Loop

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

The value is the parameter that determines the element's value within the iterable sequence on each iteration. When a sequence contains expression statements, they are processed first. The first element in the sequence is then assigned to the iterating variable iterating_variable. From that point onward, the planned block is run. Each element in the sequence is assigned to iterating_variable during the statement block until the sequence as a whole is completed. Using indentation, the contents of the Loop are distinguished from the remainder of the program.

Example of Python for Loop

Code

1. # Code to find the sum of squares of each element of the list using for loop
- 2.
3. # creating the list of numbers
4. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
- 5.
6. # initializing a variable that will store the sum
7. sum_ = 0
- 8.
9. # using for loop to iterate over the list
10. **for** num **in** numbers:
- 11.

```
12. sum_ = sum_ + num ** 2  
13.  
14. print("The sum of squares is: ", sum_)
```

Output:

```
The sum of squares is: 774
```

The range() Function

Since the "range" capability shows up so habitually in for circles, we could erroneously accept the reach as a part of the punctuation of for circle. It's not: It is a built-in Python method that fulfills the requirement of providing a series for the for expression to run over by following a particular pattern (typically serial integers). Mainly, they can act straight on sequences, so counting is unnecessary. This is a typical novice construct if they originate from a language with distinct loop syntax:

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.  
3. # creating the list of numbers  
4. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]  
5.  
6. # initializing a variable that will store the sum  
7. sum_ = 0  
8.  
9. # using for loop to iterate over list  
10. for num in range( len(numbers) ):  
11.
```

```
12. sum_ = sum_ + numbers[num] ** 2  
13.  
14. print("The sum of squares is: ", sum_)
```

Output:

```
The sum of squares is: 774
```

The len() worked in a technique that profits the complete number of things in the rundown or tuple, and the implicit capability range(), which returns the specific grouping to emphasize over, proved helpful here.

Using else Statement with for Loop

A loop expression and an else expression can be connected in Python.

After the circuit has finished iterating over the list, the else clause is combined with a for Loop.

The following example demonstrates how to extract students' marks from the record by combining a for expression with an otherwise statement.

Code

```
1. # code to print marks of a student from the record  
2. student_name_1 = 'Itika'  
3. student_name_2 = 'Parker'  
4.  
5.  
6. # Creating a dictionary of records of the students  
7. records = {'Itika': 90, 'Arshia': 92, 'Peter': 46}  
8. def marks( student_name ):  
9.     for a_student in record: # for loop will iterate over the keys of the dictionary  
10.        if a_student == student_name:  
11.            return records[ a_student ]  
12.            break  
13.    else:  
14.        return f'There is no student of name {student_name} in the records'  
15.  
16. # giving the function marks() name of two students  
17. print( f'Marks of {student_name_1} are: ', marks( student_name_1 ) )  
18. print( f'Marks of {student_name_2} are: ', marks( student_name_2 ) )
```

Output:

```
Marks of Itika are: 90
```

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

Nested Loops

If we have a piece of content that we need to run various times and, afterward, one more piece of content inside that script that we need to run B several times, we utilize a "settled circle." While working with an iterable in the rundowns, Python broadly uses these.

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

In this article, we are discussing while loops in Python. 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

Now, here we discuss the syntax of the Python while loop. The syntax is given below -

1. Statement
2. **while** Condition:
3. Statement

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.

Example

Now we give some examples of while Loop in Python. The examples are given in below -

Program code 1:

Now we give code examples of while loops in Python for printing numbers from 1 to 10. The code is given below -

1. `i=1`
2. `while i<=10:`
3. `print(i, end=' ')`
4. `i+=1`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
1 2 3 4 5 6 7 8 9 10
```

Program Code 2:

Now we give code examples of while loops in Python for Printing those numbers divisible by either 5 or 7 within 1 to 50 using a while loop. The code is given below -

1. `i=1`
2. `while i<51:`
3. `if i%5 == 0 or i%7==0 :`
4. `print(i, end=' ')`
5. `i+=1`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

Program Code:

Now we give code examples of while loops in Python, the sum of squares of the first 15 natural numbers using a while loop. The code is given below -

```

1. # Python program example to show the use of while loop
2.
3. num = 15
4.
5. # initializing summation and a counter for iteration
6. summation = 0
7. c = 1
8.
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
13.
14. # print the final sum
15. print("The sum of squares is", summation)

```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
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.

Program Code:

Now we give code examples of while loops in Python for a number is Prime number or not. The code is given below -

```
1. num = [34, 12, 54, 23, 75, 34, 11]
2.
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
11.
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:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
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
```

2. Armstrong and Python While Loop

We will construct a Python program using a while loop to verify whether the given integer is an Armstrong number.

Program Code:

Now we give code examples of while loops in Python for a number is Armstrong number or not. The code is given below -

```
1. n = int(input())
2. n1=str(n)
3. l=len(n1)
4. temp=n
```

```
5. s=0
6. while n!=0:
7.     r=n%10
8.     s=s+(r**1)
9.     n=n//10
10. if s==temp:
11.     print("It is an Armstrong number")
12. else:
13.     print("It is not an Armstrong number ")
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
342
It is not an Armstrong number
```

Multiplication Table using While Loop

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

Program Code:

In this example, we will use the while loop for printing the multiplication table of a given number. The code is given below -

```
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:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
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

Program Code 1:

Now we give code examples of while loops in Python for square every number of a list. The code is given below -

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 of all numbers.
9. print(squares)

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
[36, 16, 1, 25, 9]
```

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

Program Code 2:

Now we give code examples of while loops in Python for determine odd and even number from every number of a list. The code is given below -

1. list_ = [3, 4, 8, 10, 34, 45, 67, 80] # Initialize the list
2. index = 0
3. **while** index < len(list_):
4. element = list_[index]
5. **if** element % 2 == 0:
6. print('It is an even number') # Print **if** the number is even.
7. **else**:
8. print('It is an odd number') # Print **if** the number is odd.
9. index += 1

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
It is an odd number
It is an even number
It is an odd number
It is an odd number
It is an even number
```

Program Code 3:

Now we give code examples of while loops in Python for determine the number letters of every word from the given list. The code is given below -

1. List_= ['Priya', 'Neha', 'Cow', 'To']
2. index = 0
3. **while** index < len(List_):
4. element = List_[index]
5. print(len(element))
6. index += 1

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
5
4
3
2
```

Python While Loop Multiple Conditions

We must recruit logical operators to combine two or more expressions specifying conditions into a single while loop. This instructs Python on collectively analyzing all 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 the Loop will execute the statements until both conditions give Boolean True.

Program Code:

Now we give code examples of while loops in Python for multiple condition. The code is given below -

1. num1 = 17
2. num2 = -12

```
3.  
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:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

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

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
(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:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
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
- 2.
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)

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

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
- 2.
3. # Initiating the loop
4. **for** string in "Python Loops":
5. **if** string == 'n':
6. **break**
7. print('Current Letter: ', string)

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
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**('The Last Letter of given string is:', string)

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

Output:

```
The Last Letter of given string is: 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 statement in Python is given below.

Syntax:

1. #loop statements
2. **break;**

Example 1 : break statement with for loop

Code

```
1. # break statement example
2. my_list = [1, 2, 3, 4]
3. count = 1
4. for item in my_list:
5.     if item == 4:
6.         print("Item matched")
7.         count += 1
8.         break
9. print("Found at location", count)
```

Output:

```
Item matched
Found at location 2
```

In the above example, a list is iterated using a for loop. When the item is matched with value 4, the break statement is executed, and the loop terminates. Then the count is printed by locating the item.

Example 2 : Breaking out of a loop early

Code

```
1. # break statement example
2. my_str = "python"
3. for char in my_str:
```

```
4. if char == 'o':  
5.     break  
6. print(char)
```

Output:

```
p  
y  
t  
h
```

When the character is found in the list of characters, break starts executing, and iterating stops immediately. Then the next line of the print statement is printed.

Example 3: break statement with while loop

Code

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

Output:

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

It is the same as the above programs. The while loop is initialised to True, which is an infinite loop. When the value is 10 and the condition becomes true, the break statement will be executed and jump to the later print statement by terminating the while loop.

Example 4 : break statement with nested loops

Code

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

```
9. if choice == 0:  
10.     print("Exiting the program...")  
11.     break  
12.     n += 1  
13. print("Program finished successfully.")
```

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  
Exiting the program...  
Program finished successfully.
```

There are two nested loops in the above program. Inner loop and outer loop. The inner loop is responsible for printing the multiplication table, whereas the outer loop is responsible for incrementing the n value. When the inner loop completes execution, the user will have to continue printing. When 0 is entered, the break statement finally executes, and the nested loop is terminated.

Python continue Statement

Python continue keyword is used to skip the remaining statements of the current loop and go to the next iteration. 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. Both Python while and Python for loops can leverage the continue statements.

Syntax:

1. **continue**

Python Continue Statements in for Loop

Printing numbers from 10 to 20 except 15 can be done using continue statement and for loop. The following code is an example of the above scenario:

Code

```
1. # Python code to show example of continue statement
2.
3. # looping from 10 to 20
4. for iterator in range(10, 21):
5.
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
```

Explanation: 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.

Python Continue Statements in while Loop

Code

```
1. # Creating a string
2. string = "JavaTpoint"
3. # initializing an iterator
4. iterator = 0
5.
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
```

Explanation: 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.

Python Continue statement in list comprehension

Let's see the example for continue statement in list comprehension.

Code

```
1. numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2.
3. # Using a list comprehension with continue
4. sq_num = [num ** 2 for num in numbers if num % 2 == 0]
5. # This will skip odd numbers and only square the even numbers
6. print(sq_num)
```

Output:

```
[4, 16, 36, 64, 100]
```

Explanation: In the above code, list comprehension will square the numbers from the list. And continue statement will be encountered when odd numbers come and the loop will skip the execution and moves to the next iteration.

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

In this tutorial, we will learn more about past statements. It is interpreted as a placeholder for future functions, classes, loops, and other operations.

What is Python's Pass Statement?

The pass statement is also known as the null statement. The Python mediator doesn't overlook a Remark, though a pass proclamation isn't. As a result, these two Python keywords are distinct.

We can use the pass statement as a placeholder when unsure of the code to provide. Therefore, the pass only needs to be placed on that line. The pass might be utilized when we wish no code to be executed. We can simply insert a pass in cases where empty code is prohibited, such as in loops, functions, class definitions, and if-else statements.

Syntax

1. Keyword:
2. **pass**

Ordinarily, we use it as a perspective for what's to come.

Let's say we have an if-else statement or loop that we want to fill in the future but cannot. An empty body for the pass keyword would be grammatically incorrect. A mistake would be shown by the Python translator proposing to occupy the space. As a result, we use the pass statement to create a code block that does nothing.

An Illustration 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
2.
3. # Empty function:
4. def empty():
5.     pass
6.
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)**
- 7.
8. #Using triple quotes
9. str3 = """Triple quotes are generally used for
represent the multiline or
docstring"""
10. **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.

str = "HELLO"

| | | | | |
|---|---|---|---|---|
| H | E | L | L | O |
| 0 | 1 | 2 | 3 | 4 |

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

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.

str = "HELLO"

| | | | | |
|---|---|---|---|---|
| H | E | L | L | O |
| 0 | 1 | 2 | 3 | 4 |

str[0] = 'H' str[:] = 'HELLO'

str[1] = 'E' str[0:] = 'HELLO'

str[2] = 'L' str[:5] = 'HELLO'

str[3] = 'L' str[:3] = 'HEL'

str[4] = 'O' str[0:2] = 'HE'

str[1:4] = 'ELL'

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 0th 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:

```
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.

| str = "HELLO" | | | | |
|---------------|----|---------------------|----|----|
| H | E | L | L | O |
| -5 | -4 | -3 | -2 | -1 |
| str[-1] = 'O' | | str[-3:-1] = 'LL' | | |
| str[-2] = 'L' | | str[-4:-1] = 'ELL' | | |
| str[-3] = 'L' | | str[-5:-3] = 'HE' | | |
| str[-4] = 'E' | | str[-4:] = 'ELLO' | | |
| str[-5] = 'H' | | str[::-1] = 'OLLEH' | | |

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
11
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?""")`
- 3.
4. `# escaping single quotes`
5. `print('They said, "What\'s going on?")`
- 6.
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. | <pre>print("Python1 \ Python2 \ Python3")</pre> Output: Python1 Python2 Python3 |
| 2. | \\" | Backslash | <pre>print("\\")</pre> Output: \ |
| 3. | ' | Single Quotes | <pre>print('\'')</pre> Output: ' |
| 4. | \\" | Double Quotes | <pre>print("\\\"")</pre> Output: " |
| 5. | \a | ASCII Bell | <pre>print("\a")</pre> |
| 6. | \b | ASCII Backspace(BS) | <pre>print("Hello \b World")</pre> Output: Hello World |
| 7. | \f | ASCII Formfeed | <pre>print("Hello \f World!")</pre> Hello World! |
| 8. | \n | ASCII Linefeed | <pre>print("Hello \n World!")</pre> Output: Hello World! |
| 9. | \r | ASCII Carrige Return(CR) | <pre>print("Hello \r World!")</pre> Output: World! |
| 10. | \t | ASCII Horizontal Tab | <pre>print("Hello \t World!")</pre> Output: Hello World! |
| 11. | \v | ASCII Vertical Tab | <pre>print("Hello \v World!")</pre> |

| | | | |
|-----|------|----------------------------|--|
| | | | 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"))`
- 3.
4. **#Positional Argument**
5. `print("{1} and {0} best players ".format("Virat","Rohit"))`
- 6.
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"\n%(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 |
|---|---|
| <u>capitalize()</u> | It capitalizes the first character of the String. This function is deprecated in python3 |
| <u>casefold()</u> | It returns a version of s suitable for case-less comparisons. |
| <u>center(width ,fillchar)</u> | It returns a space padded string with the original string centred with equal number of left and right spaces. |
| <u>count(string,begin,end)</u> | It counts the number of occurrences of a substring in a String between begin and end index. |
| <u>decode(encoding = 'UTF8', errors = 'strict')</u> | Decodes the string using codec registered for encoding. |

| | |
|--|---|
| <u>encode()</u> | Encode S using the codec registered for encoding. Default encoding is 'utf-8'. |
| <u>endswith(suffix, begin=0,end=len(string))</u> | It returns a Boolean value if the string terminates with given suffix between begin and end. |
| <u>expandtabs(tabsize = 8)</u> | It defines tabs in string to multiple spaces. The default space value is 8. |
| <u>find(substring, beginIndex, endIndex)</u> | It returns the index value of the string where substring is found between begin index and end index. |
| <u>format(value)</u> | It returns a formatted version of S, using the passed value. |
| <u>index(substring, beginIndex, endIndex)</u> | It throws an exception if string is not found. It works same as find() method. |
| <u>isalnum()</u> | 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. |
| <u>isalpha()</u> | It returns true if all the characters are alphabets and there is at least one character, otherwise False. |
| <u>isdecimal()</u> | It returns true if all the characters of the string are decimals. |
| <u>isdigit()</u> | It returns true if all the characters are digits and there is at least one character, otherwise False. |
| <u>isidentifier()</u> | It returns true if the string is the valid identifier. |
| <u>islower()</u> | It returns true if the characters of a string are in lower case, otherwise false. |
| <u>isnumeric()</u> | It returns true if the string contains only numeric characters. |
| <u>isprintable()</u> | It returns true if all the characters of s are printable or s is empty, false otherwise. |
| <u>isupper()</u> | It returns false if characters of a string are in Upper case, otherwise False. |

| | |
|---|---|
| <u>isspace()</u> | It returns true if the characters of a string are white-space, otherwise false. |
| <u>istitle()</u> | 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. |
| <u>isupper()</u> | It returns true if all the characters of the string(if exists) is true otherwise it returns false. |
| <u>join(seq)</u> | It merges the strings representation of the given sequence. |
| <u>len(string)</u> | It returns the length of a string. |
| <u>ljust(width[,fillchar])</u> | It returns the space padded strings with the original string left justified to the given width. |
| <u>lower()</u> | It converts all the characters of a string to Lower case. |
| <u>lstrip()</u> | It removes all leading whitespaces of a string and can also be used to remove particular character from leading. |
| <u>partition()</u> | 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. |
| <u>maketrans()</u> | It returns a translation table to be used in translate function. |
| <u>replace(old,new[,count])</u> | It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given. |
| <u>rfind(str,beg=0,end=len(str))</u> | It is similar to find but it traverses the string in backward direction. |
| <u>rindex(str,beg=0,end=len(str))</u> | It is same as index but it traverses the string in backward direction. |
| <u>rjust(width[,fillchar])</u> | Returns a space padded string having original string right justified to the number of characters specified. |
| <u>rstrip()</u> | It removes all trailing whitespace of a string and can also be used to remove particular character from trailing. |

| | |
|---|---|
| <u>rsplit(sep=None, maxsplit = -1)</u> | 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. |
| <u>split(str,num=string.count(str))</u> | 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. |
| <u>splitlines(num=string.count('\n'))</u> | It returns the list of strings at each line with newline removed. |
| <u>startswith(str,beg=0,end=len(str))</u> | It returns a Boolean value if the string starts with given str between begin and end. |
| <u>strip([chars])</u> | It is used to perform lstrip() and rstrip() on the string. |
| <u>swapcase()</u> | It inverts case of all characters in a string. |
| <u>title()</u> | It is used to convert the string into the title-case i.e., The string meEruT will be converted to Meerut. |
| <u>translate(table,deletechars = "")</u> | It translates the string according to the translation table passed in the function . |
| <u>upper()</u> | It converts all the characters of a string to Upper Case. |
| <u>zfill(width)</u> | Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| <u>rpartition()</u> | |

Python List

In Python, the sequence of various data types is stored in a list. A list is a collection of different kinds of values or items. Since Python lists are mutable, we can change their elements after forming. The comma (,) and the square brackets [enclose the List's items] serve as separators.

Although six Python data types can hold sequences, the List is the most common and reliable form. A list, a type of sequence data, is used to store the collection of data. Tuples and Strings are two similar data formats for sequences.

Lists written in Python are identical to dynamically scaled arrays defined in other languages, such as ArrayList in Java and Vector in C++. A list is a collection of items separated by commas and denoted by the symbol [].

List Declaration

Code

```
1. # a simple list
2. list1 = [1, 2, "Python", "Program", 15.9]
3. list2 = ["Amy", "Ryan", "Henry", "Emma"]
4.
5. # printing the list
6. print(list1)
7. print(list2)
8.
9. # printing the type of list
10. print(type(list1))
11. print(type(list2))
```

Output:

```
[1, 2, 'Python', 'Program', 15.9]
['Amy', 'Ryan', 'Henry', 'Emma']
< class 'list' >
< class 'list' >
```

Characteristics of Lists

The characteristics of the List are as follows:

- The lists are in order.
- The list element can be accessed via the index.
- The mutable type of List is
- The rundowns are changeable sorts.
- The number of various elements can be stored in a list.

Ordered List Checking

Code

```
1. # example
2. a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
3. b = [ 1, 2, 5, "Ram", 3.50, "Rahul", 6 ]
4. a == b
```

Output:

```
False
```

The indistinguishable components were remembered for the two records; however, the subsequent rundown changed the file position of the fifth component, which is against the rundowns' planned request. False is returned when the two lists are compared.

Code

1. `# example`
2. `a = [1, 2, "Ram", 3.50, "Rahul", 5, 6]`
3. `b = [1, 2, "Ram", 3.50, "Rahul", 5, 6]`
4. `a == b`

Output:

```
True
```

Records forever protect the component's structure. Because of this, it is an arranged collection of things.

Let's take a closer look at the list example.

Code

1. `# list example in detail`
2. `emp = ["John", 102, "USA"]`
3. `Dep1 = ["CS",10]`
4. `Dep2 = ["IT",11]`
5. `HOD_CS = [10,"Mr. Holding"]`
6. `HOD_IT = [11, "Mr. Bewon"]`
7. `print("printing employee data ...")`
8. `print(" Name : %s, ID: %d, Country: %s" %(emp[0], emp[1], emp[2]))`
9. `print("printing departments ...")`
10. `print("Department 1:\nName: %s, ID: %d\n Department 2:\n Name: %s, ID: %s"%(Dep1[0], Dep2[1], Dep2[0], Dep2[1]))`
11. `print("HOD Details")`
12. `print("CS HOD Name: %s, Id: %d" %(HOD_CS[1], HOD_CS[0]))`
13. `print("IT HOD Name: %s, Id: %d" %(HOD_IT[1], HOD_IT[0]))`
14. `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 preceding illustration, we printed the employee and department-specific details from lists that we had created. To better comprehend the List's concept, look at the code above.

List Indexing and Splitting

The indexing procedure is carried out similarly to string processing. The slice operator [] can be used to get to the List's components.

The index ranges from 0 to length -1. The 0th index is where the List's first element is stored; the 1st index is where the second element is stored, and so on.

$$\text{List} = [0, 1, 2, 3, 4, 5]$$

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

$$\text{List}[0] = 0$$

$$\text{List}[0:] = [0, 1, 2, 3, 4, 5]$$

$$\text{List}[1] = 1$$

$$\text{List}[:] = [0, 1, 2, 3, 4, 5]$$

$$\text{List}[2] = 2$$

$$\text{List}[2:4] = [2, 3]$$

$$\text{List}[3] = 3$$

$$\text{List}[1:3] = [1, 2]$$

$$\text{List}[4] = 4$$

$$\text{List}[:4] = [0, 1, 2, 3]$$

$$\text{List}[5] = 5$$

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

1. list_variable(start:stop:step)

- The beginning indicates the beginning record position of the rundown.
- The stop signifies the last record position of the rundown.
- Within a start, the step is used to skip the nth element: stop.

The start parameter is the initial index, the step is the ending index, and the value of the end parameter is the number of elements that are "stepped" through. The default value for the step is one without a specific value. Inside the resultant Sub List, the same with

record start would be available, yet the one with the file finish will not. The first element in a list appears to have an index of zero.

Consider the following example:

Code

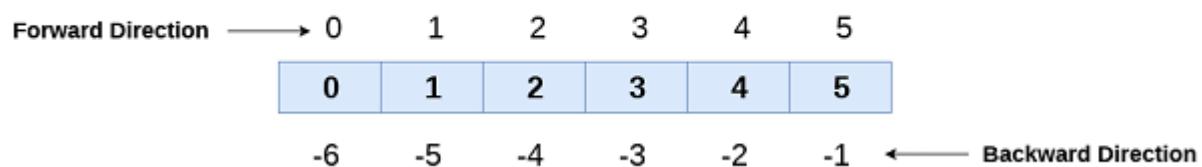
```
1. list = [1,2,3,4,5,6,7]
2. print(list[0])
3. print(list[1])
4. print(list[2])
5. print(list[3])
6. # Slicing the elements
7. print(list[0:6])
8. # By default, the index value is 0 so its starts from the 0th element and go for index -1.
9. print(list[:])
10. print(list[2:5])
11. print(list[1:6:2])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

Output:

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

In contrast to other programming languages, Python lets you use negative indexing as well. The negative indices are counted from the right. The index -1 represents the final element on the List's right side, followed by the index -2 for the next member on the left, and so on, until the last element on the left is reached.

List = [0, 1, 2, 3, 4, 5]



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

Code

```
1. # negative indexing example
```

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

Output:

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

Negative indexing allows us to obtain an element, as previously mentioned. The rightmost item in the List was returned by the first print statement in the code above. The second print statement returned the sub-list, and so on.

Updating List Values

Due to their mutability and the slice and assignment operator's ability to update their values, lists are Python's most adaptable data structure. Python's append() and insert() methods can also add values to a list.

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

Code

1. **# updating list values**
2. list = [1, 2, 3, 4, 5, 6]
3. **print**(list)
4. **# It will assign value to the value to the second index**
5. list[2] = 10
6. **print**(list)
7. **# Adding multiple-element**
8. list[1:3] = [89, 78]
9. **print**(list)
10. **# It will add value at the end of the list**
11. list[-1] = 25
12. **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.

Code

```
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. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Iteration
5. Membership

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

1. Repetition

The redundancy administrator empowers the rundown components to be rehashed on different occasions.

Code

1. `# repetition of list`
2. `# declaring the list`
3. `list1 = [12, 14, 16, 18, 20]`
4. `# repetition operator *`
5. `l = list1 * 2`
6. `print(l)`

Output:

```
[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]
```

2. Concatenation

It concatenates the list mentioned on either side of the operator.

Code

1. `# concatenation of two lists`
2. `# declaring the lists`
3. `list1 = [12, 14, 16, 18, 20]`
4. `list2 = [9, 10, 32, 54, 86]`
5. `# concatenation operator +`
6. `l = list1 + list2`
7. `print(l)`

Output:

```
[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]
```

3. Length

It is used to get the length of the list

Code

1. `# size of the list`
2. `# declaring the list`
3. `list1 = [12, 14, 16, 18, 20, 23, 27, 39, 40]`
4. `# finding length of the list`
5. `len(list1)`

Output:

9

4. Iteration

The for loop is used to iterate over the list elements.

Code

```
1. # iteration of the list
2. # declaring the list
3. list1 = [12, 14, 16, 39, 40]
4. # iterating
5. for i in list1:
6.     print(i)
```

Output:

```
12
14
16
39
40
```

5. Membership

It returns true if a particular item exists in a particular list otherwise false.

Code

```
1. # membership of the list
2. # declaring the list
3. list1 = [100, 200, 300, 400, 500]
4. # true will be printed if value exists
5. # and false if not
6.
7. print(600 in list1)
8. print(700 in list1)
9. print(1040 in list1)
10.
11. print(300 in list1)
12. print(100 in list1)
13. print(500 in list1)
```

Output:

```
False
False
False
True
True
True
```

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.

Code

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

Output:

```
John
David
James
Jonathan
```

Adding Elements to the List

The append() function in Python can add a new item to the List. In any case, the annex() capability can enhance the finish of the rundown.

Consider the accompanying model, where we take the components of the rundown from the client and print the rundown on the control center.

Code

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:10
Enter the item:32
Enter the item:56
Enter the item:81
```

```
Enter the item:2
Enter the item:34
Enter the item:65
Enter the item:09
Enter the item:66
Enter the item:12
Enter the item:18
printing the list items..
32 56 81 2 34 65 09 66 12 18
```

Removing Elements from the List

The `remove()` function in Python can remove an element from the List. To comprehend this idea, look at the example that follows.

Example -

Code

```
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.

1. `len()`
2. `max()`
3. `min()`

len()

It is used to calculate the length of the list.

Code

```
1. # size of the list
```

```
2. # declaring the list
3. list1 = [12, 16, 18, 20, 39, 40]
4. # finding length of the list
5. len(list1)
```

Output:

6

Max()

It returns the maximum element of the list

Code

```
1. # maximum of the list
2. list1 = [103, 675, 321, 782, 200]
3. # large element in the list
4. print(max(list1))
```

Output:

782

Min()

It returns the minimum element of the list

Code

```
1. # minimum of the list
2. list1 = [103, 675, 321, 782, 200]
3. # smallest element in the list
4. print(min(list1))
```

Output:

103

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

Example: 1- Create a program to eliminate the List's duplicate items.

Code

```
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- Compose a program to track down the amount of the component in the rundown.

Code

```
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  
In [8]:
```

Example: 3- Compose the program to find the rundown comprise of somewhere around one normal component.

Code

```
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 comma-separated group of items is called a Python triple. The ordering, settled items, and reiterations of a tuple are to some degree like those of a rundown, but in contrast to a rundown, a tuple is unchanging.

The main difference between the two is that we cannot alter the components of a tuple once they have been assigned. On the other hand, we can edit the contents of a list.

Example

1. ("Suzuki", "Audi", "BMW", "Skoda") is a tuple.

Features of Python Tuple

- o Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.
- o Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

Forming a Tuple:

All the objects-also known as "elements"-must be separated by a comma, enclosed in parenthesis (). Although parentheses are not required, they are recommended.

Any number of items, including those with various data types (dictionary, string, float, list, etc.), can be contained in a tuple.

Code

```
1. # Python program to show how to create a tuple
2. # Creating an empty tuple
3. empty_tuple = ()
4. print("Empty tuple: ", empty_tuple)
5.
6. # Creating tuple having integers
7. int_tuple = (4, 6, 8, 10, 12, 14)
8. print("Tuple with integers: ", int_tuple)
9.
10. # Creating a tuple having objects of different data types
11. mixed_tuple = (4, "Python", 9.3)
12. print("Tuple with different data types: ", mixed_tuple)
13.
14. # Creating a nested tuple
15. nested_tuple = ("Python", {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))
16. 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 necessary for the construction of multiples. This is known as triple pressing.

Code

```
1. # Python program to create a tuple without using parentheses
2. # Creating a tuple
3. tuple_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
4. # Displaying the tuple created
5. print(tuple_)
6. # Checking the data type of object tuple_
7. print(type(tuple_))
8. # Trying to modify tuple_
9. try:
10.     tuple_[1] = 4.2
11. except:
12.     print(TypeError)
```

Output:

```
(4, 5.7, 'Tuples', ['Python', 'Tuples'])
<class 'tuple'>
<class 'TypeError'>
```

The development of a tuple from a solitary part may be complex.

Essentially adding a bracket around the component is lacking. A comma must separate the element to be recognized as a tuple.

Code

```
1. # Python program to show how to create a tuple having a single element
2. single_tuple = ("Tuple")
3. print( type(single_tuple) )
4. # Creating a tuple that has only one element
5. single_tuple = ("Tuple",)
6. print( type(single_tuple) )
7. # Creating tuple without parentheses
8. single_tuple = "Tuple",
9. print( type(single_tuple) )
```

Output:

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

Accessing Tuple Elements

A tuple's objects can be accessed in a variety of ways.

Indexing

Indexing We can use the index operator [] to access an object in a tuple, where the index starts at 0.

The indices of a tuple with five items will range from 0 to 4. An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record. An index above four will be out of range in this scenario.

Because the index in Python must be an integer, we cannot provide an index of a floating data type or any other type. If we provide a floating index, the result will be TypeError.

The method by which elements can be accessed through nested tuples can be seen in the example below.

Code

```
1. # Python program to show how to access tuple elements
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Collection")
4. print(tuple_[0])
5. print(tuple_[1])
6. # trying to access element index more than the length of a tuple
7. try:
8.     print(tuple_[5])
9. except Exception as e:
10.    print(e)
11. # trying to access elements through the index of floating data type
12. try:
13.     print(tuple_[1.0])
14. except Exception as e:
15.     print(e)
16. # Creating a nested tuple
17. nested_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))
18.
19. # Accessing the index of a nested tuple
20. print(nested_tuple[0][3])
21. print(nested_tuple[1][1])
```

Output:

```
Python
Tuple
tuple index out of range
tuple indices must be integers or slices, not float
1
6
```

- o **Negative Indexing**

Python's sequence objects support negative indexing.

The last thing of the assortment is addressed by - 1, the second last thing by - 2, etc.

Code

```
1. # Python program to show how negative indexing works in Python tuples
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Collection")
4. # Printing elements using negative indices
5. print("Element at -1 index: ", tuple_[-1])
6. 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

Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues. Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.

To gain access to various tuple elements, we can use the slicing operator colon (:).

Code

```
1. # Python program to show how slicing works in Python tuples
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
4. # Using slicing to access elements of the tuple
5. print("Elements between indices 1 and 3: ", tuple_[1:3])
6. # Using negative indexing in slicing
7. print("Elements between indices 0 and -4: ", tuple_[:-4])
8. # Printing the entire tuple by using the default start and end values.
9. 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

A tuple's parts can't be modified, as was recently said. We are unable to eliminate or remove tuple components as a result.

However, the keyword `del` can completely delete a tuple.

Code

```
1. # Python program to show how to delete elements of a Python tuple
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
4. # Deleting a particular element of the tuple
5. try:
6.     del tuple_[3]
7.     print(tuple_)
8. except Exception as e:
9.     print(e)
10. # Deleting the variable from the global space of the program
11. del tuple_
12. # Trying accessing the tuple after deleting it
13. try:
14.     print(tuple_)
15. except Exception as e:
16.     print(e)
```

Output:

```
'tuple' object does not support item deletion
name 'tuple_' is not defined
```

Repetition Tuples in Python

Code

```
1. # Python program to show repetition in tuples
2. tuple_ = ('Python', "Tuples")
3. print("Original tuple is: ", tuple_)
4. # Repeting the tuple elements
5. tuple_ = tuple_ * 3
6. print("New tuple is: ", tuple_)
```

Output:

```
Original tuple is: ('Python', 'Tuples')
New tuple is: ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')
```

Tuple Methods

Like the list, Python Tuples is a collection of immutable objects. There are a few ways to work with tuples in Python. With some examples, this essay will go over these two approaches in detail.

The following are some examples of these methods.

- **Count () Method**

The times the predetermined component happens in the Tuple is returned by the count () capability of the Tuple.

Code

```
1. # Creating tuples
2. T1 = (0, 1, 5, 6, 7, 2, 2, 4, 2, 3, 2, 3, 1, 3, 2)
3. T2 = ('python', 'java', 'python', 'Tpoint', 'python', 'java')
4. # counting the appearance of 3
5. res = T1.count(2)
6. print('Count of 2 in T1 is:', res)
7. # counting the appearance of java
8. res = T2.count('java')
9. print('Count of Java in T2 is:', res)
```

Output:

```
Count of 2 in T1 is: 5
Count of java in T2 is: 2
```

Index() Method:

The Index() function returns the first instance of the requested element from the Tuple.

Parameters:

- The thing that must be looked for.
- Start: (Optional) the index that is used to begin the final (optional) search: The most recent index from which the search is carried out
- Index Method

Code

```
1. # Creating tuples
2. Tuple_data = (0, 1, 2, 3, 2, 3, 1, 3, 2)
3. # getting the index of 3
4. res = Tuple_data.index(3)
```

```
5. print('First occurrence of 1 is', res)
6. # getting the index of 3 after 4th
7. # index
8. res = Tuple_data.index(3, 4)
9. print('First occurrence of 1 after 4th index is:', res)
```

Output:

```
First occurrence of 1 is 2
First occurrence of 1 after 4th index is: 6
```

Tuple Membership Test

Utilizing the watchword, we can decide whether a thing is available in the given Tuple.

Code

```
1. # Python program to show how to perform membership test for tuples
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")
4. # In operator
5. print('Tuple' in tuple_)
6. print('Items' in tuple_)
7. # Not in operator
8. print('Immutable' not in tuple_)
9. print('Items' not in tuple_)
```

Output:

```
True
False
False
True
```

Iterating Through a Tuple

A for loop can be used to iterate through each tuple element.

Code

```
1. # Python program to show how to iterate over tuple elements
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable")
4. # Iterating over tuple elements using a for loop
5. for item in tuple_:
6.     print(item)
```

Output:

Python
Tuple
Ordered
Immutable

Changing a Tuple

Tuples, instead of records, are permanent articles.

This suggests that once the elements of a tuple have been defined, we cannot change them. However, the nested elements can be altered if the element itself is a changeable data type like a list.

Multiple values can be assigned to a tuple through reassignment.

Code

```
1. # Python program to show that Python tuples are immutable objects
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable", [1,2,3,4])
4. # Trying to change the element at index 2
5. try:
6.     tuple_[2] = "Items"
7.     print(tuple_)
8. except Exception as e:
9.     print( e )
10. # But inside a tuple, we can change elements of a mutable object
11. tuple_[-1][2] = 10
12. print(tuple_)
13. # Changing the whole tuple
14. tuple_ = ("Python", "Items")
15. print(tuple_)
```

Output:

```
'tuple' object does not support item assignment
('Python', 'Tuple', 'Ordered', 'Immutable', [1, 2, 10, 4])
('Python', 'Items')
```

The + operator can be used to combine multiple tuples into one. This phenomenon is known as concatenation.

We can also repeat the elements of a tuple a predetermined number of times by using the * operator. This is already demonstrated above.

The aftereffects of the tasks + and * are new tuples.

Code

1. # Python program to show how to concatenate tuples
2. # Creating a tuple
3. tuple_ = ("Python", "Tuple", "Ordered", "Immutable")
4. # Adding a tuple to the tuple_
5. print(tuple_ + (4, 5, 6))

Output:

```
('Python', 'Tuple', 'Ordered', 'Immutable', 4, 5, 6)
```

Tuples have the following advantages over lists:

- o Triples take less time than lists do.
- o Due to tuples, the code is protected from accidental modifications. It is desirable to store non-changing information in "tuples" instead of "records" if a program expects it.
- o A tuple can be used as a dictionary key if it contains immutable values like strings, numbers, or another tuple. "Lists" cannot be utilized as dictionary keys because they are mutable.

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, "JavaPoint", 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))`
- 4.
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'}

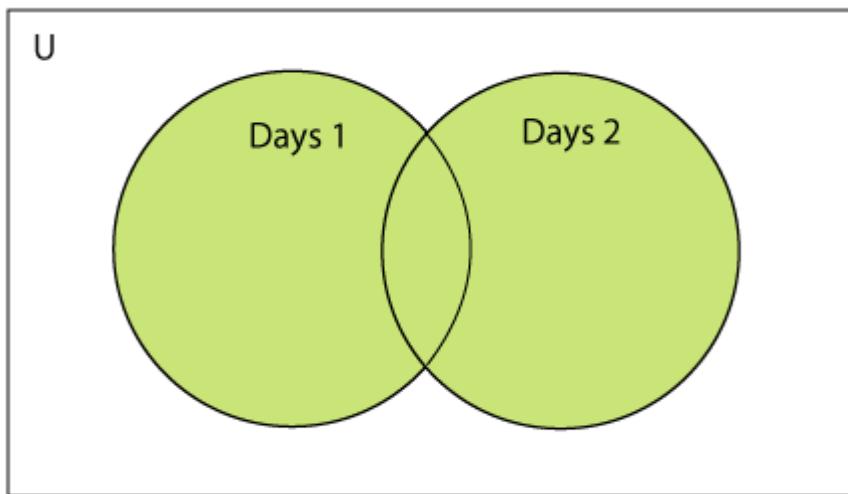
Removing items through remove() method...
Traceback (most recent call last):
  File "set.py", line 9, in 
    Months.remove("Jan")
KeyError: 'Jan'
```

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

To combine two or more sets into one set in Python, use the `union()` function. All of the distinctive characteristics from each combined set are present in the final set. As parameters, one or more sets may be passed to the `union()` function. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set containing all the different items from all the arguments if more than one set is supplied as an argument.



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'}
```

Now, we can also make the union of more than two sets using the `union()` function, for example:

Program:

1. `# Create three sets`
2. `set1 = {1, 2, 3}`
3. `set2 = {2, 3, 4}`
4. `set3 = {3, 4, 5}`
- 5.
6. `# Find the common elements between the three sets`
7. `common_elements = set1.union(set2, set3)`
- 8.
9. `# Print the common elements`
10. `print(common_elements)`

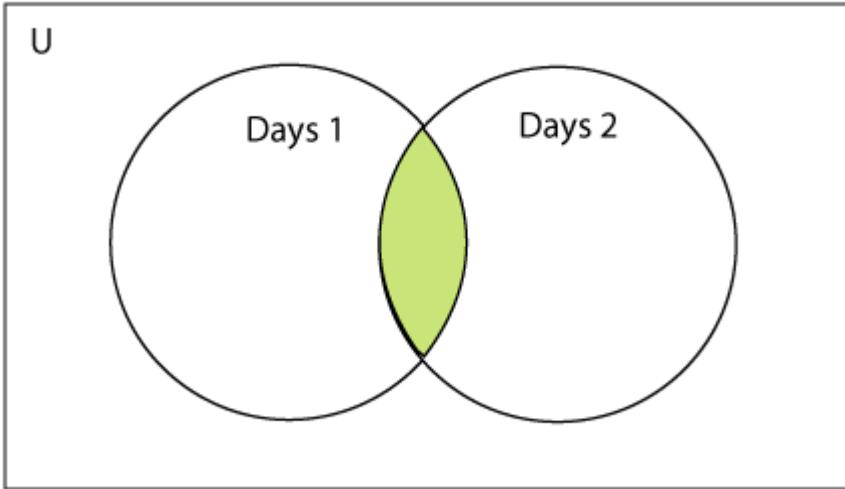
Output:

```
{1, 2, 3, 4, 5}
```

The intersection of two sets

To discover what is common between two or more sets in Python, apply the `intersection()` function. Only the items in all sets being compared are included in the final set. One or more sets can also be used as the `intersection()` function parameters. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set that only contains the elements in all the compared sets if multiple sets are supplied as arguments.

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}
```

Similarly, as the same as union function, we can perform the intersection of more than two sets at a time,

For Example:

Program

```
1. # Create three sets
2. set1 = {1, 2, 3}
3. set2 = {2, 3, 4}
4. set3 = {3, 4, 5}
5.
6. # Find the common elements between the three sets
7. common_elements = set1.intersection(set2, set3)
8.
9. # Print the common elements
10. print(common_elements)
```

Output:

```
{ 3 }
```

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.

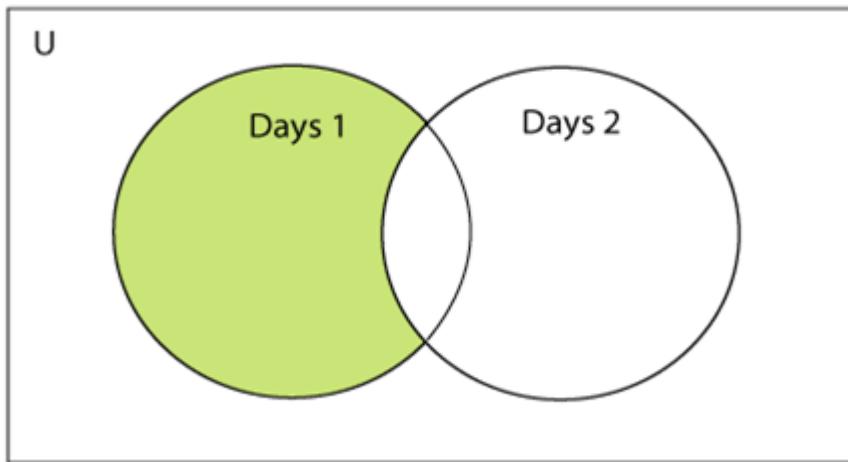
```
1. a = {"Devansh", "bob", "castle"}
2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
4.
5. a.intersection_update(b, c)
6.
7. print(a)
```

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

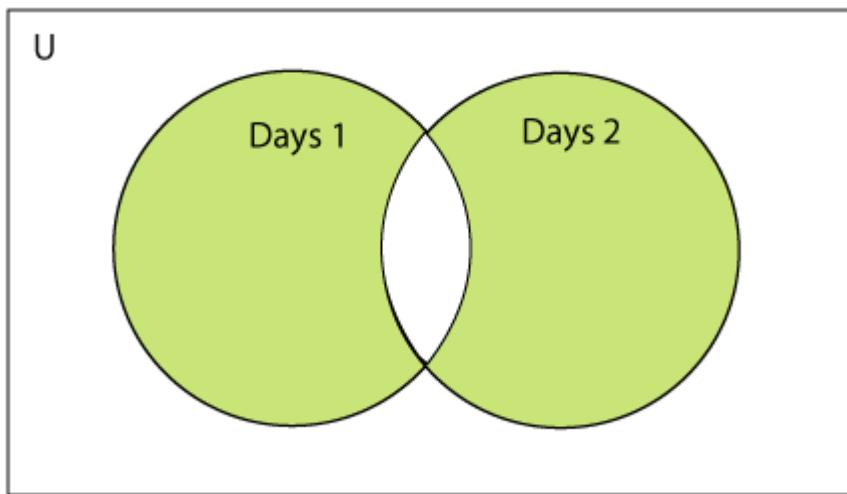
Output:

```
{'Thursday', 'Wednesday'}
```

Symmetric Difference of two sets

In Python, the symmetric Difference between set1 and set2 is the set of elements present in one set or the other but not in both sets. In other words, the set of elements is in set1 or set2 but not in their intersection.

The Symmetric Difference of two sets can be computed using Python's `symmetric_difference()` method. This method returns a new set containing all the elements in either but not in both. 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

In Python, you can compare sets to check if they are equal, if one set is a subset or superset of another, or if two sets have elements in common.

Here are the set comparison operators available in Python:

- `==`: checks if two sets have the same elements, regardless of their order.

- `!=`: checks if two sets are not equal.
- `<`: checks if the left set is a proper subset of the right set (i.e., all elements in the left set are also in the right set, but the right set has additional elements).
- `<=`: checks if the left set is a subset of the right set (i.e., all elements in the left set are also in the right set).
- `>`: checks if the left set is a proper superset of the right set (i.e., all elements in the right set are also in the left set, but the left set has additional elements).
- `>=`: checks if the left set is a superset of the right set (i.e., all elements in the right set are also in the left set).

Consider the following example.

1. Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday"}
3. Days3 = {"Monday", "Tuesday", "Friday"}
- 4.
5. #Days1 is the superset of Days2 hence it will print true.
6. **print** (Days1>Days2)
- 7.
8. #prints false since Days1 is not the subset of Days2
9. **print** (Days1<Days2)
- 10.
11. #prints false since Days2 and Days3 are not equivalent
12. **print** (Days2 == Days3)

Output:

```
True
False
False
```

FrozenSets

In Python, a frozen set is an immutable version of the built-in set data type. It is similar to a set, but its contents cannot be changed once a frozen set is created.

Frozen set objects are unordered collections of unique elements, just like sets. They can be used the same way as sets, except they cannot be modified. Because they are immutable, frozen set objects can be used as elements of other sets or dictionary keys, while standard sets cannot.

One of the main advantages of using frozen set objects is that they are hashable, meaning they can be used as keys in dictionaries or as elements of other sets. Their contents cannot change, so their hash values remain constant. Standard sets are not hashable because they can be modified, so their hash values can change.

Frozen set objects support many of the assets of the same operation, such as union, intersection, Difference, and symmetric Difference. They also support operations that do not modify the frozen set, such as len(), min(), max(), and in.

Consider the following example to create the frozen set.

1. Frosenset = frozenset([1,2,3,4,5])
2. **print**(type(Frosenset))
3. **print**("\nprinting the content of frozen set...")
4. **for** i **in** Frosenset:
5. **print**(i);
6. Frosenset.add(6) #gives an error since we cannot change the content of Frosenset 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>
    Frosenset.add(6) #gives an error since we can change the content of Frosenset
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. Frosenset = frozenset(Dictionary); #Frosenset will contain the keys of the dictionary
4. **print**(type(Frosenset))
5. **for** i **in** Frosenset:
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","Cameroon","Ricky","Donald"])
2. set2 = set(["Cameroon","Washington","Peter"])
3. set3 = set(["Peter"])
- 4.
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) | 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 | <code>copy()</code> | It returns a shallow copy of the set. |
| 4 | <code>difference_update(...)</code> | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | <code><u>discard(item)</u></code> | It removes the specified item from the set. |
| 6 | <code>intersection()</code> | 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 | <code>intersection_update(...)</code> | 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 | <code>Isdisjoint(...)</code> | Return True if two sets have a null intersection. |
| 9 | <code>Issubset(...)</code> | Report whether another set contains this set. |
| 10 | <code>Issuperset(...)</code> | Report whether this set contains another set. |
| 11 | <code><u>pop()</u></code> | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | <code><u>remove(item)</u></code> | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | <code>symmetric_difference(...)</code> | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | <code>symmetric_difference_update(...)</code> | Update a set with the symmetric difference of itself and another. |
| 15 | <code>union(...)</code> | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | <code>update()</code> | Update a set with the union of itself and others. |

Python Dictionary

Dictionaries are a useful data structure for storing data in Python because they are capable of imitating real-world data arrangements where a certain value exists for a given key.

The data is stored as key-value pairs using a Python dictionary.

- o This data structure is mutable

- o The components of dictionary were made using keys and values.
- o Keys must only have one component.
- o Values can be of any type, including integer, list, and tuple.

A dictionary is, in other words, a group of key-value pairs, where the values can be any Python object. The keys, in contrast, are immutable Python objects, such as strings, tuples, or numbers. Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

Creating the Dictionary

Curly brackets are the simplest way to generate a Python dictionary, although there are other approaches as well. With many key-value pairs surrounded in curly brackets and a colon separating each key from its value, the dictionary can be built. (:). The following provides the syntax for defining the dictionary.

Syntax:

1. Dict = {"Name": "Gayle", "Age": 25}

In the above dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Let's see an example to create a dictionary and print its content.

Code

1. Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"^TCS"}
2. **print**(type(Employee))
3. **print**("printing Employee data ")
4. **print**(Employee)

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'Johnny', 'Age': 32, 'salary': 26000, 'Company': TCS}
```

Python provides the built-in function **dict()** method which is also used to create the dictionary.

The empty curly braces {} is used to create empty dictionary.

Code

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")

```
4. print(Dict)
5.
6. # Creating a Dictionary
7. # with dict() method
8. Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
9. print("\nCreate Dictionary by using dict(): ")
10. print(Dict)
11.
12. # Creating a Dictionary
13. # with each item as a Pair
14. Dict = dict([(4, 'Rinku'), (2, Singh)])
15. print("\nDictionary with each item as a pair: ")
16. print(Dict)
```

Output

```
Empty Dictionary:
{}

Create Dictionary by using dict():
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}

Dictionary with each item as a pair:
{4: 'Rinku', 2: 'Singh'}
```

Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

Code

```
1. Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print("Name : %s" %Employee["Name"])
5. print("Age : %d" %Employee["Age"])
6. print("Salary : %d" %Employee["salary"])
7. print("Company : %s" %Employee["Company"])
```

Output

```
ee ["Company"]
Output
<class 'dict'>
printing Employee data ....
Name : Dev
Age : 20
```

```
Salary : 45000
Company : WIPRO
```

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 utilising the right keys allows you to change its values. Dict[key] = value and the value can both be modified. An existing value can also be updated using the update() method.

Note: *The value is updated if the key-value pair is already present in the dictionary. Otherwise, the dictionary's newly added keys.*

Let's see an example to update the dictionary values.

Example - 1:

Code

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

Output

```
Empty Dictionary:  
{ }  
  
Dictionary after adding 3 elements:  
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}  
  
Dictionary after adding 3 elements:  
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}  
  
Updated key value:  
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

Code

1. Employee = {"Name": "Dev", "Age": 20, "salary": 45000, "Company": "WIPRO"}
2. **print**(type(Employee))
3. **print**("printing Employee data ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company: ");
10. **print**("printing the new data");
11. **print**(Employee)

Output

```
<class 'dict'>  
printing Employee data ....  
Employee = {"Name": "Dev", "Age": 20, "salary": 45000, "Company": "WIPRO"} Enter  
the details of the new employee....  
Name: Sunny  
Age: 38  
Salary: 39000  
Company: Hcl  
printing the new data  
{'Name': 'Sunny', 'Age': 38, 'salary': 39000, 'Company': 'Hcl'}
```

Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

Code

1. Employee = {"Name": "David", "Age": 30, "salary": 55000, "Company": "WIPRO"}
2. **print**(type(Employee))
3. **print**("printing Employee data ")
4. **print**(Employee)

```
5. print("Deleting some of the employee data")
6. del Employee["Name"]
7. del Employee["Company"]
8. print("printing the modified information ")
9. print(Employee)
10. print("Deleting the dictionary: Employee");
11. del Employee
12. print("Lets try to print it again ");
13. print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'David', 'Age': 30, 'salary': 55000, 'Company': 'WIPRO'}
Deleting some of the employee data
printing the modified information
{'Age': 30, 'salary': 55000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined.
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

Deleting Elements using pop() Method

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys. The pop() method is one of the ways to get rid of elements from a dictionary. In this post, we'll talk about how to remove items from a Python dictionary using the pop() method.

The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

Code

```
1. # Creating a Dictionary
2. Dict1 = {1: 'JavaTpoint', 2: 'Educational', 3: 'Website'}
3. # Deleting a key
4. # using pop() method
5. pop_key = Dict1.pop(2)
6. print(Dict1)
```

Output

```
{1: 'JavaTpoint', 3: 'Website'}
```

Additionally, Python offers built-in functions `popitem()` and `clear()` for removing dictionary items. In contrast to the `clear()` method, which removes all of the elements from the entire dictionary, `popitem()` removes any element from a dictionary.

Iterating Dictionary

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

Example 1

Code

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

Output

```
Name
Age
salary
Company
```

Example 2

Code

1. `#for loop to print all the values of the dictionary`
2. `Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"} for x in Employee:`
3. `print(Employee[x])`

Output

```
John
29
25000
WIPRO
```

Example - 3

Code

1. `#for loop to print the values of the dictionary by using values() method.`
2. `Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}`
3. `for x in Employee.values():`
4. `print(x)`

Output

```
John  
29  
25000  
WIPRO
```

Example 4

Code

1. `#for loop to print the items of the dictionary by using items() method`
2. `Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}`
3. `for x in Employee.items():`
4. `print(x)`

Output

```
('Name', 'John')  
(('Age', 29)  
(('salary', 25000)  
(('Company', 'WIPRO')
```

Properties of Dictionary Keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

Code

1. `Employee={"Name":"John","Age":29,"Salary":25000,"Company":"WIPRO","Name":`
2. `"John"}`
3. `for x,y in Employee.items():`
4. `print(x,y)`

Output

```
Name John  
Age 29  
Salary 25000  
Company WIPRO
```

2. The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however mutable objects like lists cannot be used as the key in a dictionary.

Consider the following example.

Code

- Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO", [100,201,301] : "Department ID"}
- for** x,y **in** Employee.items():
- print**(x,y)

Output

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in 
    Employee      =      {"Name":           "John",           "Age":           29,
"salary":26000,"Company":"WIPRO", [100,201,301] :"Department ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions

A function is a method that can be used on a construct to yield a value. Additionally, the construct is unaltered. A few of the Python methods can be combined with a Python dictionary.

The built-in Python dictionary methods are listed below, along with a brief description.

- o **len()**

The dictionary's length is returned via the len() function in Python. The string is lengthened by one for each key-value pair.

Code

- dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
- len(dict)

Output

```
4
```

- o **any()**

Like how it does with lists and tuples, the any() method returns True indeed if one dictionary key does have a Boolean expression that evaluates to True.

Code

- dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
- any({":",":",'3':"})

Output

```
True
```

- o **all()**

Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

Code

```
1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}  
2. all({1:"",2:"",3:""})
```

Output

```
False
```

- o **sorted()**

Like it does with lists and tuples, the sorted() method returns an ordered series of the dictionary's keys. The ascending sorting has no effect on the original Python dictionary.

Code

```
1. dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}  
2. sorted(dict)
```

Output

```
[ 1, 5, 7, 8]
```

Built-in Dictionary methods

The built-in python dictionary methods along with the description and Code are given below.

- o **clear()**

It is mainly used to delete all the items of the dictionary.

Code

```
1. # dictionary methods  
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}  
3. # clear() method  
4. dict.clear()  
5. print(dict)
```

Output

```
{ }
```

- o **copy()**

It returns a shallow copy of the dictionary which is created.

Code

```
1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # copy() method
4. dict_demo = dict.copy()
5. print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

- **pop()**

It mainly eliminates the element using the defined key.

Code

```
1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # pop() method
4. dict_demo = dict.copy()
5. x = dict_demo.pop(1)
6. print(x)
```

Output

```
{2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

popitem()

removes the most recent key-value pair entered

Code

```
1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # popitem() method
4. dict_demo.popitem()
5. print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}
```

- **keys()**

It returns all the keys of the dictionary.

Code

1. `# dictionary methods`
2. `dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}`
3. `# keys() method`
4. `print(dict_demo.keys())`

Output

```
dict_keys([1, 2, 3, 4, 5])
```

- o `items()`

It returns all the key-value pairs as a tuple.

Code

1. `# dictionary methods`
2. `dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}`
3. `# items() method`
4. `print(dict_demo.items())`

Output

```
dict_items([(1, 'Hcl'), (2, 'WIPRO'), (3, 'Facebook'), (4, 'Amazon'), (5, 'Flipkart')])
```

- o `get()`

It is used to get the value specified for the passed key.

Code

1. `# dictionary methods`
2. `dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}`
3. `# get() method`
4. `print(dict_demo.get(3))`

Output

```
Facebook
```

- o `update()`

It mainly updates all the dictionary by adding the key-value pair of dict2 to this dictionary.

Code

1. `# dictionary methods`
2. `dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}`

3. `# update() method`
4. `dict_demo.update({3: "TCS"})`
5. `print(dict_demo)`

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'TCS'}
```

- o `values()`

It returns all the values of the dictionary with respect to given input.

Code

1. `# dictionary methods`
2. `dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}`
3. `# values() method`
4. `print(dict_demo.values())`

Output

```
dict_values(['Hcl', 'WIPRO', 'TCS'])
```

Python Functions

This tutorial will go over the fundamentals of Python functions, including what they are, their syntax, their primary parts, return keywords, and significant types. We'll also look at some examples of Python function definitions.

What are Python Functions?

A collection of related assertions that carry out a mathematical, analytical, or evaluative operation is known as a function. An assortment of proclamations called Python Capabilities returns the specific errand. Python functions are necessary for intermediate-level programming and are easy to define. Function names meet the same standards as variable names do. The objective is to define a function and group-specific frequently performed actions. Instead of repeatedly creating the same code block for various input variables, we can call the function and reuse the code it contains with different variables.

Client-characterized and worked-in capabilities are the two primary classes of capabilities in Python. It aids in maintaining the program's uniqueness, conciseness, and structure.

Advantages of Python Functions

Pause We can stop a program from repeatedly using the same code block by including functions.

- o Once defined, Python functions can be called multiple times and from any location in a program.

- Our Python program can be broken up into numerous, easy-to-follow functions if it is significant.
- The ability to return as many outputs as we want using a variety of arguments is one of Python's most significant achievements.
- However, Python programs have always incurred overhead when calling functions.

However, calling functions has always been overhead in a Python program.

Syntax

1. `# An example Python Function`
2. `def function_name(parameters):`
3. `# code block`

The accompanying components make up to characterize a capability, as seen previously.

- The start of a capability header is shown by a catchphrase called def.
- `function_name` is the function's name, which we can use to distinguish it from other functions. We will utilize this name to call the capability later in the program. Name functions in Python must adhere to the same guidelines as naming variables.
- Using `parameters`, we provide the defined function with arguments. Notwithstanding, they are discretionary.
- A colon (`:`) marks the function header's end.
- We can utilize a documentation string called docstring in the short structure to make sense of the reason for the capability.
- Several valid Python statements make up the function's body. The entire code block's indentation depth-typically four spaces-must be the same.
- A `return` expression can get a value from a defined function.

Illustration of a User-Defined Function

We will define a function that returns the argument number's square when called.

1. `# Example Python Code for User-Defined function`
2. `def square(num):`
3. `"""`
4. `This function computes the square of the number.`
5. `"""`
6. `return num**2`
7. `object_ = square(6)`
8. `print("The square of the given number is: ", object_)`

Output:

```
The square of the given number is: 36
```

Calling a Function

Calling a Function To define a function, use the def keyword to give it a name, specify the arguments it must receive, and organize the code block.

When the fundamental framework for a function is finished, we can call it from anywhere in the program. An illustration of how to use the a_function function can be found below.

```
1. # Example Python Code for calling a function
2. # Defining a function
3. def a_function( string ):
4.     "This prints the value of length of string"
5.     return len(string)
6.
7. # Calling the function we defined
8. print( "Length of the string Functions is: ", a_function( "Functions" ) )
9. 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. Pass by Value

In the Python programming language, all parameters are passed by reference. It shows that if we modify the worth of contention within a capability, the calling capability will similarly mirror the change. For instance,

Code

```
1. # Example Python Code for Pass by Reference vs. Value
2. # defining the function
3. def square( item_list ):
4.     """This function will find the square of items in the list"""
5.     squares = [ ]
6.     for i in item_list:
7.         squares.append( i**2 )
8.     return squares
9.
10. # calling the defined function
```

```
11. my_list = [17, 52, 8];
12. my_result = square( my_list )
13. print( "Squares of the list are: ", my_result )
```

Output:

```
Squares of the list are: [289, 2704, 64]
```

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

1) Default Arguments

A default contention is a boundary that takes as information a default esteem, assuming that no worth is provided for the contention when the capability is called. The following example demonstrates default arguments.

Code

```
1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. def function( n1, n2 = 20 ):
4.     print("number 1 is: ", n1)
5.     print("number 2 is: ", n2)
6.
7.
8. # Calling the function and passing only one argument
9. print( "Passing only one argument" )
10. function(30)
11.
12. # Now giving two arguments to the function
13. print( "Passing two arguments" )
14. function(50,30)
```

Output:

```
Passing only one argument
number 1 is: 30
number 2 is: 20
```

```
Passing two arguments
number 1 is: 50
number 2 is: 30
```

2) Keyword Arguments

Keyword arguments are linked to the arguments of a called function. While summoning a capability with watchword contentions, the client might tell whose boundary esteem it is by looking at the boundary name.

We can eliminate or orchestrate specific contentions in an alternate request since the Python translator will interface the furnished watchwords to connect the qualities with its boundaries. One more method for utilizing watchwords to summon the capability() strategy is as per the following:

Code

```
1. # Python code to demonstrate the use of keyword arguments
2. # Defining a function
3. def function( n1, n2 ):
4.     print("number 1 is: ", n1)
5.     print("number 2 is: ", n2)
6.
7. # Calling function and passing arguments without using keyword
8. print( "Without using keyword" )
9. function( 50, 30)
10.
11. # Calling function and passing arguments using keyword
12. print( "With using keyword" )
13. function( n2 = 50, n1 = 30)
```

Output:

```
Without using keyword
number 1 is: 50
number 2 is: 30
With using keyword
number 1 is: 30
number 2 is: 50
```

3) Required Arguments

Required arguments are those supplied to a function during its call in a predetermined positional sequence. The number of arguments required in the method call must be the same as those provided in the function's definition.

We should send two contentions to the capability() all put together; it will return a language structure blunder, as seen beneath.

Code

```
1. # Python code to demonstrate the use of default arguments
2. # Defining a function
3. def function( n1, n2 ):
4.     print("number 1 is: ", n1)
5.     print("number 2 is: ", n2)
6.
7. # Calling function and passing two arguments out of order, we need num1 to be 20 and
8. # num2 to be 30
8. print( "Passing out of order arguments" )
9. function( 30, 20 )
10.
11. # Calling function and passing only one argument
12. print( "Passing only one argument" )
13. try:
14.     function( 30 )
15. except:
16.     print( "Function needs two positional arguments" )
```

Output:

```
Passing out of order arguments
number 1 is: 30
number 2 is: 20
Passing only one argument
Function needs two positional arguments
```

4) Variable-Length Arguments

We can involve unique characters in Python capabilities to pass many contentions. However, we need a capability. This can be accomplished with one of two types of characters:

"args" and "kwargs" refer to arguments not based on keywords.

To help you understand arguments of variable length, here's an example.

Code

```
1. # Python code to demonstrate the use of variable-length arguments
2. # Defining a function
3. def function( *args_list ):
4.     ans = []
5.     for l in args_list:
6.         ans.append( l.upper() )
```

```
7.     return ans
8. # Passing args arguments
9. object = function('Python', 'Functions', 'tutorial')
10. print( object )
11.
12. # defining a function
13. def function( **kargs_list ):
14.     ans = []
15.     for key, value in kargs_list.items():
16.         ans.append([key, value])
17.     return ans
18. # Paasing kwargs arguments
19. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
20. print(object)
```

Output:

```
['PYTHON', 'FUNCTIONS', 'TUTORIAL']
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]
```

return Statement

When a defined function is called, a return statement is written to exit the function and return the calculated value.

Syntax:

1. **return** < expression to be returned as output >

The return statement can be an argument, a statement, or a value, and it is provided as output when a particular job or function is finished. A declared function will return an empty string if no return statement is written.

A return statement in Python functions is depicted in the following example.

Code

```
1. # Python code to demonstrate the use of return statements
2. # Defining a function with return statement
3. def square( num ):
4.     return num**2
5.
6. # Calling function and passing arguments.
7. print( "With return statement" )
8. print( square( 52 ) )
```

```
9.  
10. # Defining a function without return statement  
11. def square( num ):  
12.     num**2  
13.  
14. # Calling function and passing arguments.  
15. print( "Without return statement" )  
16. print( square( 52 ) )
```

Output:

```
With return statement  
2704  
Without return statement  
None
```

The Anonymous Functions

Since we do not use the `def` keyword to declare these kinds of Python functions, they are unknown. The `lambda` keyword can define anonymous, short, single-output functions.

Arguments can be accepted in any number by `lambda` expressions; However, the function only produces a single value from them. They cannot contain multiple instructions or expressions. Since `lambda` needs articulation, a mysterious capability can't be straightforwardly called to print.

Lambda functions can only refer to variables in their argument list and the global domain name because they contain their distinct local domain.

In contrast to inline expressions in C and C++, which pass function stack allocations at execution for efficiency reasons, `lambda` expressions appear to be one-line representations of functions.

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. `# Python code to demonstrate anaymous functions`
2. `# Defining a function`
3. `lambda_ = lambda argument1, argument2: argument1 + argument2;`
- 4.

5. `# Calling the function and passing values`
6. `print("Value of the function is : ", lambda_(20, 30))`
7. `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

A variable's scope refers to the program's domain wherever it is declared. A capability's contentions and factors are not external to the characterized capability. They only have a local domain as a result.

The length of time a variable remains in RAM is its lifespan. The lifespan of a function is the same as the lifespan of its internal variables. When we exit the function, they are taken away from us. As a result, the value of a variable in a function does not persist from previous executions.

An easy illustration of a function's scope for a variable can be found here.

Code

1. `# Python code to demonstrate scope and lifetime of variables`
2. `#defining a function to print a number.`
3. `def number():`
4. `num = 50`
5. `print("Value of num inside the function: ", num)`
6.
7. `num = 10`
8. `number()`
9. `print("Value of num outside the function:", num)`

Output:

```
Value of num inside the function: 50
Value of num outside the function: 10
```

Here, we can see that the initial value of num is 10. Even though the function number() changed the value of num to 50, the value of num outside of the function remained unchanged.

This is because the capability's interior variable num is not quite the same as the outer variable (nearby to the capability). Despite having a similar variable name, they are separate factors with discrete extensions.

Factors past the capability are available inside the capability. The impact of these variables is global. We can retrieve their values within the function, but we cannot alter or change them. The value of a variable can be changed outside of the function if it is declared global with the keyword global.

Python Capability inside Another Capability

Capabilities are viewed as top-of-the-line objects in Python. First-class objects are treated the same everywhere they are used in a programming language. They can be stored in built-in data structures, used as arguments, and in conditional expressions. If a programming language treats functions like first-class objects, it is considered to implement first-class functions. Python lends its support to the concept of First-Class functions.

A function defined within another is called an "inner" or "nested" function. The parameters of the outer scope are accessible to inner functions. Internal capabilities are developed to cover them from the progressions outside the capability. Numerous designers see this interaction as an embodiment.

Code

```
1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. def word():
4.     string = 'Python functions tutorial'
5.     x = 5
6.     def number():
7.         print( string )
8.         print( x )
9.
10.    number()
11. word()
```

Output:

```
Python functions tutorial
5
```

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))
4.
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))
4.
5. # all values false
6. k = [0, False]
7. print(all(k))
8.
9. # one false value
10. k = [1, 3, 7, 0]
11. print(all(k))
12.
13. # one true value
14. k = [0, False, 5]
15. print(all(k))
16.
```

```
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 bytarray() 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)
3.
4. s = sum([1, 2, 4], 10)
5. print(s)
```

Output:

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, 0]
2. **print**(any(l))
- 3.
4. l = [0, False]
5. **print**(any(l))
- 6.
7. l = [0, False, 5]
8. **print**(any(l))
- 9.
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))
- 3.
4. otherText = 'Pythön is interesting'
5. **print**(ascii(otherText))
- 6.
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."
- 2.
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))
- 3.
4. # for floats

```
5. print(float(8.19))
6.
7. # for string floats
8. print(float("-24.27"))
9.
10. # for string floats with whitespaces
11. print(float(" -17.19\n"))
12.
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
2.
3. # integer
4. print(format(123, "d"))
5.
6. # float arguments
7. print(format(123.4567898, "f"))
8.
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')
3.
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"
4.
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
- 2.
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))**
- 3.
4. l = [0, False]
5. **print(any(l))**
- 6.
7. l = [0, False, 5]
8. **print(any(l))**
- 9.
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]
- 3.

```
4. listIter = iter(list)
5.
6. # prints '1'
7. print(next(listIter))
8.
9. # prints '2'
10. print(next(listIter))
11.
12. # prints '3'
13. print(next(listIter))
14.
15. # prints '4'
16. print(next(listIter))
17.
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())
```

```
3.  
4. # string  
5. String = 'abcde'  
6. print(list(String))  
7.  
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()  
3.  
4. def localsPresent():  
5.     present = True  
6.     return locals()  
7.  
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  
3.  
4. numbers = (1, 2, 3, 4)  
5. result = map(calculateAddition, numbers)  
6. print(result)  
7.  
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')  
3.  
4. mv = memoryview(randomByteArray)  
5.  
6. # access the memory view's zeroth index  
7. print(mv[0])  
8.  
9. # It create byte from memory view  
10. print(bytes(mv[0:2]))  
11.  
12. # It create list from memory view  
13. print(list(mv[0:3]))
```

Output:

```
65  
b'AB'
```

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()`
- 2.
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:

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 = ""`
- 4.
5. `def __init__(self, id, name):`
6. `self.id = id`
7. `self.name = name`
- 8.
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`
- 7.
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'))`
- 3.
4. `# Code point of an alphabet`
5. `print(ord('R'))`
- 6.
7. `# Code point of a character`
8. `print(ord('&'))`

Output:

```
56  
82
```

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))
- 3.
4. # negative x, positive y
5. **print**(pow(-4, 2))
- 6.
7. # positive x, negative y ($x^{**}-y$)
8. **print**(pow(4, -2))
- 9.
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.")
- 2.
3. x = 7
4. # Two objects passed
5. **print**("x =", x)
- 6.
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)))`
- 3.
4. `# using the range(stop)`
5. `print(list(range(4)))`
- 6.
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)))`
- 4.
5. `# for tuple`
6. `Tuple = ('J', 'a', 'v', 'a')`
7. `print(list(reversed(Tuple)))`
- 8.
9. `# for range`

```
10. Range = range(8, 12)
11. print(list(reversed(Range)))
12.
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))
3.
4. # for floating point
5. print(round(10.8))
6.
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)
```

```
4.  
5. class Square(Rectangle):  
6.     def __init__(self):  
7.         Rectangle.__init__('square')  
8.  
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)  
3.  
4. # creating a tuple from a list  
5. t2 = tuple([1, 6, 9])  
6. print('t2=', t2)  
7.  
8. # creating a tuple from a string  
9. t1 = tuple('Java')  
10. print('t1=', t1)
```

```
11.  
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))  
3.  
4. Dict = {4: 'four', 5: 'five'}  
5. print(type(Dict))  
6.  
7. class Python:  
8.     a = 0  
9.  
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
5.
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']
3.
4. # No iterables are passed
5. result = zip()
6.
7. # Converting itertor to list
8. resultList = list(result)
9. print(resultList)
10.
11. # Two iterables are passed
12. result = zip(numList, strList)
13.
14. # Converting itertor to set
15. resultSet = set(result)
16. print(resultSet)
```

Output:

```
[]  
{(5, 'five'), (4, 'four'), (6, 'six')}
```

Python Lambda Functions

This tutorial will study anonymous, commonly called lambda functions in Python. A lambda function can take n number of arguments at a time. But it returns only one argument at a time. We will 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

The syntax of the Lambda Function is given below -

1. **lambda** arguments: expression

This function accepts any count of inputs but only evaluates and returns one expression. That means it takes many inputs but returns only one output.

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

Here we share some examples of lambda functions in Python for learning purposes. **Program Code 1:**

Now we gave an example of a lambda function that adds 4 to the input number is shown below.

1. # Code to demonstrate how we can use a lambda function for adding 4 numbers
2. add = **lambda** num: num + 4
3. **print**(add(6))

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

10

Here we explain the above code. 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 written using the add function, and the code is given below: **Program Code 2:**

Now we gave an example of a lambda function that adds 4 to the input number using the add function. The code is shown below -

1. **def** add(num):
2. **return** num + 4
3. **print**(add(6))

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

10

Program Code 3:

Now we gave an example of a lambda function that multiply 2 numbers and return one result. The code is shown below -

1. a = **lambda** x, y : (x * y)
2. **print**(a(4, 5))

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

20

Program Code 4:

Now we gave another example of a lambda function that adds 2 numbers and return one result. The code is shown below -

1. a = **lambda** x, y, z : (x + y + z)
2. **print**(a(4, 5, 5))

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

14

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:

Program Code:

```
1. # Python code to show the reciprocal of the given number to highlight the difference bet  
2. ween def() and lambda().  
3. 2. def reciprocal( num ):  
4.     return 1 / num  
5.  
6. 5. lambda_reciprocal = lambda num: 1 / num  
7.  
8. 7. # using the function defined by def keyword  
9. 8. print( "Def keyword: ", reciprocal(6) )  
10.  
11. 10. # using the function defined by lambda keyword  
12. 11. print( "Lambda keyword: ", lambda_reciprocal(6) )
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
Def keyword:  0.1666666666666666  
Lambda keyword:  0.1666666666666666
```

Explanation:

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.

Program Code:

Here we give an example of lambda function with filter() in Python. The code is given below -

1. **# This code used to filter the odd numbers from the given list**
2. `list_ = [35, 12, 69, 55, 75, 14, 73]`
3. `odd_list = list(filter(lambda num: (num % 2 != 0) , list_))`
4. `print('The list of odd number is:',odd_list)`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
The list of odd number is: [35, 69, 55, 75, 73]
```

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.

Program Code:

Here we give an example of lambda function with map() in Python. Then code is given below -

1. **#Code to calculate the square of each number of a list using the map() function**
2. `numbers_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]`
3. `squared_list = list(map(lambda num: num ** 2 , numbers_list))`
4. `print('Square of each number in the given list:' ,squared_list)`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
Square of each number in the given list: [4, 16, 25, 1, 9, 49, 64, 81, 100]
```

Using Lambda Function with List Comprehension

In this instance, we will apply the lambda function combined with list comprehension and the lambda keyword with a for loop. Using the Lambda Function with List Comprehension, we can print the square value from 0 to 10. For printing the square value from 0 to 10, we create a loop range from 0 to 11.

Program Code:

Here we give an example of lambda function with List Comprehension in Python. Then code is given below -

1. **#Code to calculate square of each number of lists using list comprehension**
2. `squares = [lambda num = num: num ** 2 for num in range(0, 11)]`
3. **for** square **in** squares:
4. `print("The square value of all numbers from 0 to 10:",square(), end = " ")`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
The square value of all numbers from 0 to 10: 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. In the program code below, we check which number is greater than the given two numbers using the if-else block.

Program Code:

Here we give an example of a lambda function with an if-else block in Python. The code is given below -

1. **# Code to use lambda function with if-else**
2. `Minimum = lambda x, y : x if (x < y) else y`
3. `print('The greater number is:', Minimum(35, 74))`

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
The greater number is: 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. We are sorting every sub-list from the given list in the below program. Let us use lambda to discover the third largest number from every sub-list.

Program Code:

Here we give an example of lambda function with Multiple Statements in Python. The code is given below -

1. `# Code to print the third largest number of the given list using the lambda function`
2.
3. `my_List = [[3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5]]`
4. `# sorting every sublist of the above list`
5. `sort_List = lambda num : (sorted(n) for n in num)`
6. `# Getting the third largest number of the sublist`
7. `third_Largest = lambda num, func : [l[len(l) - 2] for l in func(num)]`
8. `result = third_Largest(my_List, sort_List)`
9. `print('The third largest number from every sub list is:', result)`

Output:

Now we compile the above code, in python and after successful compilation, we run it. Then the output is given below -

```
The third largest number from every sub list is: [6, 54, 5]
```

Conclusion:

So, in this, we discuss the Lambda function in Python. A lambda function can take n number of arguments at a time. But it returns only one argument at a time. Here we discuss some lambda functions with the program code in Python, and we also share some examples of them. Here we discuss the Lambda function with the list, map, filter, multiple statements, if-else, and some basic programs of lambda function in Python.

Python File Handling

Introduction:

In this tutorial, we are discussing Python file handling. Python supports the file-handling process. Till now, we were taking the input from the console and writing it back to the console to interact with the user. Users can easily handle the files, like read and write the files in Python. In another programming language, the file-handling process is lengthy and complicated. But we know Python is an easy programming language. So, like other things, file handling is also effortless and short in Python.

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.

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 like a comma (,) or a newline character. Python executes the code line by line. So, it works in one line and then asks the interpreter to start the new line again. This is a continuous process in Python.

Hence, a file operation can be done in the following order.

- o Open a file
- o Read or write - Performing operation
- o Close the file

Opening a file

A file operation starts with the file opening. At first, open the File then Python will start the operation. File opening is done with the `open()` function in Python. This function will accept two arguments, file name and access mode in which the file is accessed. When we use the `open()` function, that time we must be specified the mode for which the File is opening. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

The syntax for opening a file in Python is given below -

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 | r means to read. So, it opens a file for read-only operation. 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.

Program code for read mode:

It is a read operation in Python. We open an existing file with the given code and then read it. The code is given below -

1. #opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
- 3.
4. if fileptr:
5. print("file is opened successfully")

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
<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

Program code for Write Mode:

It is a write operation in Python. We open an existing file using the given code and then write on it. The code is given below -

1. file = open('file.txt','w')
2. file.write("Here we write a command")
3. file.write("Hello users of JAVATPOINT")
4. file.close()

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

1. > Hi
2. ERROR!
3. Traceback (most recent call last):
4. File "<stdin>", line 1, in <module>
5. NameError: name 'Hi' is not defined

The close() Method

The close method used to terminate the program. 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. Earlier use of the close() method can cause the loss of destroyed some information that you want to write in your File.

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

Syntax

The syntax for closing a file in Python is given below -

1. fileobject.close()

Consider the following example.

Program code for Closing Method:

Here we write the program code for the closing method in Python. The code is given below -

1. **# opens the file file.txt in read mode**
2. fileptr = open("file.txt","r")
- 3.
4. **if** fileptr:
5. **print**("The existing file is opened successfully in Python")
- 6.
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.

Syntax:

The syntax of with statement of a file in Python 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.

Program code 1 for with statement:

Here we write the program code for with statement in Python. The code is given below -

1. with open("file.txt", 'r') as f:
2. content = f.read();
3. print(content)

Program code 2 for with statement:

Here we write the program code for with statement in Python. The code is given below -

1. with open("file.txt", "H") as f:
2. A = f.write("Hello Coders")
3. Print(A)

Writing the file

To write some text to a file, we need to open the file using the open method and then we can use the write method for writing in this File. If we want to open a file that does not exist in our system, it creates a new one. On the other hand, if the File exists, then erase the past content and add new content to this File. the It is done by 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.

Program code 1 for Write Method:

Here we write the program code for write method in Python. The code is given below -

1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")

```
3.  
4. # appending the content to the file  
5. fileptr.write("Python is the modern programming language. It is done any kind of pro  
gram in shortest way.")  
6.  
7. # closing the opened the file  
8. fileptr.close()
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
File2.txt  
Python is the modern programming language. It is done any kind of program in  
shortest way.
```

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

Program code 2 for Write Method:

Here we write the program code for write method in Python. The code is given below -

```
1. with open('test1.txt', 'w') as file2:  
2.     file2.write('Hello coders')  
3.     file2.write('Welcome to javaTpoint')
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Hello coders  
Welcome to javaTpoint
```

Program code 3 for Write Method:

Here we write the program code for write method in Python. The code is given below -

```
1. #open the file.txt in write mode.  
2. fileptr = open("file2.txt","a")  
3.  
4. #overwriting the content of the file  
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")  
6.  
7. #closing the opened file
```

- fileptr.close()

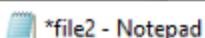
Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Python is the modern day language. It makes things so simple.
```

```
It is the fastest growing 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.

Syntax:

The syntax of **read()** method of a file in Python is given below -

- 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.

Program code for **read()** Method:

Here we write the program code for **read()** method in Python. The code is given below -

- #open the file.txt in read mode. causes error if no such file exists.
- fileptr = open("file2.txt","r")
- #stores all the data of the file into the variable content
- 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:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
<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. So, it only prints 'Python is'. For read the whole file contents, the code is given below -

1. `content = fileptr.read()`
2. `print(content)`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Python is the modern-day language. It makes things so simple.
It is the fastest-growing programming language Python has easy an syntax and
user-friendly interaction.
```

Read file through for loop

We can use **read()** method when we open the file. Read method is also done through the for loop. We can read the file using for loop. Consider the following example.

Program code 1 for Read File using For Loop:

Here we give an example of read file using for loop. The code is given below -

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:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Python is the modern day language.
```

```
It makes things so simple.
```

```
Python has easy syntax and user-friendly interaction.
```

Program code 2 for Read File using For Loop:

Here we give an example of read file using for loop. The code is given below -

1. `A = ["Hello\n", "Coders\n", "JavaTpoint\n"]`
2. `f1 = open('myfile.txt', 'w')`
3. `f1.writelines(A)`
4. `f1.close()`
5. `f1 = open('myfile.txt', 'r')`
6. `Lines = f1.read()`
7. `count = 0`
8. `for line in Lines:`
9. `count += 1`
10. `print("Line{}: {}".format(count, line.strip()))`

Output:

```
Line1: H
Line2: e
Line3: l
Line4: l
Line5: o
Line6:
Line7: C
Line8: o
Line9: d
Line10: e
Line11: r
Line12: s
Line13:
Line14: J
Line15: a
Line16: v
Line17: a
Line18: T
Line19: p
Line20: o
Line21: i
```

```
Line22: n  
Line23: t  
Line24:
```

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.

Here we give the example of reading the lines using the readline() function in Python. The code is given below -

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:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
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. That means, if you called readline() function n times in your program, then it read n number of lines from the file. This is the uses of readline() function in Python. 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:

Here we give the example of reading the lines using the readline() function in Python. The code is given below -

1. `#open the file.txt in read mode. causes error if no such file exists.`

```
2. fileptr = open("file2.txt","r");
3.
4. #stores all the data of the file into the variable content
5. content = fileptr.readlines()
6.
7. #prints the content of the file
8. print(content)
9.
10. #closes the opened file
11. fileptr.close()
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']}
```

Example 3:

Here we give the example of reading the lines using the readline() function in Python. The code is given below -

```
1. A = ["Hello\n", "Coders\n", "JavaTpoint\n"]
2. f1 = open('myfile.txt', 'w')
3. f1.writelines(A)
4. f1.close()
5. f1 = open('myfile.txt', 'r')
6. Lines = f1.readlines()
7. count = 0
8. for line in Lines:
9.     count += 1
10.    print("Line{}: {}".format(count, line.strip()))
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Line1: Hello
Line2: Coders
Line3: JavaTpoint
```

Creating a new file

The new file can be created by using one of the following access modes with the function open(). The open() function used so many parameters. The syntax of it is given below -

```
file = open(path_to_file, mode)
```

x, a and w is the modes of open() function. The uses of these modes are given below -

x: it creates a new file with the specified name. It causes an error if 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.

Program code1 for Creating a new file:

Here we give an example for creating a new file in Python. For creates a file, we have to used the open() method. The code is given below -

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:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

Program code2 for creating a new file:

Here we give an example for creating a new file in Python. For creates a file, we have to use the open() method. Here we use try block for erase the errors. The code is given below -

1. `try:`
2. `with open('file1.txt', 'w') as f:`
3. `f.write('Here we create a new file')`
4. `except FileNotFoundError:`
5. `print("The file is does not exist")`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The file is does not exist
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. The tell() methods is return the position of read or write pointer in this file. The syntax of tell() method is given below -

1. fileobject.tell()

Program code1 for File Pointer Position:

Here we give an example for how to find file pointer position in Python. Here we use tell() method and it is return byte number. The code is given below -

1. `# open the file file2.txt in read mode`
2. `fileptr = open("file2.txt","r")`
- 3.
4. `#initially the filepointer is at 0`
5. `print("The filepointer is at byte :",fileptr.tell())`
- 6.
7. `#reading the content of the file`
8. `content = fileptr.read();`
- 9.
10. `#after the read operation file pointer modifies. tell() returns the location of the fileptr.`
- 11.
12. `print("After reading, the filepointer is at:",fileptr.tell())`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Program code2 for File Pointer Position:

Here we give another example for how to find file pointer position in Python. Here we also use tell() method, which is return byte number. The code is given below -

1. `file = open("File2.txt", "r")`

2. `print("The pointer position is: ", file.tell())`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The pointer position is: 0
```

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. That means, using `seek()` method we can easily change the cursor in the file, from where we want to read or write a file.

Syntax:

The syntax for `seek()` method is given below -

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.

Here we give the example of how to modifying the pointer position using `seek()` method in Python. The code is given below -

1. `# open the file file2.txt in read mode`
2. `fileptr = open("file2.txt","r")`
- 3.
4. `#initially the filepointer is at 0`
5. `print("The filepointer is at byte : ",fileptr.tell())`
- 6.
7. `#changing the file pointer location to 10.`
8. `fileptr.seek(10);`
- 9.

10. `#tell() returns the location of the fileptr.`
11. `print("After reading, the filepointer is at:",fileptr.tell())`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
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. It comes from the Python standard utility module. The **os** module provides a portable way to use the operating system-dependent functionality in Python. 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. Using the **rename()** method, we can easily rename the existing File. This method has not any return value. The syntax to use the **rename()** method is given below.

Syntax:

The syntax of **rename** method in Python is given below -

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.

Program code 1 for rename() Method:

Here we give an example of the renaming of the files using **rename()** method in Python. The current file name is `file2.txt`, and the new file name is `file3.txt`. The code is given below -

1. `import os`
- 2.
3. `#rename file2.txt to file3.txt`
4. `os.rename("file2.txt","file3.txt")`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The above code renamed current file2.txt to file3.txt
```

Program code 2 for rename() Method:

Here we give an example of the renaming of the files using rename() method in Python. The current file name is the source, and the new file name is the destination. The code is given below -

```
1. import os
2. def main():
3.     i = 0
4.     path="D:/JavaTpoint/"
5.     for filename in os.listdir(path):
6.         destination = "new" + str(i) + ".png"
7.         source = path + filename
8.         destination = path + destination
9.         os.rename(source, destination)
10.        i += 1
11.
12. if __name__ == '__main__':
13.     main()
```

Removing the file

The os module provides the **remove()** method which is used to remove the specified file.

Syntax:

The syntax of remove method is given below -

```
1. remove(file-name)
```

Program code 1 for remove() method:

```
1. import os;
2. #deleting the file named file3.txt
3. os.remove("file3.txt")
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The file named file3.txt is deleted.
```

Program code 2 for remove() Method:

Here we give an example of removing a file using the remove() method in Python. The file name is file3.txt, which the remove() method deletes. Print the command "This file is not existed" if the File does not exist. The code is given below -

1. import os
2. if os.path.exists("file3.txt "):
3. os.remove("file3.txt ")
4. else:
5. print("This file is not existed")

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
This file is not existed
```

Creating the new directory

The **mkdir()** method is used to create the directories in the current working directory. It creates dictionary in numeric mode. If the file already presents in the system, then it occurs error, which is known as FileExistsError in Python. The mkdir() method does not return any kind of value. The syntax to create the new directory is given below.

Syntax:

The syntax of mkdir() method in Python is given below -

1. os.mkdir (path, mode = 0o777, *, dir_fd = None)

Output:

Parameter:

The syntax of mkdir() method in Python is given below -

path - A path like object represent a path either bytes or the strings object.

mode - Mode is represented by integer value, which means mode is created. If mode is not created then the default value will be 0o777. Its use is optional in mkdir() method.

dir_fd - When the specified path is absolute, in that case dir_fd is ignored. Its use is optional in mkdir() method.

Program code 1 for mkdir() Method:

Here we give the example of mkdir() method by which we can create new dictionary in Python. The code is given below -

```
1. import os  
2.  
3. #creating a new directory with the name new  
4. os.mkdir("new")
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Create a new dictionary which is named new
```

Program code 2 for mkdir() Method:

Here we give the example of mkdir() method by which we can create new dictionary in Python. The code is given below -

```
1. import os  
2. path = '/D:/JavaTpoint'  
3. try:  
4.     os.mkdir(path)  
5. except OSError as error:  
6.     print(error)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
[Error 20] File exists: '/D:/JavaTpoint'
```

The getcwd() method:

This method returns the current working directory which have absolute value. The getcwd() method returns the string value which represents the working dictionary in Python. In getcwd() method, do not require any parameter.

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

Syntax

The syntax of getcwd() method in Python is given below -

```
1. os.getcwd()
```

Program code 1 for getcwd() Method:

Here we give the example of getcwd() method by which we can create new dictionary in Python. The code is given below -

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

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
'C:\\\\Users\\\\DEVANSH SHARMA'
```

Program code 2 for getcwd() Method:

Here we give the example of getcwd() method by which we can create new dictionary in Python. The code is given below -

1. import os
2. c = os.getcwd()
3. print("The working directory is:", c)

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The working directory is: C:\\\\Users\\\\JavaTpoint
```

Changing the current working directory

ADVERTISEMENT

The chdir() method is used to change the current working directory to a specified directory. The chdir() method takes a single argument for the new dictionary path. The chdir() method does not return any kind of value.

Syntax

The syntax of chdir() method is given below -

1. chdir("new-directory")

Program code 1 for chdir() Method:

Here we give the example of chdir() method by which we can change the current working dictionary into new dictionary in Python. The code is given below -

1. **import** os

2. `# Changing current directory with the new directory`
3. `os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")`
4. `#It will display the current working directory`
5. `os.getcwd()`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
'C:\\Users\\DEVANSH SHARMA\\Documents'
```

Program code 2 for chdir() Method:

Here we give another example of chdir() method by which we can change the current working dictionary into new dictionary in Python. The code is given below -

1. `import os`
2. `os.chdir(r"C:\\Users\\JavaTpoint")`
3. `print("Currently working directory is changed")`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Currently working directory is changed
```

Deleting directory:

The rmdir() method is used to delete the specified directory. If the directory is not empty then there occurs OSError. The rmdir() method does not have any kind of return value.

Syntax

1. `os.rmdir(directory name)`

Program code 1 for rmdir() Method:

Here we give the example of rmdir() method by which we can delete a dictionary in Python. The code is given below -

1. `import os`
2. `#removing the new directory`
3. `os.rmdir("directory_name")`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

It will remove the specified directory.

Program code 2 for rmdir() Method:

Here we give another example of rmdir() method by which we can delete a dictionary in Python. The code is given below -

1. import os
2. **directory = "JavaTpoint"**
3. **parent = "/D:/User/Documents"**
4. **path = os.path.join(parent, directory)**
5. **os.rmdir(path)**
6. **print("The directory '%s' is successfully removed", %directory)**

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

The directory 'JavaTpoint' is successfully removed

Output:

Here we give the example of rmdir() method by which we can delete a dictionary in Python. Here we use try block for handle the error. The code is given below -

1. import os
2. **dir = "JavaTpoint"**
3. **parent = "/D:/User/Documents"**
4. **path = os.path.join(parent, dir)**
5. **try:**
6. **os.rmdir(path)**
7. **print("The directory '%s' is successfully removed", %dir)**
8. **except OSError as error:**
9. **print(error)**
10. **print("The directory '%s' cannot be removed successfully", %dir)**

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

[Error 30] Permission denied: '/D:/User/Documents/JavaTpoint'
The directory 'JavaTpoint' cannot be removed successfully

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**.

Program code:

file1.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
2.
3. with open("output.txt", "wb") as f:
4.     subprocess.check_call(["python", "file.py"], stdout=f)
```

File Related Methods:

The file object provides the following methods to manipulate the files on various operating systems. Here we discuss the method and their uses in Python.

| 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.flush() | 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. |

Conclusion:

In this tutorial, we briefly discussed the Python file handling. Users can easily handle the files, like read and write the files in Python. Here we discuss various methods in Python by which we can easily read, write, delete, or rename a file. We also give the program code of these methods for better understanding.

Python Modules

In this tutorial, we 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.

Example:

1. # Here, we are creating a simple Python program to show how to create a module.
2. # defining a function in the module to reuse it

```
3. def square( number ):  
4.     # here, the above function will square the number passed as the input  
5.     result = number ** 2  
6.     return result    # here, we are returning the result of the function
```

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.

Syntax:

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:

Example:

1. # here, we are calling the module square method and passing the value 4
2. result = example_module.square(4)
3. **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. `# Here, we are creating a simple Python program to show how to import a standard module`
2. `# Here, we are import the math module which is a standard module`
3. `import math`
4. `print("The value of euler's number is", math.e)`
5. `# here, we are printing the euler's number from the math module`

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. `# Here, we are creating a simple Python program to show how to import a module and rename it`
2. `# Here, we are import the math module and give a different name to it`
3. `import math as mt # here, we are importing the math module as mt`
4. `print("The value of euler's number is", mt.e)`
5. `# here, we are printing the euler's number from the math module`

Output:

```
The value of euler's number is 2.718281828459045
```

The math module is now named mt in this program. In some situations, 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. # Here, we are creating a simple Python program to show how to import specific
2. # objects from a module
3. # Here, we are import euler's number from the math module using the from keyword
4. **from** math **import** e
5. # here, the e value represents the euler's number
6. **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. # Here, we are creating a simple Python program to show how to import multiple
2. # objects from a module
3. **from** math **import** e, tau
4. **print**("The value of tau constant is: ", tau)
5. **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. # Here, we are importing the complete math module using *
2. **from** math **import** *
3. # Here, we are accessing functions of math module without using the dot operator

4. `print("Calculating square root: ", sqrt(25))`
5. `# here, we are getting the sqrt method and finding the square root of 25`
6. `print("Calculating tangent of an angle: ", tan(pi/6))`
7. `# 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. `# Here, we are importing the sys module`
2. `import sys`
3. `# Here, we are printing the path using sys.path`
4. `print("Path of the sys module in the system is:", sys.path)`

Output:

```
Path of the sys module in the system is:
['/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. `# Here, we are creating a simple 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(): # here, we are defining a function with the name Add Number`
3. `# Here, we are accessing the global namespace`
4. `global Number`
5. `Number = Number + 200`
6. `print("The number is:", Number)`
7. `# here, we are printing the number after performing the addition`
8. `AddNumber() # here, we are calling the function`
9. `print("The number is:", Number)`

Output:

```
The number is: 204  
The number is: 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"`
- 3.
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"
3.
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
2.
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 l
ength 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)**
- 6.
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**

```
2.  
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  
2.  
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 | ArithmetError | 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
4.
5. except:
6.     # This block is optional.
7.     # If the try block encounters an exception, this block will handle it.
8.
9. else:
10.    # If there is no exception, this code block will be executed by the Python interpreter
11.
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 $24 \times 60 \times 60$ 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;
- 2.
3. #returns a time tuple
- 4.
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 week | 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**
- 3.
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
- 2.
3. **#returns the datetime object for the specified time**
- 4.
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:

| 2020 | | | | | | | | | | | |
|----------------|----|----|----|-----------------|----|----|----|------------------|----|----|----|
| January | | | | February | | | | March | | | |
| Mo | Tu | We | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr |
| 1 | 2 | 3 | 4 | 5 | | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 3 | 4 | 5 | 6 | 7 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 10 | 11 | 12 | 13 | 14 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 17 | 18 | 19 | 20 | 21 |
| 27 | 28 | 29 | 30 | 31 | | | 24 | 25 | 26 | 27 | 28 |
| | | | | | | | 29 | 30 | 31 | 29 | 30 |
| April | | | | May | | | | June | | | |
| Mo | Tu | We | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr |
| 1 | 2 | 3 | 4 | 5 | | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 4 | 5 | 6 | 7 | 8 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 18 | 19 | 20 | 21 | 22 |
| 27 | 28 | 29 | 30 | | | | 25 | 26 | 27 | 28 | 29 |
| | | | | | | | 30 | | | 30 | |
| July | | | | August | | | | September | | | |
| Mo | Tu | We | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr |
| 1 | 2 | 3 | 4 | 5 | | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 3 | 4 | 5 | 6 | 7 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 10 | 11 | 12 | 13 | 14 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 17 | 18 | 19 | 20 | 21 |
| 27 | 28 | 29 | 30 | 31 | | | 24 | 25 | 26 | 27 | 28 |
| | | | | | | | 31 | | | 30 | |
| October | | | | November | | | | December | | | |
| Mo | Tu | We | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr |
| 1 | 2 | 3 | 4 | | | | 1 | 2 | 3 | 4 | 5 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 | 2 | 3 | 4 | 5 | 6 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 9 | 10 | 11 | 12 | 13 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 | 16 | 17 | 18 | 19 | 20 |
| 26 | 27 | 28 | 29 | 30 | 31 | | 23 | 24 | 25 | 26 | 27 |
| | | | | | | | 30 | | | 29 | |

Python Regex - RegEx Functions | Metacharacters | Special Sequences

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.

Import the re Module

1. `# Importing re module`

2. `import re`

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 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 called metacharacters.***

The majority of symbols and characters will easily match. (A case-insensitive feature can be enabled, allowing this RE to match Python or PYTHON.) For example, the regular expression 'check' 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 have an effect on other parts of the RE by recurring or modifying their meaning.

Metacharacters or Special Characters

As the name suggests, there are some characters with special meanings:

| Characters | Meaning |
|------------|---|
| . | Dot - It matches any characters except the newline character. |
| ^ | Caret - It is used to match the pattern from the start of the string. (Starts With) |
| \$ | Dollar - It matches the end of the string before the new line character. (Ends with) |
| * | Asterisk - It matches zero or more occurrences of a pattern. |
| + | Plus - It is used when we want a pattern to match at least one. |
| ? | Question mark - It matches zero or one occurrence of a pattern. |
| {} | Curly Braces - It matches the exactly specified number of occurrences of a pattern |
| [] | Bracket - It defines the set of characters |
| | Pipe - It matches any of two defined patterns. |

Special Sequences:

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 once.

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.

Special Sequences consist of '\' followed by a character listed below. Each character has a different meaning.

| Character | Meaning |
|-----------|--|
| \d | It matches any digit and is equivalent to [0-9]. |
| \D | It matches any non-digit character and is equivalent to [^0-9]. |
| \s | It matches any white space character and is equivalent to [\t\n\r\f\v] |
| \S | It matches any character except the white space character and is equivalent to [^\t\n\r\f\v] |
| \w | It matches any alphanumeric character and is equivalent to [a-zA-Z0-9] |
| \W | It matches any characters except the alphanumeric character and is equivalent to [^a-zA-Z0-9] |
| \A | It matches the defined pattern at the start of the string. |
| \b | r"\bxt" - It matches the pattern at the beginning of a word in a string. r"xt\b" - It matches the pattern at the end of a word in a string. |
| \B | This is the opposite of \b. |
| \z | It returns a match object when the pattern is at the end of the string. |

RegEx Functions:

- **compile** - It is used to turn a regular pattern into an object of a regular expression that may be used in a number of ways for matching patterns in a string.
- **search** - It is used to find the first occurrence of a regex pattern in a given string.
- **match** - It starts matching the pattern at the beginning of the string.

- **fullmatch** - It is used to match the whole string with a regex pattern.
- **split** - It is used to split the pattern based on the regex pattern.
- **findall** - It is used to find all non-overlapping patterns in a string. It returns a list of matched patterns.
- **finditer** - It returns an iterator that yields match objects.
- **sub** - It returns a string after substituting the first occurrence of the pattern by the replacement.
- **subn** - It works the same as 'sub'. It returns a tuple (new_string, num_of_substitution).
- **escape** - It is used to escape special characters in a pattern.
- **purge** - It is used to clear the regex expression cache.

1. re.compile(pattern, flags=0)

It is used to create a regular expression object that can be used to match patterns in a string.

Example:

```

1. # Importing re module
2. import re
3.
4. # Defining regEx pattern
5. pattern = "amazing"
6.
7. # Createing a regEx object
8. regex_object = re.compile(pattern)
9.
10. # String
11. text = "This tutorial is amazing!"
12.
13. # Searching for the pattern in the string
14. match_object = regex_object.search(text)
15.
16. # Output
17. print("Match Object:", match_object)

```

Output:

```
Match Object:
```

This is equivalent to:

| | | |
|--------------------------------|---|-------------------------------------|
| re_obj = re.compile(pattern) | = | result = re.search(pattern, string) |
| result = re_obj.search(string) | | |

Note - When it comes to using regular expression objects several times, the `re.compile()` version of the program is much more efficient.

2. `re.match(pattern, string, flags=0)`

- It starts matching the pattern from the beginning of the string.
- Returns a match object if any match is found with information like start, end, span, etc.
- Returns a `NONE` value in the case no match is found.

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.

Example:

1. # Importing re module
2. import re
- 3.
4. # Our pattern
5. pattern = "hello"
- 6.
7. # Returns a match object if found else Null
8. match = re.match(pattern, "hello world")
- 9.
10. print(match) # Printing the match object
11. print("Span:", match.span()) # Return the tuple (start, end)
12. print("Start:", match.start()) # Return the starting index
13. print("End:", match.end()) # Returns the ending index

Output:

```
Span: (0, 5)
Start: 0
End: 5
```

Another example of the implementation of 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.

- o The for loop is used in this Python re.match() illustration to inspect for matches for every element in the list of words.

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)
4.
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!!

3. re.search(pattern, string, flags=0)

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.

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
2.
3. line = "Learn Python through tutorials on javatpoint";
4.

```

```

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

```

4. re.sub(pattern, repl, string, count=0, flags=0)

- It substitutes the matching pattern with the 'repl' in the string
- Pattern - is simply a regex pattern to be matched
- repl - repl stands for "replacement" which replaces the pattern in string.
- Count - This parameter is used to control the number of substitutions

Example 1:

```

1. # Importing re module
2. import re
3.
4. # Defining parameters
5. pattern = "like" # to be replaced
6. repl = "love" # Replacement
7. text = "I like Javatpoint!" # String
8.
9. # Returns a new string with a substituted pattern
10. new_text = re.sub(pattern, repl, text)
11.
12. # Output
13. print("Original text:", text)
14. print("Substituted text: ", new_text)

```

Output:

```

Original text: I like Javatpoint!
Substituted text: I love Javatpoint!

```

In the above example, the sub-function replaces the 'like' with 'love'.

Example 2 - Substituting 3 occurrences of a pattern.

```
1. # Importing re package
2. import re
3.
4. # Defining parameters
5. pattern = "I" # to be replaced
6. repl = "L" # Replacement
7. text = "I like Javatpoint! I also like tutorials!" # String
8.
9. # Returns a new string with the substituted pattern
10. new_text = re.sub(pattern, repl, text, 3)
11.
12. # Output
13. print("Original text:", text)
14. print("Substituted text:", new_text)
```

Output:

```
Original text: I like Javatpoint! I also like tutorials!
Substituted text: I Like Javatpoint! I aLso Like tutorials!
```

Here, first three occurrences of 'I' is substituted with the "L".

5. **re.subn(pattern, repl, string, count=0, flags=0)**

- o Working of subn if same as sub-function
- o It returns a tuple (new_string, num_of_substitutions)

Example:

```
1. # Importing re module
2. import re
3.
4. # Defining parameters
5. pattern = "I" # to be replaced
6. repl = "L" # Replacement
7. text = "I like Javatpoint! I also like tutorials!" # String
8.
9. # Returns a new string with the substituted pattern
10. new_text = re.subn(pattern, repl, text, 3)
11.
12. # Output
13. print("Original text:", text)
14. print("Substituted text:", new_text)
```

Output:

```
Original text: I like Javatpoint! I also like tutorials!
Substituted text: ('I Like Javatpoint! I also Like tutorials!', 3)
```

In the above program, the `subn` function replaces the first three occurrences of 'l' with 'L' in the string.

6. `re.fullmatch(pattern, string, flags=0)`

- It matches the whole string with the pattern.
- Returns a corresponding match object.
- Returns `None` in case no match is found.
- On the other hand, the `search()` function will only search the first occurrence that matches the pattern.

Example:

```
1. # Importing re module
2. import re
3.
4. # Sample string
5. line = "Hello world";
6.
7. # Using re.fullmatch()
8. print(re.fullmatch("Hello", line))
9. print(re.fullmatch("Hello world", line))
```

Output:

```
None
```

In the above program, only the 'Hello world' has completely matched the pattern, not 'Hello'.

Q. When to use `re.findall()`?

Ans. Suppose we have a line of text and 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.

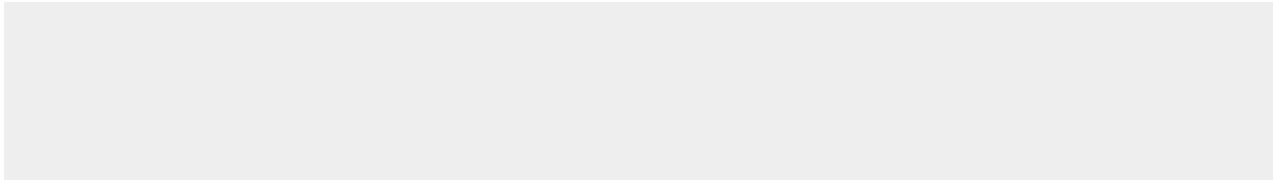
7. `re.finditer(pattern, string, flags=0)`

- Returns an iterator that yields all non-overlapping matches of pattern in a string.
- String is scanned from left to right.
- Returning matches in the order they were discovered

```
1. # Importing re module
```

```
2. import re
3.
4. # Sample string
5. line = "Hello world. I am Here!";
6.
7. # Regex pattern
8. pattern = r'[aeiou]'
9.
10. # Using re.finditer()
11. iter_ = re.finditer(pattern, line)
12.
13. # Iterating the itre_
14. for i in iter_:
15.     print(i)
```

Output:



8. re.split(pattern, string, maxsplit=0, flags=0)

- It splits the pattern by the occurrences of patterns.
- If maxsplit is zero, then the maximum number of splits occurs.
- If maxsplit is one, then it splits the string by the first occurrence of the pattern and returns the remaining string as a final result.

Example:

```
1. # Import re module
2. import re
3.
4. # Pattern
5. pattern = ''
6. # Sample string
7. line = "Learn Python through tutorials on javatpoint"
8.
9. # Using split function to split the string after ''
10. result = re.split( pattern, line)
11.
12. # Printing the result
13. print("When maxsplit = 0, result:", result)
```

```
14.  
15. # When Maxsplit is one  
16. result = re.split(pattern, line, maxsplit=1)  
17. print("When maxsplit = 1, result =", result)
```

Output:

```
When maxsplit = 0, result: ['Learn', 'Python', 'through', 'tutorials', 'on',  
'javatpoint']  
When maxsplit = 1, result = ['Learn', 'Python through tutorials on javatpoint']
```

9. re.escape(pattern)

- It escapes the special character in the pattern.
- The escape function become more important when the string contains regular expression metacharacters in it.

Example:

```
1. # Import re module  
2. import re  
3.  
4. # Pattern  
5. pattern = 'https://www.javatpoint.com/'  
6.  
7. # Using escape function to escape metacharacters  
8. result = re.escape( pattern)  
9.  
10. # Printing the result  
11. print("Result:", result)
```

Output:

```
Result: https://www\.javatpoint\.com/
```

The escape function escapes the metacharacter '.' from the pattern. This is useful when want to treat metacharacters as regular characters to match the actual characters themselves.

10. re.purge()

- The purge function does not take any argument that simply clears the regular expression cache.

Example:

```
1. # Importing re module  
2. import re
```

```
3.  
4. # Define some regular expressions  
5. pattern1 = r'\d+'  
6. pattern2 = r'[a-z]+'  
7.  
8. # Use the regular expressions  
9. print(re.search(pattern1, '123abc'))  
10. print(re.search(pattern2, '123abc'))  
11.  
12. # Clear the regular expression cache  
13. re.purge()  
14.  
15. # Use the regular expressions again  
16. print(re.search(pattern1, '456def'))  
17. print(re.search(pattern2, '456def'))
```

Output:



- After using, pattern1 and pattern2 to search for matches in the string '123abc'.
- We have cleared the cache using re.purge().
- We have again used pattern1 and pattern2 to search for matches in the string '456def'.
- Since the regular expression cache has been cleared. The regular expressions are recompiled, and searching for matches in the '456def' has been performed with the new regular expression object.

Matching Versus Searching - re.match() vs. re.search()

Python has two primary regular expression functions: match and search. The match function looks for a match only where the string starts, whereas the search function looks for a match everywhere in the string.

CODE:

```
1. # Import re module  
2. import re  
3.  
4. # Sample string  
5. line = "Learn Python through tutorials on javatpoint"  
6.  
7. # Using match function to match 'through'
```

```

8. match_object = re.match(r'through', line, re.M|re.I)
9. if match_object:
10.   print("match object group : ", match_object)
11. else:
12.   print("There isn't any match!!")
13.
14. # using search function to search
15. search_object = re.search(r'through', line, re.M|re.I)
16. if search_object:
17.   print("Search object group : ", search_object)
18. else:
19.   print("Nothing found!!")

```

Output:

```

There isn't any match!!
Search object group :

```

The match function checks whether the string is starting with 'through' or not, and the search function checks whether there is 'through' in the string or not.

CONCLUSION

The re-module in Python supports regular expression. Regular expressions are an advanced tool for text processing and pattern matching. We can find patterns in text strings using the re-module and split and replace text depending on patterns, among other things.

Also, using the re-package isn't always a good idea. If we're only searching a fixed string or a specific character class and not leveraging any re-features like the IGNORECASE flag, regular expressions' full capability would not be needed. Strings offer various ways of 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](#), 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)

- 7.
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.

ADVERTISEMENT

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. |

Classes and Objects in Python

Python is an object-oriented programming language that offers classes, which are a potent tool for writing reusable code. To describe objects with shared characteristics and behaviours, classes are utilised. We shall examine Python's ideas of classes and objects in this article.

Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

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.

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.

Example:

Code:

1. `class` Person:
2. `def __init__(self, name, age):`
 `# This is the constructor method that is called when creating a new Person object`
3. `# It takes two parameters, name and age, and initializes them as attributes of the object`
4. `self.name = name`
5. `self.age = age`
6. `def greet(self):`
 `# This is a method of the Person class that prints a greeting message`
7. `print("Hello, my name is " + self.name)`

Name and age are the two properties of the Person class. Additionally, it has a function called `greet` that prints a greeting.

Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

Syntax:

1. `# Declare an object of a class`

- object_name = Class_Name(arguments)

Example:

Code:

```
1. class Person:  
2.     def __init__(self, name, age):  
3.         self.name = name  
4.         self.age = age  
5.     def greet(self):  
6.         print("Hello, my name is " + self.name)  
7.  
8. # Create a new instance of the Person class and assign it to the variable person1  
9. person1 = Person("Ayan", 25)  
10. person1.greet()
```

Output:

```
"Hello, my name is Ayan"
```

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.

__init__ method

In order to make an instance of a class in Python, a specific function called `__init__` is called. Although it is used to set the object's attributes, it is often referred to as a constructor.

The self-argument is the only one required by the `__init__` method. This argument refers to the newly generated instance of the class. To initialise the values of each attribute associated with the objects, you can declare extra arguments in the `__init__` method.

Class and Instance Variables

All instances of a class exchange class variables. They function independently of any class methods and may be accessed through the use of the class name. Here's an illustration:

Code:

```
1. class Person:  
2.     count = 0 # This is a class variable  
3.
```

```
4. def __init__(self, name, age):
5.     self.name = name # This is an instance variable
6.     self.age = age
7.     Person.count += 1 # Accessing the class variable using the name of the class
8. person1 = Person("Ayan", 25)
9. person2 = Person("Bobby", 30)
10. print(Person.count)
```

Output:

```
2
```

Whereas, instance variables are specific to each instance of a class. They are specified using the self-argument in the `__init__` method. Here's an illustration:

Code:

```
1. class Person:
2.     def __init__(self, name, age):
3.         self.name = name # This is an instance variable
4.         self.age = age
5.     person1 = Person("Ayan", 25)
6.     person2 = Person("Bobby", 30)
7.     print(person1.name)
8.     print(person2.age)
```

Output:

```
Ayan
30
```

Class variables are created separately from any class methods and are shared by all class copies. Every instance of a class has its own instance variables, which are specified in the `__init__` method utilising the self-argument.

Conclusion:

In conclusion, Python's classes and objects notions are strong ideas that let you write reusable programmes. You may combine information and capabilities into a single entity that is able to be used to build many objects by establishing a class. Using the dot notation, you may access an object's methods and properties after it has been created. You can develop more logical, effective, and manageable code by comprehending Python's classes and objects.

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  
5.  
6.     def display(self):  
7.         print("ID: %d \nName: %s" % (self.id, self.name))  
8.  
9.  
10. emp1 = Employee("John", 101)  
11. emp2 = Employee("David", 102)  
12.  
13. # accessing display() method to print employee 1 information  
14.  
15. emp1.display()  
16.
```

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 parameterized constructor")  
5.         self.name = name  
6.     def show(self):  
7.         print("Hello",self.name)  
8. student = Student("John")  
9. student.show()
```

Output:

```
This is parameterized 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"  
4.  
5.     def display(self):  
6.         print(self.roll_num,self.name)  
7.  
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")  
6.  
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  
6.  
7.     # creates the object of the class Student  
8. s = Student("John", 101, 22)  
9.
```

```

10. # prints the attribute name of the object s
11. print(getattr(s, 'name'))
12.
13. # reset the value of attribute age to 23
14. setattr(s, "age", 23)
15.
16. # prints the modified value of age
17. print(getattr(s, 'age'))
18.
19. # prints true if the student contains the attribute with name id
20.
21. print(hasattr(s, 'id'))
22. # deletes the attribute age
23. delattr(s, 'age')
24.
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:

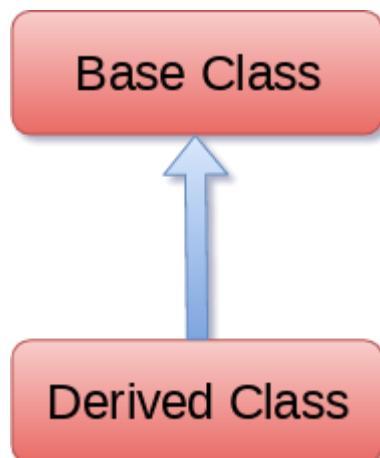
```
None  
{'name': 'John', 'id': 101, 'age': 22}  
__main__
```

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.



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

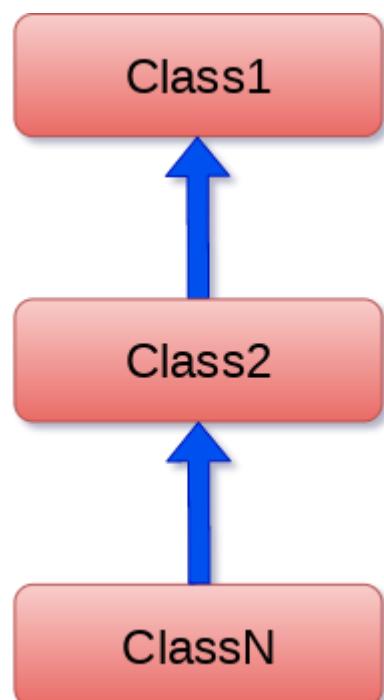
1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. d = Dog()
9. d.bark()
10. d.speak()

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.



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

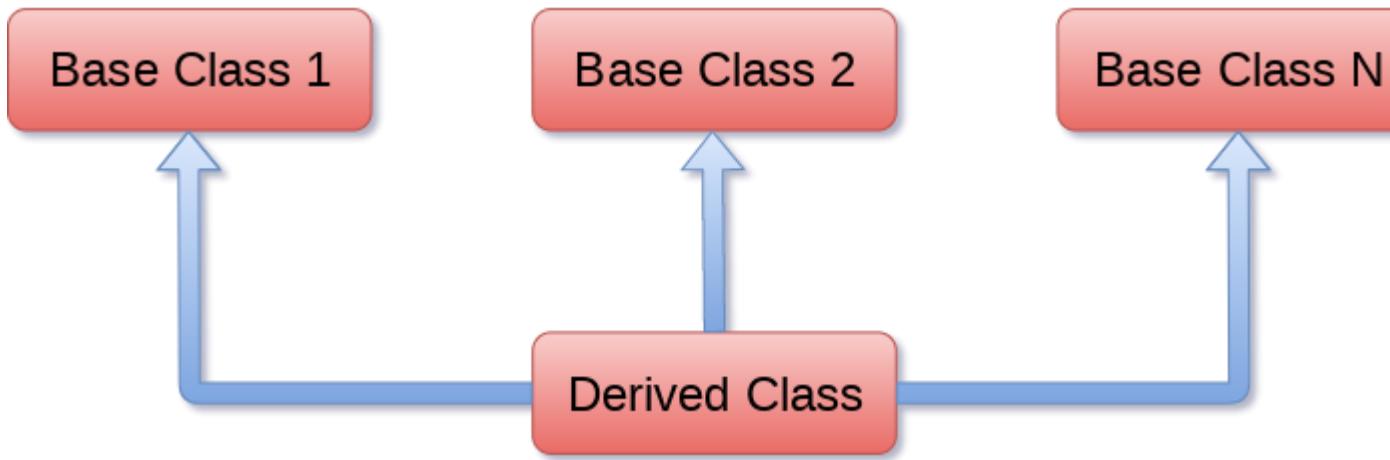
1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. **class** DogChild(Dog):
10.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



The syntax to perform multiple inheritance is given below.

Syntax

```

1. class Base1:
2.     <class-suite>
3.
4. class Base2:
5.     <class-suite>
6. .
7. .
8. .
9. class BaseN:
10.    <class-suite>
11.
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;
```

```
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](#).

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](#).

Abstraction classes in Python

In [Python](#), 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
7.
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 ")
17.
18. class Renault(Car):
19.     def mileage(self):
20.        print("The mileage is 27kmph ")
21.
22. # Driver code
23. t= Tesla ()
24. t.mileage()
25.
26. r = Renault()
27. r.mileage()
28.
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  
3.  
4. from abc import ABC  
5.  
6. class Polygon(ABC):  
7.  
8.     # abstract method  
9.     def sides(self):  
10.        pass  
11.  
12. class Triangle(Polygon):  
13.  
14.  
15.     def sides(self):  
16.         print("Triangle has 3 sides")  
17.  
18. class Pentagon(Polygon):  
19.  
20.  
21.     def sides(self):  
22.         print("Pentagon has 5 sides")
```

```
23.  
24. class Hexagon(Polygon):  
25.  
26.     def sides(self):  
27.         print("Hexagon has 6 sides")  
28.  
29. class square(Polygon):  
30.  
31.     def sides(self):  
32.         print("I have 4 sides")  
33.  
34. # Driver code  
35. t = Triangle()  
36. t.sides()  
37.  
38. s = square()  
39. s.sides()  
40.  
41. p = Pentagon()  
42. p.sides()  
43.  
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.

- o 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.

\$ 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.

```
javatpoint@localhost:~$ python3
Python 3.4.9 (default, Aug 14 2018, 21:28:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import mysql.connector
>>>
```

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  
2.  
3. #Create the connection object  
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")  
5.  
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  
2.  
3. #Create the connection object  
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", database = "mydb")  
5.  
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")
4.
5. #printing the connection object
6. print(myconn)
7.
8. #creating the cursor object
9. cur = myconn.cursor()
10.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #creating a new database
11.     cur.execute("create database PythonDB2")
12.
13.     #getting the list of all the databases which will now include the new database PythonD
14.     B
15.     dbs = cur.execute("show databases")
16. except:
17.     myconn.rollback()
18.
19. for x in cur:
20.     print(x)
21.
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
- 2.
3. **#Create the connection object**
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
- 5.
6. **#creating the cursor object**
7. cur = myconn.cursor()
- 8.
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()
- 14.
15. myconn.close()

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
Database changed  
MariaDB [PythonDB]> show tables;  
+-----+  
| Tables_in_PythonDB |  
+-----+  
| Employee |  
+-----+  
1 row in set (0.00 sec)  
  
MariaDB [PythonDB]> desc Employee;  
+-----+-----+-----+-----+-----+  
| Field | Type   | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| name  | varchar(20) | NO   |     | NULL    |       |  
| id    | int(20)      | NO   | PRI  | NULL    |       |  
| salary | float        | NO   |     | NULL    |       |  
| Dept_id | int(11)      | NO   |     | NULL    |       |  
+-----+-----+-----+-----+-----+  
4 rows in set (0.01 sec)  
  
MariaDB [PythonDB]>
```

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
- 2.
3. **#Create the connection object**
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
- 5.
6. **#creating the cursor object**
7. cur = myconn.cursor()
- 8.
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()
14.
15. myconn.close()

```

```

javatpoint@localhost:~ - □ ×
File Edit View Search Terminal Help
Server version: 10.1.30-MariaDB MariaDB Server
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use PythonDB
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [PythonDB]> desc Employee;
+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| name       | varchar(20) | NO   |     | NULL    |          |
| id         | int(20)    | NO   | PRI | NULL    |          |
| salary     | float      | NO   |     | NULL    |          |
| Dept_id    | int(11)    | NO   |     | NULL    |          |
| branch_name | varchar(20) | NO   |     | NULL    |          |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

MariaDB [PythonDB]>

```

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)"
7.
8. #The row values are provided in the form of tuple
9. val = ("John", 110, 25000.00, 201, "Newyork")
10.
11. try:
12.     #inserting the values into the table
13.     cur.execute(sql,val)
14.
15.     #commit the transaction
16.     myconn.commit()
17.
18. except:
19.     myconn.rollback()
20.
21. print(cur.rowcount,"record inserted!")
22. myconn.close()

```

Output:

```

1 record inserted!
[javatpoint@localhost ~]$ mysql -u root -p
Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 56
Server version: 10.1.30-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use PythonDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [PythonDB]> select * from Employee;
+----+----+----+----+
| name | id  | salary | Dept_id | branch_name |
+----+----+----+----+
| John | 101 | 25000 |    201 | Newyork      |
+----+----+----+----+
1 row in set (0.00 sec)

MariaDB [PythonDB]>

```

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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.
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")]
10.
11. try:
12.     #inserting the values into the table
13.     cur.executemany(sql,val)
14.
15.     #commit the transaction
16.     myconn.commit()
17.     print(cur.rowcount,"records inserted!")
18.
19. except:
20.     myconn.rollback()
21.
22. myconn.close()
```

Output:

```
3 records inserted!
```

```
javatpoint@localhost:~ - □ ×
File Edit View Search Terminal Help
Your MariaDB connection id is 61
Server version: 10.1.30-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use PythonDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [PythonDB]> select * from Employee;
+-----+-----+-----+-----+
| name | id | salary | Dept_id | branch_name |
+-----+-----+-----+-----+
| John | 101 | 25000 | 201 | Newyork |
| John | 102 | 25000 | 201 | Newyork |
| David | 103 | 25000 | 202 | Port of spain |
| Nick | 104 | 90000 | 201 | Newyork |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

MariaDB [PythonDB]> █
```

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()`
- 6.
7. `sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s, %s)"`
- 8.
9. `val = ("Mike",105,28000,202,"Guyana")`
- 10.
11. `try:`
12. `#inserting the values into the table`

```
13. cur.execute(sql,val)
14.
15. #commit the transaction
16. myconn.commit()
17.
18. #getting rowid
19. print(cur.rowcount,"record inserted! id:",cur.lastrowid)
20.
21. except:
22.     myconn.rollback()
23.
24. myconn.close()
```

Output:

```
1 record inserted! Id: 0
```

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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #Reading the Employee data
11.     cur.execute("select * from Employee")
12.
13.     #fetching the rows from the cursor object
14.     result = cur.fetchall()
```

```
15. #printing the result
16.
17. for x in result:
18.     print(x);
19. except:
20.     myconn.rollback()
21.
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")
9.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
   = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee")
12.
13.    #fetching the first row from the cursor object
14.    result = cur.fetchone()
15.
16.    #printing the result
17.    print(result)
18.
19. except:
20.    myconn.rollback()
21.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
   = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.
11. #Reading the Employee data
12. cur.execute("select name, id, salary from Employee")
13.
14. #fetching the rows from the cursor object
15. result = cur.fetchall()
16.
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()
22.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #Reading the Employee data
11.     cur.execute("select name, id, salary from Employee where name like 'J%'")
12.
13.     #fetching the rows from the cursor object
14.     result = cur.fetchall()
15.
16.     print("Name    id    Salary");
17.
18.     for row in result:
19.         print("%s    %d    %d"%(row[0],row[1],row[2]))
20. except:
21.     myconn.rollback()
22.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.
```

```

6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee where id in (101,102,103)")
12.
13.    #fetching the rows from the cursor object
14.    result = cur.fetchall()
15.
16.    print("Name  id  Salary");
17.
18.    for row in result:
19.        print("%s  %d  %d"%(row[0],row[1],row[2]))
20. except:
21.    myconn.rollback()
22.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
   = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee order by name")

```

```

12.
13. #fetching the rows from the cursor object
14. result = cur.fetchall()
15.
16. print("Name  id  Salary");
17.
18. for row in result:
19.     print("%s  %d  %d"%(row[0],row[1],row[2]))
20. except:
21.     myconn.rollback()
22.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee order by name desc")
12.
13.    #fetching the rows from the cursor object
14.    result = cur.fetchall()
15.
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]))
20.
21. except:
22.     myconn.rollback()
23.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
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:
14.
15.    myconn.rollback()
16.
17. myconn.close()

```

File Edit View Search Terminal Help

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement

MariaDB [(none)]> use PythonDB;

Reading table information for completion of table and column names

You can turn off this feature to get a quicker startup with -A

Database changed

MariaDB [PythonDB]> select * from Employee;

| name | id | salary | Dept_id | branch_name |
|-------|-----|--------|---------|---------------|
| John | 101 | 25000 | 201 | Newyork |
| John | 102 | 25000 | 201 | Newyork |
| David | 103 | 25000 | 202 | Port of spain |
| Nick | 104 | 90000 | 201 | Newyork |
| Mike | 105 | 28000 | 202 | Guyana |
| alex | 110 | 25000 | 201 | Newyork |

6 rows in set (0.00 sec)

MariaDB [PythonDB]> █

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
- 2.
3. **#Create the connection object**
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
- 5.

```

6. #creating the cursor object
7. cur = myconn.cursor()
8.
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:
14.
15.     myconn.rollback()
16.
17. myconn.close()

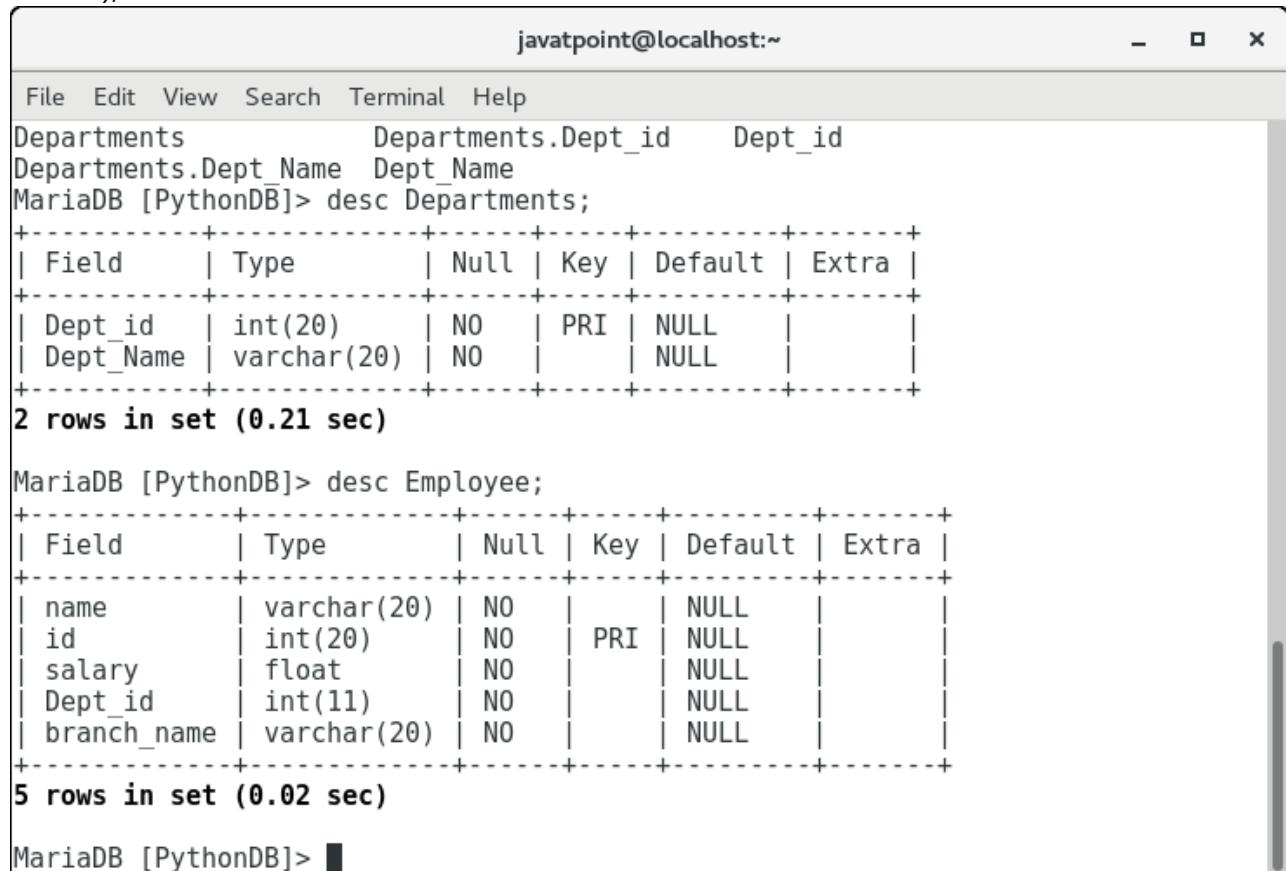
```

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**);



```

javatpoint@localhost:~ - □ ×
File Edit View Search Terminal Help
Departments          Departments.Dept_id      Dept_id
Departments.Dept_Name Dept_Name
MariaDB [PythonDB]> desc Departments;
+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| Dept_id    | int(20)   | NO   | PRI | NULL    |       |
| Dept_Name  | varchar(20)| NO   |     | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.21 sec)

MariaDB [PythonDB]> desc Employee;
+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| name       | varchar(20) | NO   |     | NULL    |       |
| id         | int(20)    | NO   | PRI | NULL    |       |
| salary     | float      | NO   |     | NULL    |       |
| Dept_id    | int(11)    | NO   |     | NULL    |       |
| branch_name| varchar(20)| NO   |     | NULL    |       |
+-----+-----+-----+-----+-----+
5 rows in set (0.02 sec)

MariaDB [PythonDB]> █

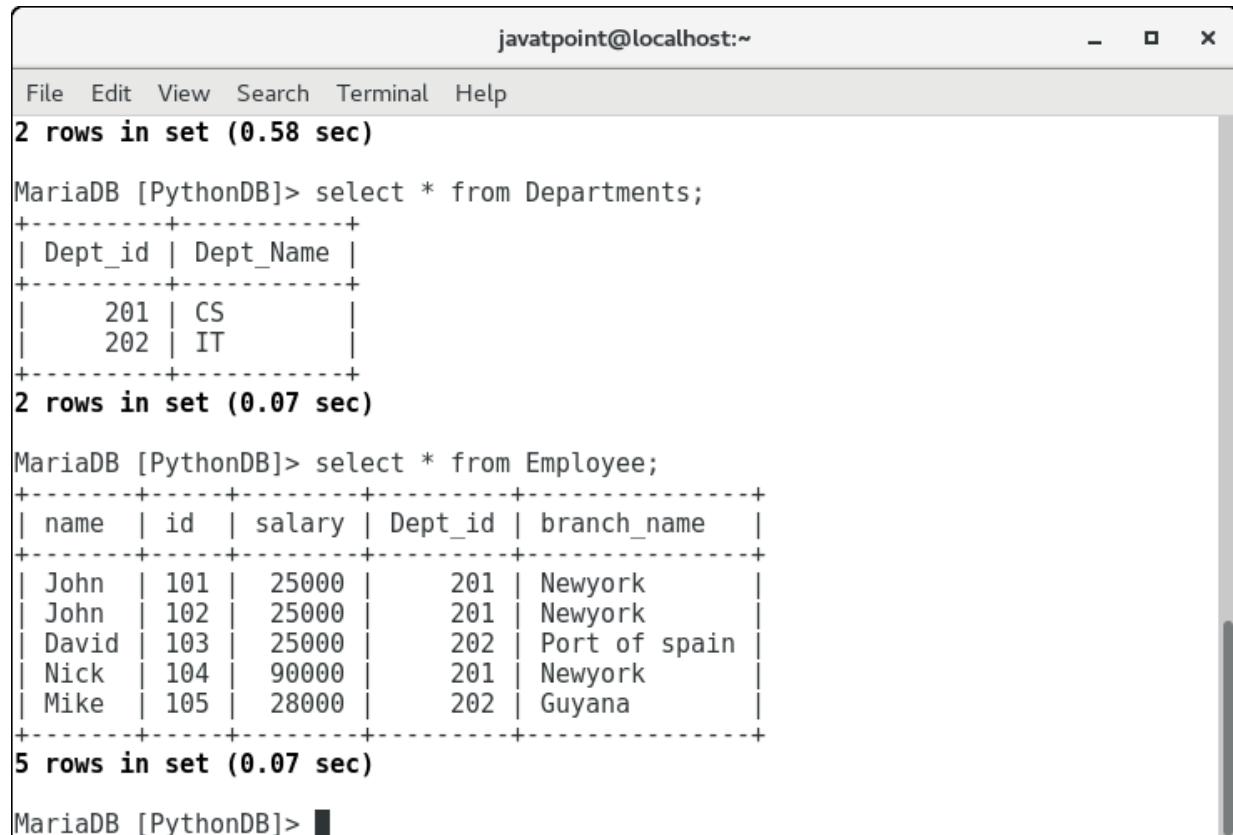
```

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.



The screenshot shows a terminal window titled 'javatpoint@localhost:~'. The window contains the following MySQL session:

```
File Edit View Search Terminal Help
2 rows in set (0.58 sec)

MariaDB [PythonDB]> select * from Departments;
+-----+-----+
| Dept_id | Dept_Name |
+-----+-----+
| 201    | CS      |
| 202    | IT      |
+-----+
2 rows in set (0.07 sec)

MariaDB [PythonDB]> select * from Employee;
+-----+-----+-----+-----+-----+
| name  | id   | salary | Dept_id | branch_name |
+-----+-----+-----+-----+-----+
| John  | 101  | 25000 | 201    | Newyork     |
| John  | 102  | 25000 | 201    | Newyork     |
| David | 103  | 25000 | 202    | Port of spain |
| Nick  | 104  | 90000 | 201    | Newyork     |
| Mike  | 105  | 28000 | 202    | Guyana      |
+-----+-----+-----+-----+
5 rows in set (0.07 sec)

MariaDB [PythonDB]> █
```

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`
- 2.
3. `#Create the connection object`
4. `myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")`
- 5.
6. `#creating the cursor object`
7. `cur = myconn.cursor()`
- 8.
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]))
15.
16. except:
17.     myconn.rollback()
18.
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
- 2.
3. **#Create the connection object**
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

```

5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #joining the two tables on departments_id
11.     result = cur.execute("select Employee.id, Employee.name, Employee.salary, Department
12.         s.Dept_id, Departments.Dept_Name from Departments right join Employee on Departme
13.         nts.Dept_id = Employee.Dept_id")
14.
15.     for row in cur:
16.         print(row[0], row[1], row[2], row[3], row[4])
17.
18.
19.
20. except:
21.     myconn.rollback()
22.
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
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database
= "PythonDB")
5.

```

```

6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #joining the two tables on departments_id
11.     result = cur.execute("select Employee.id, Employee.name, Employee.salary, Department
12.         s.Dept_id, Departments.Dept_Name from Departments left join Employee on Department
13.         s.Dept_id = Employee.Dept_id")
14.     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])
17.
18. except:
19.     myconn.rollback()
20.
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`
- 2.
3. `#Create the connection object`
4. `myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")`
- 5.
6. `#creating the cursor object`

```
7. cur = myconn.cursor()
8.
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()
16.
17. myconn.close()
```

Output:

Deleted !

Python Tkinter Tutorial



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

Python provides the standard library Tkinter for creating the graphical user interface for desktop based applications.

Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

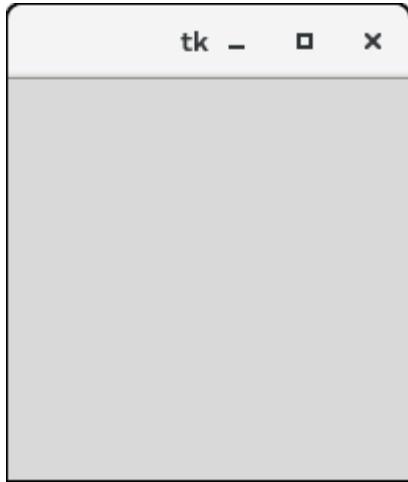
1. import the Tkinter module.
2. Create the main application window.
3. Add the widgets like labels, buttons, frames, etc. to the window.
4. Call the main event loop so that the actions can take place on the user's computer screen.

Example

```
1. #!/usr/bin/python3
```

```
2. from tkinter import *
3. #creating the application main window.
4. top = Tk()
5. #Entering the event main loop
6. top.mainloop()
```

Output:



Tkinter widgets

There are various widgets like button, canvas, checkbutton, entry, etc. that are used to build the python GUI applications.

| SN | Widget | Description |
|----|-----------------------------|--|
| 1 | Button | The Button is used to add various kinds of buttons to the python application. |
| 2 | Canvas | The canvas widget is used to draw the canvas on the window. |
| 3 | Checkbutton | The Checkbutton is used to display the CheckButton on the window. |
| 4 | Entry | The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values. |
| 5 | Frame | It can be defined as a container to which, another widget can be added and organized. |
| 6 | Label | A label is a text used to display some message or information about the other widgets. |
| 7 | ListBox | The ListBox widget is used to display a list of options to the user. |

| | | |
|----|------------------------------------|--|
| 8 | <u>Menubutton</u> | The Menubutton is used to display the menu items to the user. |
| 9 | <u>Menu</u> | It is used to add menu items to the user. |
| 10 | <u>Message</u> | The Message widget is used to display the message-box to the user. |
| 11 | <u>Radiobutton</u> | The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them. |
| 12 | <u>Scale</u> | It is used to provide the slider to the user. |
| 13 | <u>Scrollbar</u> | It provides the scrollbar to the user so that the user can scroll the window up and down. |
| 14 | <u>Text</u> | It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it. |
| 14 | <u>Toplevel</u> | It is used to create a separate window container. |
| 15 | <u>Spinbox</u> | It is an entry widget used to select from options of values. |
| 16 | <u>PanedWindow</u> | It is like a container widget that contains horizontal or vertical panes. |
| 17 | <u>LabelFrame</u> | A LabelFrame is a container widget that acts as the container |
| 18 | <u>MessageBox</u> | This module is used to display the message-box in the desktop based applications. |

Python Tkinter Geometry

The Tkinter geometry specifies the method by using which, the widgets are represented on display. The python Tkinter provides the following geometry methods.

1. The pack() method
2. The grid() method
3. The place() method

Let's discuss each one of them in detail.

Python Tkinter pack() method

The pack() widget is used to organize widget in the block. The positions widgets added to the python application using the pack() method can be controlled by using the various options specified in the method call.

However, the controls are less and widgets are generally added in the less organized manner.

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

syntax

1. `widget.pack(options)`

A list of possible options that can be passed in pack() is given below.

- o **expand:** If the expand is set to true, the widget expands to fill any space.
- o **Fill:** By default, the fill is set to NONE. However, we can set it to X or Y to determine whether the widget contains any extra space.
- o **size:** it represents the side of the parent to which the widget is to be placed on the window.

Example

1. `#!/usr/bin/python3`
2. `from tkinter import *`
3. `parent = Tk()`
4. `redbutton = Button(parent, text = "Red", fg = "red")`
5. `redbutton.pack(side = LEFT)`
6. `greenbutton = Button(parent, text = "Black", fg = "black")`
7. `greenbutton.pack(side = RIGHT)`
8. `bluebutton = Button(parent, text = "Blue", fg = "blue")`
9. `bluebutton.pack(side = TOP)`
10. `blackbutton = Button(parent, text = "Green", fg = "red")`
11. `blackbutton.pack(side = BOTTOM)`
12. `parent.mainloop()`

Output:



Python Tkinter grid() method

The grid() geometry manager organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call. We can also specify the column span (width) or rowspan(height) of a widget.

This is a more organized way to place the widgets to the python application. The syntax to use the grid() is given below.

Syntax

1. `widget.grid(options)`

A list of possible options that can be passed inside the grid() method is given below.

- **Column**
The column number in which the widget is to be placed. The leftmost column is represented by 0.
- **Columnspan**
The width of the widget. It represents the number of columns up to which, the column is expanded.
- **ipadx,** **ipady**
It represents the number of pixels to pad the widget inside the widget's border.
- **padx,** **pady**
It represents the number of pixels to pad the widget outside the widget's border.
- **row**
The row number in which the widget is to be placed. The topmost row is represented by 0.
- **rowspan**
The height of the widget, i.e. the number of the row up to which the widget is expanded.
- **Sticky**
If the cell is larger than a widget, then sticky is used to specify the position of the widget inside the cell. It may be the concatenation of the sticky letters representing the position of the widget. It may be N, E, W, S, NE, NW, NS, EW, ES.

Example

1. `#!/usr/bin/python3`
2. `from tkinter import *`
3. `parent = Tk()`
4. `name = Label(parent,text = "Name").grid(row = 0, column = 0)`
5. `e1 = Entry(parent).grid(row = 0, column = 1)`
6. `password = Label(parent,text = "Password").grid(row = 1, column = 0)`
7. `e2 = Entry(parent).grid(row = 1, column = 1)`
8. `submit = Button(parent, text = "Submit").grid(row = 4, column = 0)`

9. parent.mainloop()

Output:



Python Tkinter place() method

The place() geometry manager organizes the widgets to the specific x and y coordinates.

Syntax

1. widget.place(options)

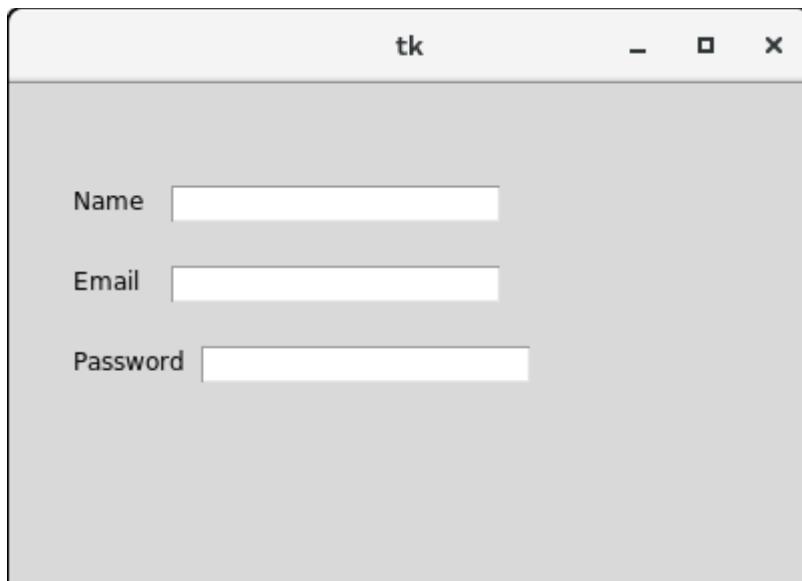
A list of possible options is given below.

- **Anchor:** It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)
- **bordermode:** The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
- **height, width:** It refers to the height and width in pixels.
- **relheight, relwidth:** It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
- **relx, rely:** It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
- **x, y:** It refers to the horizontal and vertical offset in the pixels.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3. top = Tk()
4. top.geometry("400x250")
5. name = Label(top, text = "Name").place(x = 30,y = 50)
6. email = Label(top, text = "Email").place(x = 30, y = 90)
7. password = Label(top, text = "Password").place(x = 30, y = 130)
8. e1 = Entry(top).place(x = 80, y = 50)
9. e2 = Entry(top).place(x = 80, y = 90)
10. e3 = Entry(top).place(x = 95, y = 130)
11. top.mainloop()
```

Output:



Prerequisite

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

Audience

Our Python Tkinter tutorial is designed to help beginners and professionals.

Problem

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

Python Tkinter Button

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

The syntax to use the button widget is given below.

Syntax

1. `W = Button(parent, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|------------------|---|
| 1 | activebackground | Background color when the button is active. |

| | | |
|----|------------------|--|
| 1 | activebackground | It represents the background of the button when the mouse hover the button. |
| 2 | activeforeground | It represents the font color of the button when the mouse hover the button. |
| 3 | Bd | It represents the border width in pixels. |
| 4 | Bg | It represents the background color of the button. |
| 5 | Command | It is set to the function call which is scheduled when the function is called. |
| 6 | Fg | Foreground color of the button. |
| 7 | Font | The font of the button text. |
| 8 | Height | The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images. |
| 10 | Highlightcolor | The color of the highlight when the button has the focus. |
| 11 | Image | It is set to the image displayed on the button. |
| 12 | justify | It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center. |
| 13 | Padx | Additional padding to the button in the horizontal direction. |
| 14 | pady | Additional padding to the button in the vertical direction. |
| 15 | Relief | It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE. |
| 17 | State | This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button. |
| 18 | Underline | Set this option to make the button text underlined. |
| 19 | Width | The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons. |

| | | |
|----|------------|---|
| 20 | Wraplength | If the value is set to a positive number, the text lines will be wrapped to fit within this length. |
|----|------------|---|

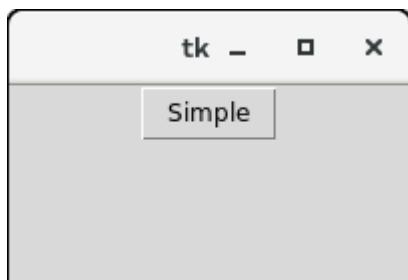
Example

```

1. #python application to create a simple button
2.
3. from tkinter import *
4.
5.
6. top = Tk()
7.
8. top.geometry("200x100")
9.
10. b = Button(top,text = "Simple")
11.
12. b.pack()
13.
14. top.mainloop()

```

Output:



Example

```

1. from tkinter import *
2.
3. top = Tk()
4.
5. top.geometry("200x100")
6.
7. def fun():
8.     messagebox.showinfo("Hello", "Red Button clicked")
9.
10.
11. b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackground
12.             = "pink",pady=10)

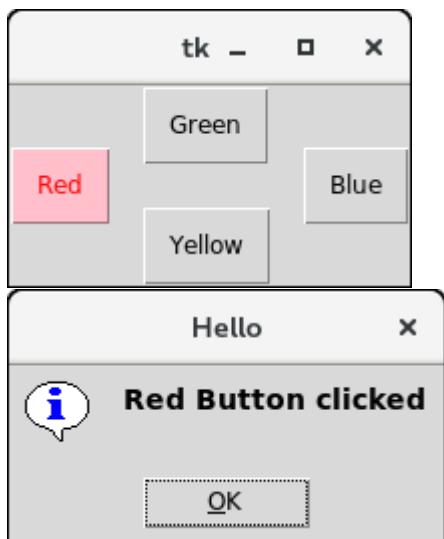
```

```

13. b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink",pady = 10)
14.
15. b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pink",pady = 10)
16.
17. b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "pink",pady = 10)
18.
19. b1.pack(side = LEFT)
20.
21. b2.pack(side = RIGHT)
22.
23. b3.pack(side = TOP)
24.
25. b4.pack(side = BOTTOM)
26.
27. top.mainloop()

```

Output:



Python Tkinter Canvas

The canvas widget is used to add the structured graphics to the python application. It is used to draw the graph and plots to the python application. The syntax to use the canvas is given below.

Syntax

- w = canvas(parent, options)

A list of possible options is given below.

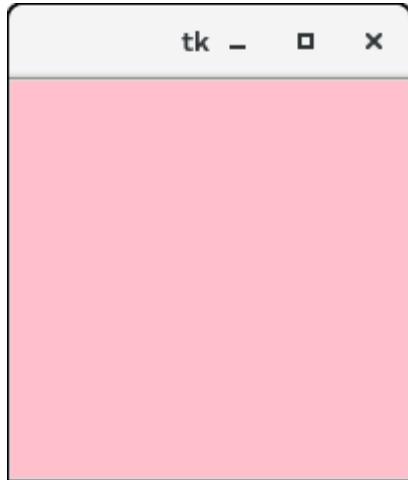
| SN | Option | Description |
|----|------------------|--|
| 1 | bd | The represents the border width. The default width is 2. |
| 2 | bg | It represents the background color of the canvas. |
| 3 | confine | It is set to make the canvas unscrollable outside the scroll region. |
| 4 | cursor | The cursor is used as the arrow, circle, dot, etc. on the canvas. |
| 5 | height | It represents the size of the canvas in the vertical direction. |
| 6 | highlightcolor | It represents the highlight color when the widget is focused. |
| 7 | relief | It represents the type of the border. The possible values are SUNKEN, RAISED, GROOVE, and RIDGE. |
| 8 | scrollregion | It represents the coordinates specified as the tuple containing the area of the canvas. |
| 9 | width | It represents the width of the canvas. |
| 10 | xscrollincrement | If it is set to a positive value. The canvas is placed only to the multiple of this value. |
| 11 | xscrollcommand | If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar. |
| 12 | yscrollincrement | Works like xscrollincrement, but governs vertical movement. |
| 13 | yscrollcommand | If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar. |

Example

1. `from tkinter import *`
- 2.
3. `top = Tk()`
- 4.
5. `top.geometry("200x200")`
- 6.
7. `#creating a simple canvas`
8. `c = Canvas(top,bg = "pink",height = "200")`

```
9.  
10.  
11. c.pack()  
12.  
13. top.mainloop()
```

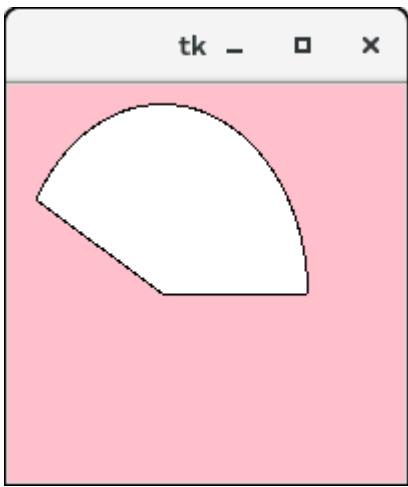
Output:



Example: Creating an arc

```
1. from tkinter import *  
2.  
3. top = Tk()  
4.  
5. top.geometry("200x200")  
6.  
7. #creating a simple canvas  
8. c = Canvas(top,bg = "pink",height = "200",width = 200)  
9.  
10. arc = c.create_arc((5,10,150,200),start = 0,extent = 150, fill= "white")  
11.  
12. c.pack()  
13.  
14. top.mainloop()
```

Output:



Python Tkinter Checkbutton

The Checkbutton is used to track the user's choices provided to the application. In other words, we can say that Checkbutton is used to implement the on/off selections.

The Checkbutton can contain the text or images. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

The syntax to use the checkbutton is given below.

Syntax

1. `w = checkbutton(master, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|------------------|---|
| 1 | activebackground | It represents the background color when the checkbutton is under the cursor. |
| 2 | activeforeground | It represents the foreground color of the checkbutton when the checkbutton is under the cursor. |
| 3 | bg | The background color of the button. |
| 4 | bitmap | It displays an image (monochrome) on the button. |
| 5 | bd | The size of the border around the corner. |
| 6 | command | It is associated with a function to be called when the state of the checkbutton is changed. |

| | | |
|----|--------------------|--|
| 7 | cursor | The mouse pointer will be changed to the cursor name when it is over the checkbutton. |
| 8 | disabledforeground | It is the color which is used to represent the text of a disabled checkbutton. |
| 9 | font | It represents the font of the checkbutton. |
| 10 | fg | The foreground color (text color) of the checkbutton. |
| 11 | height | It represents the height of the checkbutton (number of lines). The default height is 1. |
| 12 | highlightcolor | The color of the focus highlight when the checkbutton is under focus. |
| 13 | image | The image used to represent the checkbutton. |
| 14 | justify | This specifies the justification of the text if the text contains multiple lines. |
| 15 | offvalue | The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable to some other one. |
| 16 | onvalue | The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable to some other one. |
| 17 | padx | The horizontal padding of the checkbutton |
| 18 | pady | The vertical padding of the checkbutton. |
| 19 | relief | The type of the border of the checkbutton. By default, it is set to FLAT. |
| 20 | selectcolor | The color of the checkbutton when it is set. By default, it is red. |
| 21 | selectimage | The image is shown on the checkbutton when it is set. |
| 22 | state | It represents the state of the checkbutton. By default, it is set to normal. We can change it to DISABLED to make the checkbutton unresponsive. The state of the checkbutton is ACTIVE when it is under focus. |

| | | |
|----|------------|---|
| 24 | underline | It represents the index of the character in the text which is to be underlined. The indexing starts with zero in the text. |
| 25 | variable | It represents the associated variable that tracks the state of the checkbox. |
| 26 | width | It represents the width of the checkbox. It is represented in the number of characters that are represented in the form of texts. |
| 27 | wraplength | If this option is set to an integer number, the text will be broken into the number of pieces. |

Methods

The methods that can be called with the Checkbuttons are described in the following table.

| SN | Method | Description |
|----|------------|---|
| 1 | deselect() | It is called to turn off the checkbox. |
| 2 | flash() | The checkbox is flashed between the active and normal colors. |
| 3 | invoke() | This will invoke the method associated with the checkbox. |
| 4 | select() | It is called to turn on the checkbox. |
| 5 | toggle() | It is used to toggle between the different Checkbuttons. |

Example

1. `from tkinter import *`
- 2.
3. `top = Tk()`
- 4.
5. `top.geometry("200x200")`
- 6.
7. `checkvar1 = IntVar()`
- 8.
9. `checkvar2 = IntVar()`
- 10.
11. `checkvar3 = IntVar()`
- 12.

```
13. chkbtn1 = Checkbutton(top, text = "C", variable = checkvar1, onvalue = 1, offvalue = 0, height = 2, width = 10)
14.
15. chkbtn2 = Checkbutton(top, text = "C++", variable = checkvar2, onvalue = 1, offvalue = 0, height = 2, width = 10)
16.
17. chkbtn3 = Checkbutton(top, text = "Java", variable = checkvar3, onvalue = 1, offvalue = 0, height = 2, width = 10)
18.
19. chkbtn1.pack()
20.
21. chkbtn2.pack()
22.
23. chkbtn3.pack()
24.
25. top.mainloop()
```

Output:



Python Tkinter Entry

The Entry widget is used to provide the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user. It can only be used for one line of text from the user. For multiple lines of text, we must use the text widget.

The syntax to use the Entry widget is given below.

Syntax

1. w = Entry (parent, options)

A list of possible options is given below.

| SN | Option | Description |
|-----------|---------------------|--|
| 1 | bg | The background color of the widget. |
| 2 | bd | The border width of the widget in pixels. |
| 3 | cursor | The mouse pointer will be changed to the cursor type set to the arrow, dot, etc. |
| 4 | exportselection | The text written inside the entry box will be automatically copied to the clipboard by default. We can set the exportselection to 0 to not copy this. |
| 5 | fg | It represents the color of the text. |
| 6 | font | It represents the font type of the text. |
| 7 | highlightbackground | It represents the color to display in the traversal highlight region when the widget does not have the input focus. |
| 8 | highlightcolor | It represents the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus. |
| 9 | highlightthickness | It represents a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus. |
| 10 | insertbackground | It represents the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget. |
| 11 | insertborderwidth | It represents a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels. |
| 12 | insertofftime | It represents a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "off" in each blink cycle. If this option is zero, then the cursor doesn't blink: it is on all the time. |

| | | |
|----|-------------------|---|
| 13 | insertontime | Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "on" in each blink cycle. |
| 14 | insertwidth | It represents the value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels. |
| 15 | justify | It specifies how the text is organized if the text contains multiple lines. |
| 16 | relief | It specifies the type of the border. Its default value is FLAT. |
| 17 | selectbackground | The background color of the selected text. |
| 18 | selectborderwidth | The width of the border to display around the selected task. |
| 19 | selectforeground | The font color of the selected task. |
| 20 | show | It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*). |
| 21 | textvariable | It is set to the instance of the StringVar to retrieve the text from the entry. |
| 22 | width | The width of the displayed text or image. |
| 23 | xscrollcommand | The entry widget can be linked to the horizontal scrollbar if we want the user to enter more text than the actual width of the widget. |

Example

```

1. #!/usr/bin/python3
2.
3. from tkinter import *
4.
5. top = Tk()
6.
7. top.geometry("400x250")
8.
9. name = Label(top, text = "Name").place(x = 30,y = 50)
10.
11. email = Label(top, text = "Email").place(x = 30, y = 90)

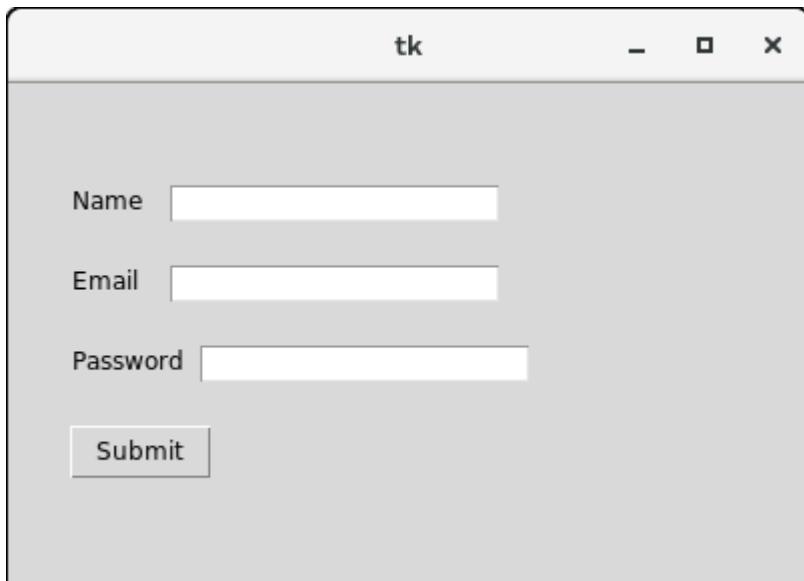
```

```

12.
13. password = Label(top, text = "Password").place(x = 30, y = 130)
14.
15. sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "blue").place(x = 30, y = 170)
16.
17. e1 = Entry(top).place(x = 80, y = 50)
18.
19.
20. e2 = Entry(top).place(x = 80, y = 90)
21.
22.
23. e3 = Entry(top).place(x = 95, y = 130)
24.
25. top.mainloop()

```

Output:



Entry widget methods

Python provides various methods to configure the data written inside the widget. There are the following methods provided by the Entry widget.

| SN | Method | Description |
|----|----------------------------|--|
| 1 | delete(first, last = none) | It is used to delete the specified characters inside the widget. |
| 2 | get() | It is used to get the text written inside the widget. |

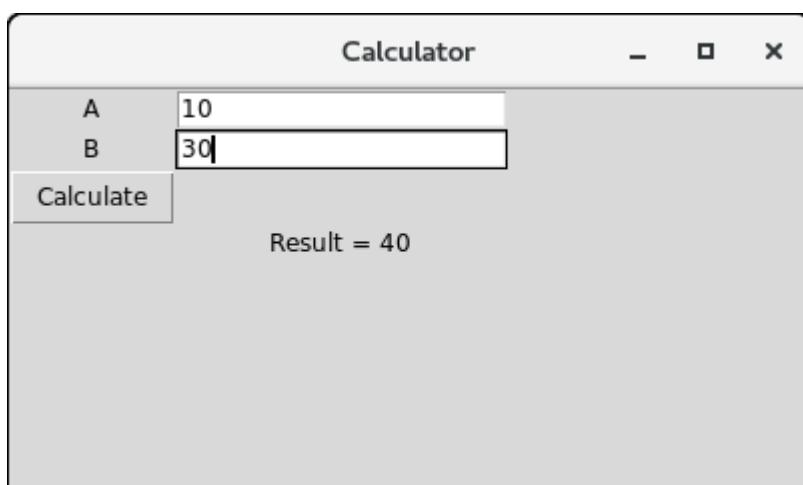
| | | |
|----|---------------------------|--|
| 3 | icursor(index) | It is used to change the insertion cursor position. We can specify the index of the character before which, the cursor to be placed. |
| 4 | index(index) | It is used to place the cursor to the left of the character written at the specified index. |
| 5 | insert(index,s) | It is used to insert the specified string before the character placed at the specified index. |
| 6 | select_adjust(index) | It includes the selection of the character present at the specified index. |
| 7 | select_clear() | It clears the selection if some selection has been done. |
| 8 | select_form(index) | It sets the anchor index position to the character specified by the index. |
| 9 | select_present() | It returns true if some text in the Entry is selected otherwise returns false. |
| 10 | select_range(start,end) | It selects the characters to exist between the specified range. |
| 11 | select_to(index) | It selects all the characters from the beginning to the specified index. |
| 12 | xview(index) | It is used to link the entry widget to a horizontal scrollbar. |
| 13 | xview_scroll(number,what) | It is used to make the entry scrollable horizontally. |

Example: A simple calculator

1. **import** tkinter as tk
2. **from** functools **import** partial
- 3.
- 4.
5. **def** call_result(label_result, n1, n2):
6. num1 = (n1.get())
7. num2 = (n2.get())
8. result = int(num1)+int(num2)
9. label_result.config(text="Result = %d" % result)
10. **return**
- 11.

```
12. root = tk.Tk()
13. root.geometry('400x200+100+200')
14.
15. root.title('Calculator')
16.
17. number1 = tk.StringVar()
18. number2 = tk.StringVar()
19.
20. labelNum1 = tk.Label(root, text="A").grid(row=1, column=0)
21.
22. labelNum2 = tk.Label(root, text="B").grid(row=2, column=0)
23.
24. labelResult = tk.Label(root)
25.
26. labelResult.grid(row=7, column=2)
27.
28. entryNum1 = tk.Entry(root, textvariable=number1).grid(row=1, column=2)
29.
30. entryNum2 = tk.Entry(root, textvariable=number2).grid(row=2, column=2)
31.
32. call_result = partial(call_result, labelResult, number1, number2)
33.
34. buttonCal = tk.Button(root, text="Calculate", command=call_result).grid(row=3, column=0)
35.
36. root.mainloop()
```

Output:



Python Tkinter Frame

Python Tkinter Frame widget is used to organize the group of widgets. It acts like a container which can be used to hold the other widgets. The rectangular areas of the screen are used to organize the widgets to the python application.

The syntax to use the Frame widget is given below.

Syntax

1. `w = Frame(parent, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|----------------------------------|--|
| 1 | <code>bd</code> | It represents the border width. |
| 2 | <code>bg</code> | The background color of the widget. |
| 3 | <code>cursor</code> | The mouse pointer is changed to the cursor type set to different values like an arrow, dot, etc. |
| 4 | <code>height</code> | The height of the frame. |
| 5 | <code>highlightbackground</code> | The color of the background color when it is under focus. |
| 6 | <code>highlightcolor</code> | The text color when the widget is under focus. |
| 7 | <code>highlightthickness</code> | It specifies the thickness around the border when the widget is under the focus. |
| 8 | <code>relief</code> | It specifies the type of the border. The default value is FLAT. |
| 9 | <code>width</code> | It represents the width of the widget. |

Example

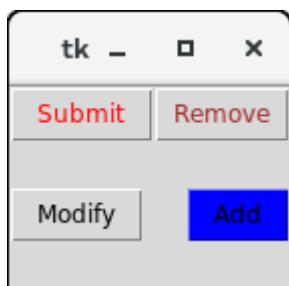
1. `from tkinter import *`
- 2.
3. `top = Tk()`
4. `top.geometry("140x100")`
5. `frame = Frame(top)`
6. `frame.pack()`
- 7.

```

8. leftframe = Frame(top)
9. leftframe.pack(side = LEFT)
10.
11. rightframe = Frame(top)
12. rightframe.pack(side = RIGHT)
13.
14. btn1 = Button(frame, text="Submit", fg="red", activebackground = "red")
15. btn1.pack(side = LEFT)
16.
17. btn2 = Button(frame, text="Remove", fg="brown", activebackground = "brown")
18. btn2.pack(side = RIGHT)
19.
20. btn3 = Button(rightframe, text="Add", fg="blue", activebackground = "blue")
21. btn3.pack(side = LEFT)
22.
23. btn4 = Button(leftframe, text="Modify", fg="black", activebackground = "white")
24. btn4.pack(side = RIGHT)
25.
26. top.mainloop()

```

Output:



Python Tkinter Label

The Label is used to specify the container box where we can place the text or images. This widget is used to provide the message to the user about other widgets used in the python application.

There are the various options which can be specified to configure the text or the part of the text shown in the Label.

The syntax to use the Label is given below.

Syntax

1. w = Label (master, options)

A list of possible options is given below.

| SN | Option | Description |
|-----------|---------------|--|
| 1 | anchor | It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space. |
| 2 | bg | The background color displayed behind the widget. |
| 3 | bitmap | It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text. |
| 4 | bd | It represents the width of the border. The default is 2 pixels. |
| 5 | cursor | The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc. |
| 6 | font | The font type of the text written inside the widget. |
| 7 | fg | The foreground color of the text written inside the widget. |
| 8 | height | The height of the widget. |
| 9 | image | The image that is to be shown as the label. |
| 10 | justify | It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification. |
| 11 | padx | The horizontal padding of the text. The default value is 1. |
| 12 | pady | The vertical padding of the text. The default value is 1. |
| 13 | relief | The type of the border. The default value is FLAT. |
| 14 | text | This is set to the string variable which may contain one or more line of text. |
| 15 | textvariable | The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly. |
| 16 | underline | We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed. |

| | | |
|----|------------|--|
| 17 | width | The width of the widget. It is specified as the number of characters. |
| 18 | wraplength | Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to this option. |

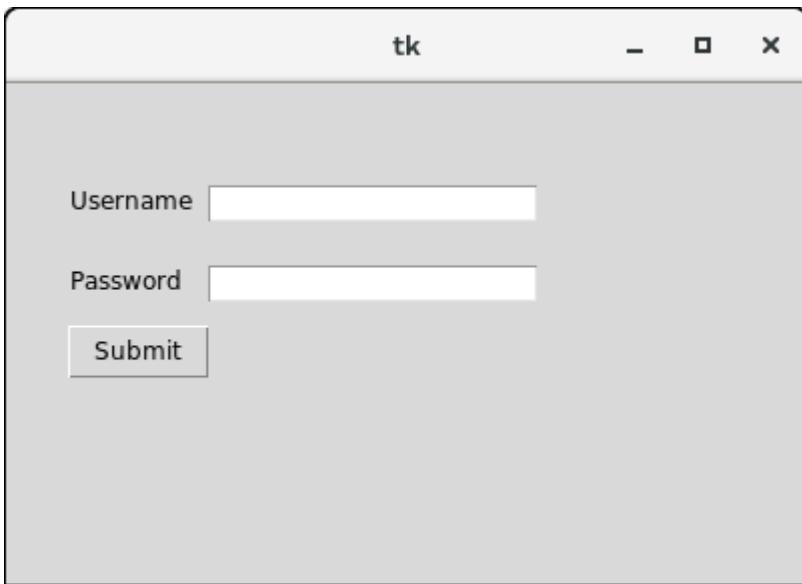
Example 1

```

1. #!/usr/bin/python3
2.
3. from tkinter import *
4.
5. top = Tk()
6.
7. top.geometry("400x250")
8.
9. #creating label
10. uname = Label(top, text = "Username").place(x = 30,y = 50)
11.
12. #creating label
13. password = Label(top, text = "Password").place(x = 30, y = 90)
14.
15.
16. sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "blue").place(x = 30, y = 120)
17.
18. e1 = Entry(top,width = 20).place(x = 100, y = 50)
19.
20.
21. e2 = Entry(top, width = 20).place(x = 100, y = 90)
22.
23.
24. top.mainloop()

```

Output:



Python Tkinter Listbox

The Listbox widget is used to display the list items to the user. We can place only text items in the Listbox and all text items contain the same font and color.

The user can choose one or more items from the list depending upon the configuration.

The syntax to use the Listbox is given below.

1. `w = Listbox(parent, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|-----------------------------|---|
| 1 | <code>bg</code> | The background color of the widget. |
| 2 | <code>bd</code> | It represents the size of the border. Default value is 2 pixel. |
| 3 | <code>cursor</code> | The mouse pointer will look like the cursor type like dot, arrow, etc. |
| 4 | <code>font</code> | The font type of the Listbox items. |
| 5 | <code>fg</code> | The color of the text. |
| 6 | <code>height</code> | It represents the count of the lines shown in the Listbox. The default value is 10. |
| 7 | <code>highlightcolor</code> | The color of the Listbox items when the widget is under focus. |

| | | |
|----|--------------------|---|
| 8 | highlightthickness | The thickness of the highlight. |
| 9 | relief | The type of the border. The default is SUNKEN. |
| 10 | selectbackground | The background color that is used to display the selected text. |
| 11 | selectmode | It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED. |
| 12 | width | It represents the width of the widget in characters. |
| 13 | xscrollcommand | It is used to let the user scroll the Listbox horizontally. |
| 14 | yscrollcommand | It is used to let the user scroll the Listbox vertically. |

Methods

There are the following methods associated with the Listbox.

| SN | Method | Description |
|----|----------------------------|--|
| 1 | activate(index) | It is used to select the lines at the specified index. |
| 2 | curselection() | It returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple. |
| 3 | delete(first, last = None) | It is used to delete the lines which exist in the given range. |
| 4 | get(first, last = None) | It is used to get the list items that exist in the given range. |
| 5 | index(i) | It is used to place the line with the specified index at the top of the widget. |
| 6 | insert(index, *elements) | It is used to insert the new lines with the specified number of elements before the specified index. |
| 7 | nearest(y) | It returns the index of the nearest line to the y coordinate of the Listbox widget. |

| | | |
|----|-----------------------------|---|
| 8 | see(index) | It is used to adjust the position of the listbox to make the lines specified by the index visible. |
| 9 | size() | It returns the number of lines that are present in the Listbox widget. |
| 10 | xview() | This is used to make the widget horizontally scrollable. |
| 11 | xview_moveto(fraction) | It is used to make the listbox horizontally scrollable by the fraction of width of the longest line present in the listbox. |
| 12 | xview_scroll(number, what) | It is used to make the listbox horizontally scrollable by the number of characters specified. |
| 13 | yview() | It allows the Listbox to be vertically scrollable. |
| 14 | yview_moveto(fraction) | It is used to make the listbox vertically scrollable by the fraction of width of the longest line present in the listbox. |
| 15 | yview_scroll (number, what) | It is used to make the listbox vertically scrollable by the number of characters specified. |

Example 1

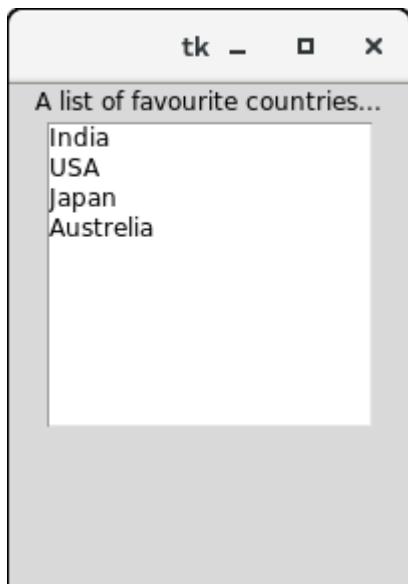
```

1. #!/usr/bin/python3
2.
3. from tkinter import *
4.
5. top = Tk()
6.
7. top.geometry("200x250")
8.
9. lbl = Label(top, text = "A list of favourite countries...")
10.
11. listbox = Listbox(top)
12.
13. listbox.insert(1, "India")
14.
15. listbox.insert(2, "USA")
16.
17. listbox.insert(3, "Japan")
18.
19. listbox.insert(4, "Australia")

```

```
20.  
21. lbl.pack()  
22. listbox.pack()  
23.  
24. top.mainloop()
```

Output:



Example 2: Deleting the active items from the list

```
1. #!/usr/bin/python3  
2.  
3. from tkinter import *  
4.  
5. top = Tk()  
6.  
7. top.geometry("200x250")  
8.  
9. lbl = Label(top, text = "A list of favourite countries...")  
10.  
11. listbox = Listbox(top)  
12.  
13. listbox.insert(1, "India")  
14.  
15. listbox.insert(2, "USA")  
16.  
17. listbox.insert(3, "Japan")  
18.  
19. listbox.insert(4, "Australia")  
20.
```

```
21. #this button will delete the selected item from the list
22.
23. btn = Button(top, text = "delete", command = lambda listbox=listbox: listbox.delete(ANCHOR))
24.
25. lbl.pack()
26.
27.
28. listbox.pack()
29.
30. btn.pack()
31. top.mainloop()
```

Output:



After pressing the delete button.



Python Tkinter Menubutton

The Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

The Menubutton is used to implement various types of menus in the python application. A Menu is associated with the Menubutton that can display the choices of the Menubutton when clicked by the user.

The syntax to use the python tkinter Menubutton is given below.

Syntax

1. w = Menubutton(Top, options)

A list of various options is given below.

| SN | Option | Description |
|----|--------------------|--|
| 1 | activebackground | The background color of the widget when the widget is under focus. |
| 2 | activeforeground | The font color of the widget text when the widget is under focus. |
| 3 | anchor | It specifies the exact position of the widget content when the widget is assigned more space than needed. |
| 4 | bg | It specifies the background color of the widget. |
| 5 | bitmap | It is set to the graphical content which is to be displayed to the widget. |
| 6 | bd | It represents the size of the border. The default value is 2 pixels. |
| 7 | cursor | The mouse pointer will be changed to the cursor type specified when the widget is under the focus. The possible value of the cursor type is arrow, or dot etc. |
| 8 | direction | It direction can be specified so that menu can be displayed to the specified direction of the button. Use LEFT, RIGHT, or ABOVE to place the widget accordingly. |
| 9 | disabledforeground | The text color of the widget when the widget is disabled. |

| | | |
|----|----------------|---|
| 10 | fg | The normal foreground color of the widget. |
| 11 | height | The vertical dimension of the Menubutton. It is specified as the number of lines. |
| 12 | highlightcolor | The highlight color shown to the widget under focus. |
| 13 | image | The image displayed on the widget. |
| 14 | justify | This specifies the exact position of the text under the widget when the text is unable to fill the width of the widget. We can use the LEFT for the left justification, RIGHT for the right justification, CENTER for the centre justification. |
| 15 | menu | It represents the menu specified with the Menubutton. |
| 16 | padx | The horizontal padding of the widget. |
| 17 | pady | The vertical padding of the widget. |
| 18 | relief | This option specifies the type of the border. The default value is RAISED. |
| 19 | state | The normal state of the Mousebutton is enabled. We can set it to DISABLED to make it unresponsive. |
| 20 | text | The text shown with the widget. |
| 21 | textvariable | We can set the control variable of string type to the text variable so that we can control the text of the widget at runtime. |
| 22 | underline | The text of the widget is not underlined by default but we can set this option to make the text of the widget underlined. |
| 23 | width | It represents the width of the widget in characters. The default value is 20. |
| 24 | wraplength | We can break the text of the widget in the number of lines so that the text contains the number of lines not greater than the specified value. |

Example

1. #!/usr/bin/python3

```
2.  
3. from tkinter import *  
4.  
5. top = Tk()  
6.  
7. top.geometry("200x250")  
8.  
9. menubutton = Menubutton(top, text = "Language", relief = FLAT)  
10.  
11. menubutton.grid()  
12.  
13. menubutton.menu = Menu(menubutton)  
14.  
15. menubutton["menu"] = menubutton.menu  
16.  
17. menubutton.menu.add_checkbutton(label = "Hindi", variable=IntVar())  
18.  
19. menubutton.menu.add_checkbutton(label = "English", variable = IntVar())  
20.  
21. menubutton.pack()  
22.  
23. top.mainloop()
```

Output:



Python Tkinter Menu

The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window. We need to create a new instance of the Menu widget and add various commands to it by using the add() method.

The syntax to use the Menu widget is given below.

Syntax

1. w = Menu(top, options)

A list of possible options is given below.

| SN | Option | Description |
|----|--------------------|--|
| 1 | activebackground | The background color of the widget when the widget is under the focus. |
| 2 | activeborderwidth | The width of the border of the widget when it is under the mouse. The default is 1 pixel. |
| 3 | activeforeground | The font color of the widget when the widget has the focus. |
| 4 | bg | The background color of the widget. |
| 5 | bd | The border width of the widget. |
| 6 | cursor | The mouse pointer is changed to the cursor type when it hovers the widget. The cursor type can be set to arrow or dot. |
| 7 | disabledforeground | The font color of the widget when it is disabled. |
| 8 | font | The font type of the text of the widget. |
| 9 | fg | The foreground color of the widget. |
| 10 | postcommand | The postcommand can be set to any of the function which is called when the mouse hovers the menu. |
| 11 | relief | The type of the border of the widget. The default type is RAISED. |
| 12 | image | It is used to display an image on the menu. |
| 13 | selectcolor | The color used to display the checkbox or radiobutton when they are selected. |

| | | |
|----|---------|---|
| 14 | tearoff | By default, the choices in the menu start taking place from position 1. If we set the tearoff = 1, then it will start taking place from 0th position. |
| 15 | title | Set this option to the title of the window if you want to change the title of the window. |

Methods

The Menu widget contains the following methods.

| SN | Option | Description |
|----|------------------------------|---|
| 1 | add_command(options) | It is used to add the Menu items to the menu. |
| 2 | add_radiobutton(options) | This method adds the radiobutton to the menu. |
| 3 | add_checkbutton(options) | This method is used to add the checkbuttons to the menu. |
| 4 | add_cascade(options) | It is used to create a hierarchical menu to the parent menu by associating the given menu to the parent menu. |
| 5 | add_seperator() | It is used to add the seperator line to the menu. |
| 6 | add(type, options) | It is used to add the specific menu item to the menu. |
| 7 | delete(startindex, endindex) | It is used to delete the menu items exist in the specified range. |
| 8 | entryconfig(index, options) | It is used to configure a menu item identified by the given index. |
| 9 | index(item) | It is used to get the index of the specified menu item. |
| 10 | insert_seperator(index) | It is used to insert a seperator at the specified index. |
| 11 | invoke(index) | It is used to invoke the associated with the choice given at the specified index. |
| 12 | type(index) | It is used to get the type of the choice specified by the index. |

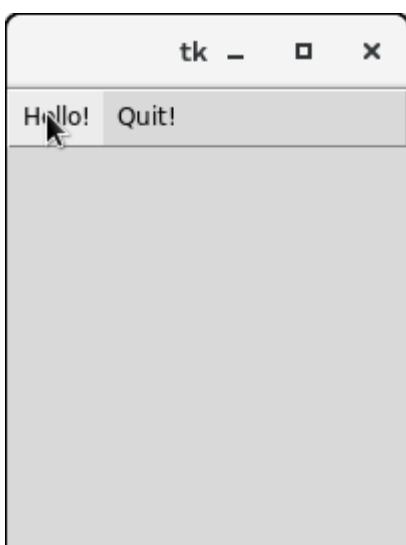
Creating a top level menu

A top-level menu can be created by instantiating the Menu widget and adding the menu items to the menu.

Example 1

```
1. #!/usr/bin/python3
2.
3. from tkinter import *
4.
5. top = Tk()
6.
7. def hello():
8.     print("hello!")
9.
10. # create a toplevel menu
11. menubar = Menu(root)
12. menubar.add_command(label="Hello!", command=hello)
13. menubar.add_command(label="Quit!", command=top.quit)
14.
15. # display the menu
16. top.config(menu=menubar)
17.
18. top.mainloop()
```

Output:

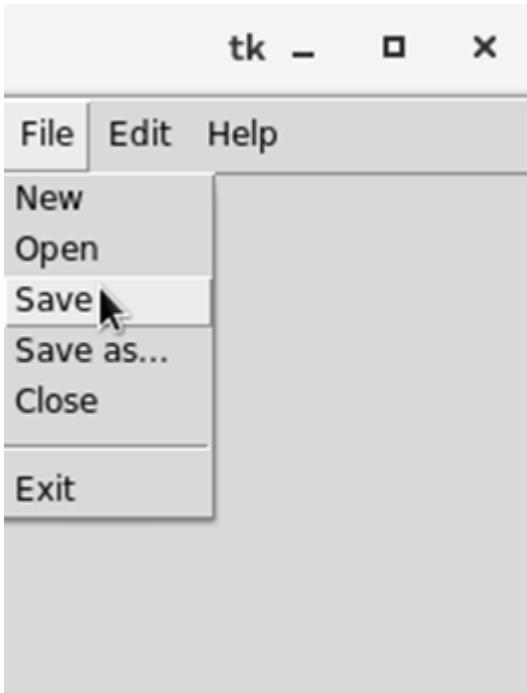


Clicking the hello Menubutton will print the hello on the console while clicking the Quit Menubutton will make an exit from the python application.

Example 2

```
1. from tkinter import Toplevel, Button, Tk, Menu
2.
3. top = Tk()
4. menubar = Menu(top)
5. file = Menu(menubar, tearoff=0)
6. file.add_command(label="New")
7. file.add_command(label="Open")
8. file.add_command(label="Save")
9. file.add_command(label="Save as...")
10. file.add_command(label="Close")
11.
12. file.add_separator()
13.
14. file.add_command(label="Exit", command=top.quit)
15.
16. menubar.add_cascade(label="File", menu=file)
17. edit = Menu(menubar, tearoff=0)
18. edit.add_command(label="Undo")
19.
20. edit.add_separator()
21.
22. edit.add_command(label="Cut")
23. edit.add_command(label="Copy")
24. edit.add_command(label="Paste")
25. edit.add_command(label="Delete")
26. edit.add_command(label="Select All")
27.
28. menubar.add_cascade(label="Edit", menu=edit)
29. help = Menu(menubar, tearoff=0)
30. help.add_command(label="About")
31. menubar.add_cascade(label="Help", menu=help)
32.
33. top.config(menu=menubar)
34. top.mainloop()
```

Output:



Python Tkinter Message

The Message widget is used to show the message to the user regarding the behaviour of the python application. The message widget shows the text messages to the user which can not be edited.

The message text contains more than one line. However, the message can only be shown in the single font.

The syntax to use the Message widget is given below.

Syntax

1. `w = Message(parent, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|--------|---|
| 1 | anchor | It is used to decide the exact position of the text within the space provided to the widget if the widget contains more space than the need of the text. The default is CENTER. |
| 2 | bg | The background color of the widget. |
| 3 | bitmap | It is used to display the graphics on the widget. It can be set to any graphical or image object. |
| 4 | bd | It represents the size of the border in the pixel. The default size is 2 pixel. |

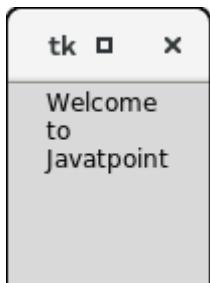
| | | |
|----|--------------|--|
| 5 | cursor | The mouse pointer is changed to the specified cursor type. The cursor type can be an arrow, dot, etc. |
| 6 | font | The font type of the widget text. |
| 7 | fg | The font color of the widget text. |
| 8 | height | The vertical dimension of the message. |
| 9 | image | We can set this option to a static image to show that onto the widget. |
| 10 | justify | This option is used to specify the alignment of multiple line of code with respect to each other. The possible values can be LEFT (left alignment), CENTER (default), and RIGHT (right alignment). |
| 11 | padx | The horizontal padding of the widget. |
| 12 | pady | The vertical padding of the widget. |
| 13 | relief | It represents the type of the border. The default type is FLAT. |
| 14 | text | We can set this option to the string so that the widget can represent the specified text. |
| 15 | textvariable | This is used to control the text represented by the widget. The textvariable can be set to the text that is shown in the widget. |
| 16 | underline | The default value of this option is -1 that represents no underline. We can set this option to an existing number to specify that nth letter of the string will be underlined. |
| 17 | width | It specifies the horizontal dimension of the widget in the number of characters (not pixel). |
| 18 | wraplength | We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters. |

Example

1. **from tkinter import ***
- 2.
3. **top = Tk()**
4. **top.geometry("100x100")**
5. **var = StringVar()**

6. msg = Message(top, text = "Welcome to Javatpoint")
- 7.
8. msg.pack()
9. top.mainloop()

Output:



Python Tkinter Radiobutton

The Radiobutton widget is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them. We can associate different methods with each of the radiobutton.

We can display the multiple line text or images on the radiobuttons. To keep track the user's selection the radiobutton, it is associated with a single variable. Each button displays a single value for that particular variable.

The syntax to use the Radiobutton is given below.

Syntax

1. w = Radiobutton(top, options)

| SN | Option | Description |
|----|------------------|---|
| 1 | activebackground | The background color of the widget when it has the focus. |
| 2 | activeforeground | The font color of the widget text when it has the focus. |
| 3 | anchor | It represents the exact position of the text within the widget if the widget contains more space than the requirement of the text. The default value is CENTER. |
| 4 | bg | The background color of the widget. |
| 5 | bitmap | It is used to display the graphics on the widget. It can be set to any graphical or image object. |

| | | |
|----|---------------------|---|
| 6 | borderwidth | It represents the size of the border. |
| 7 | command | This option is set to the procedure which must be called every-time when the state of the radiobutton is changed. |
| 8 | cursor | The mouse pointer is changed to the specified cursor type. It can be set to the arrow, dot, etc. |
| 9 | font | It represents the font type of the widget text. |
| 10 | fg | The normal foreground color of the widget text. |
| 11 | height | The vertical dimension of the widget. It is specified as the number of lines (not pixel). |
| 12 | highlightcolor | It represents the color of the focus highlight when the widget has the focus. |
| 13 | highlightbackground | The color of the focus highlight when the widget is not having the focus. |
| 14 | image | It can be set to an image object if we want to display an image on the radiobutton instead the text. |
| 15 | justify | It represents the justification of the multi-line text. It can be set to CENTER(default), LEFT, or RIGHT. |
| 16 | padx | The horizontal padding of the widget. |
| 17 | pady | The vertical padding of the widget. |
| 18 | relief | The type of the border. The default value is FLAT. |
| 19 | selectcolor | The color of the radio button when it is selected. |
| 20 | selectimage | The image to be displayed on the radiobutton when it is selected. |
| 21 | state | It represents the state of the radio button. The default state of the Radiobutton is NORMAL. However, we can set this to DISABLED to make the radiobutton unresponsive. |
| 22 | text | The text to be displayed on the radiobutton. |

| | | |
|----|--------------|---|
| 23 | textvariable | It is of String type that represents the text displayed by the widget. |
| 24 | underline | The default value of this option is -1, however, we can set this option to the number of character which is to be underlined. |
| 25 | value | The value of each radiobutton is assigned to the control variable when it is turned on by the user. |
| 26 | variable | It is the control variable which is used to keep track of the user's choices. It is shared among all the radiobuttons. |
| 27 | width | The horizontal dimension of the widget. It is represented as the number of characters. |
| 28 | wraplength | We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters. |

Methods

The radiobutton widget provides the following methods.

| SN | Method | Description |
|----|------------|---|
| 1 | deselect() | It is used to turn off the radiobutton. |
| 2 | flash() | It is used to flash the radiobutton between its active and normal colors few times. |
| 3 | invoke() | It is used to call any procedure associated when the state of a Radiobutton is changed. |
| 4 | select() | It is used to select the radiobutton. |

Example

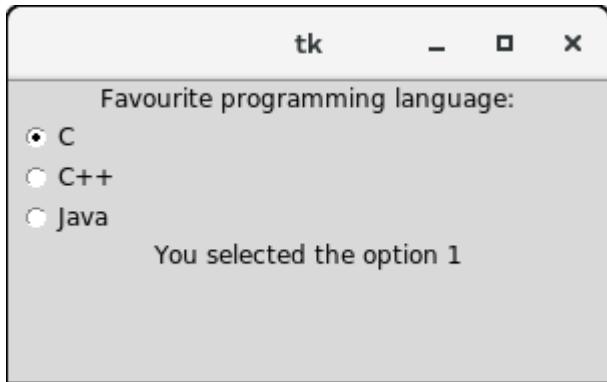
1. **from tkinter import ***
- 2.
3. **def selection():**
4. selection = "You selected the option " + str(radio.get())
5. label.config(text = selection)
- 6.
7. **top = Tk()**

```

8. top.geometry("300x150")
9. radio = IntVar()
10. lbl = Label(text = "Favourite programming language:")
11. lbl.pack()
12. R1 = Radiobutton(top, text="C", variable=radio, value=1,
13.                 command=selection)
14. R1.pack( anchor = W )
15.
16. R2 = Radiobutton(top, text="C++", variable=radio, value=2,
17.                   command=selection)
18. R2.pack( anchor = W )
19.
20. R3 = Radiobutton(top, text="Java", variable=radio, value=3,
21.                   command=selection)
22. R3.pack( anchor = W )
23.
24. label = Label(top)
25. label.pack()
26. top.mainloop()

```

Output:



Python Tkinter Scale

The Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them.

We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative to the Entry widget when the user is forced to select only one value from the given range of values.

The syntax to use the Scale widget is given below.

Syntax

1. `w = Scale(top, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|---------------------|--|
| 1 | activebackground | The background color of the widget when it has the focus. |
| 2 | bg | The background color of the widget. |
| 3 | bd | The border size of the widget. The default is 2 pixel. |
| 4 | command | It is set to the procedure which is called each time when we move the slider. If the slider is moved rapidly, the callback is done when it settles. |
| 5 | cursor | The mouse pointer is changed to the cursor type assigned to this option. It can be an arrow, dot, etc. |
| 6 | digits | If the control variable used to control the scale data is of string type, this option is used to specify the number of digits when the numeric scale is converted to a string. |
| 7 | font | The font type of the widget text. |
| 8 | fg | The foreground color of the text. |
| 9 | from_ | It is used to represent one end of the widget range. |
| 10 | highlightbackground | The highlight color when the widget doesn't have the focus. |
| 11 | highlightcolor | The highlight color when the widget has the focus. |
| 12 | label | This can be set to some text which can be shown as a label with the scale. It is shown in the top left corner if the scale is horizontal or the top right corner if the scale is vertical. |
| 13 | length | It represents the length of the widget. It represents the X dimension if the scale is horizontal or y dimension if the scale is vertical. |

| | | |
|----|--------------|---|
| 14 | orient | It can be set to horizontal or vertical depending upon the type of the scale. |
| 15 | relief | It represents the type of the border. The default is FLAT. |
| 16 | repeatdelay | This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms. |
| 17 | resolution | It is set to the smallest change which is to be made to the scale value. |
| 18 | showvalue | The value of the scale is shown in the text form by default. We can set this option to 0 to suppress the label. |
| 19 | sliderlength | It represents the length of the slider window along the length of the scale. The default is 30 pixels. However, we can change it to the appropriate value. |
| 20 | state | The scale widget is active by default. We can set this to DISABLED to make it unresponsive. |
| 21 | takefocus | The focus cycles through the scale widgets by default. We can set this option to 0 if we don't want this to happen. |
| 22 | tickinterval | The scale values are displayed on the multiple of the specified tick interval. The default value of the tickinterval is 0. |
| 23 | to | It represents a float or integer value that specifies the other end of the range represented by the scale. |
| 24 | troughcolor | It represents the color of the through. |
| 25 | variable | It represents the control variable for the scale. |
| 26 | width | It represents the width of the through part of the widget. |

Methods

| SN | Method | Description |
|----|--------|-------------|
| | | |

| | | |
|---|------------|---|
| 1 | get() | It is used to get the current value of the scale. |
| 2 | set(value) | It is used to set the value of the scale. |

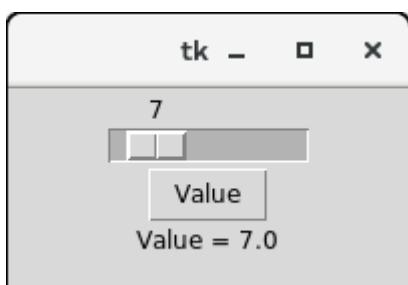
Example

```

1. from tkinter import *
2.
3. def select():
4.     sel = "Value = " + str(v.get())
5.     label.config(text = sel)
6.
7. top = Tk()
8. top.geometry("200x100")
9. v = DoubleVar()
10. scale = Scale( top, variable = v, from_ = 1, to = 50, orient = HORIZONTAL)
11. scale.pack(anchor=CENTER)
12.
13. btn = Button(top, text="Value", command=select)
14. btn.pack(anchor=CENTER)
15.
16. label = Label(top)
17. label.pack()
18.
19. top.mainloop()

```

Output:



Python Tkinter Scrollbar

The scrollbar widget is used to scroll down the content of the other widgets like listbox, text, and canvas. However, we can also create the horizontal scrollbars to the Entry widget.

The syntax to use the Scrollbar widget is given below.

Syntax

1. w = Scrollbar(top, options)

A list of possible options is given below.

| SN | Option | Description |
|----|---------------------|---|
| 1 | activebackground | The background color of the widget when it has the focus. |
| 2 | bg | The background color of the widget. |
| 3 | bd | The border width of the widget. |
| 4 | command | It can be set to the procedure associated with the list which can be called each time when the scrollbar is moved. |
| 5 | cursor | The mouse pointer is changed to the cursor type set to this option which can be an arrow, dot, etc. |
| 6 | elementborderwidth | It represents the border width around the arrow heads and slider. The default value is -1. |
| 7 | highlightbackground | The focus highlightcolor when the widget doesn't have the focus. |
| 8 | highlightcolor | The focus highlightcolor when the widget has the focus. |
| 9 | highlightthickness | It represents the thickness of the focus highlight. |
| 10 | jump | It is used to control the behavior of the scroll jump. If it set to 1, then the callback is called when the user releases the mouse button. |
| 11 | orient | It can be set to HORIZONTAL or VERTICAL depending upon the orientation of the scrollbar. |
| 12 | repeatdelay | This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms. |
| 13 | repeatinterval | The default value of the repeat interval is 100. |

| | | |
|----|-------------|--|
| 14 | takefocus | We can tab the focus through this widget by default. We can set this option to 0 if we don't want this behavior. |
| 15 | troughcolor | It represents the color of the trough. |
| 16 | width | It represents the width of the scrollbar. |

Methods

The widget provides the following methods.

| SN | Method | Description |
|----|------------------|---|
| 1 | get() | It returns the two numbers a and b which represents the current position of the scrollbar. |
| 2 | set(first, last) | It is used to connect the scrollbar to the other widget w. The yscrollcommand or xscrollcommand of the other widget to this method. |

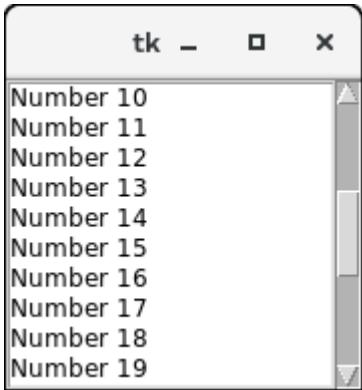
Example

```

1. from tkinter import *
2.
3. top = Tk()
4. sb = Scrollbar(top)
5. sb.pack(side = RIGHT, fill = Y)
6.
7. mylist = Listbox(top, yscrollcommand = sb.set )
8.
9. for line in range(30):
10.    mylist.insert(END, "Number " + str(line))
11.
12. mylist.pack( side = LEFT )
13. sb.config( command = mylist.yview )
14.
15. mainloop()

```

Output:



Python Tkinter Text

The Text widget is used to show the text data on the Python application. However, Tkinter provides us the Entry widget which is used to implement the single line text box.

The Text widget is used to display the multi-line formatted text with various styles and attributes. The Text widget is mostly used to provide the text editor to the user.

The Text widget also facilitates us to use the marks and tabs to locate the specific sections of the Text. We can also use the windows and images with the Text as it can also be used to display the formatted text.

The syntax to use the Text widget is given below.

Syntax

1. `w = Text(top, options)`

A list of possible options that can be used with the Text widget is given below.

| SN | Option | Description |
|----|------------------------------|--|
| 1 | <code>bg</code> | The background color of the widget. |
| 2 | <code>bd</code> | It represents the border width of the widget. |
| 3 | <code>cursor</code> | The mouse pointer is changed to the specified cursor type, i.e. arrow, dot, etc. |
| 4 | <code>exportselection</code> | The selected text is exported to the selection in the window manager. We can set this to 0 if we don't want the text to be exported. |
| 5 | <code>font</code> | The font type of the text. |

| | | |
|----|---------------------|---|
| 6 | fg | The text color of the widget. |
| 7 | height | The vertical dimension of the widget in lines. |
| 8 | highlightbackground | The highlightcolor when the widget doesn't has the focus. |
| 9 | highlightthickness | The thickness of the focus highlight. The default value is 1. |
| 10 | highlighcolor | The color of the focus highlight when the widget has the focus. |
| 11 | insertbackground | It represents the color of the insertion cursor. |
| 12 | insertborderwidth | It represents the width of the border around the cursor. The default is 0. |
| 13 | insertofftime | The time amount in Milliseconds during which the insertion cursor is off in the blink cycle. |
| 14 | insertontime | The time amount in Milliseconds during which the insertion cursor is on in the blink cycle. |
| 15 | insertwidth | It represents the width of the insertion cursor. |
| 16 | padx | The horizontal padding of the widget. |
| 17 | pady | The vertical padding of the widget. |
| 18 | relief | The type of the border. The default is SUNKEN. |
| 19 | selectbackground | The background color of the selected text. |
| 20 | selectborderwidth | The width of the border around the selected text. |
| 21 | spacing1 | It specifies the amount of vertical space given above each line of the text. The default is 0. |
| 22 | spacing2 | This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. The default is 0. |
| 23 | spacing3 | It specifies the amount of vertical space to insert below each line of the text. |

| | | |
|----|----------------|---|
| 24 | state | If the state is set to DISABLED, the widget becomes unresponsive to the mouse and keyboard unresponsive. |
| 25 | tabs | This option controls how the tab character is used to position the text. |
| 26 | width | It represents the width of the widget in characters. |
| 27 | wrap | This option is used to wrap the wider lines into multiple lines. Set this option to the WORD to wrap the lines after the word that fit into the available space. The default value is CHAR which breaks the line which gets too wider at any character. |
| 28 | xscrollcommand | To make the Text widget horizontally scrollable, we can set this option to the set() method of Scrollbar widget. |
| 29 | yscrollcommand | To make the Text widget vertically scrollable, we can set this option to the set() method of Scrollbar widget. |

Methods

We can use the following methods with the Text widget.

| SN | Method | Description |
|----|------------------------------|--|
| 1 | delete(startindex, endindex) | This method is used to delete the characters of the specified range. |
| 2 | get(startindex, endindex) | It returns the characters present in the specified range. |
| 3 | index(index) | It is used to get the absolute index of the specified index. |
| 4 | insert(index, string) | It is used to insert the specified string at the given index. |
| 5 | see(index) | It returns a boolean value true or false depending upon whether the text at the specified index is visible or not. |

Mark handling methods

Marks are used to bookmark the specified position between the characters of the associated text.

| SN | Method | Description |
|----|-----------------------------|---|
| 1 | index(mark) | It is used to get the index of the specified mark. |
| 2 | mark_gravity(mark, gravity) | It is used to get the gravity of the given mark. |
| 3 | mark_names() | It is used to get all the marks present in the Text widget. |
| 4 | mark_set(mark, index) | It is used to inform a new position of the given mark. |
| 5 | mark_unset(mark) | It is used to remove the given mark from the text. |

Tag handling methods

The tags are the names given to the separate areas of the text. The tags are used to configure the different areas of the text separately. The list of tag-handling methods along with the description is given below.

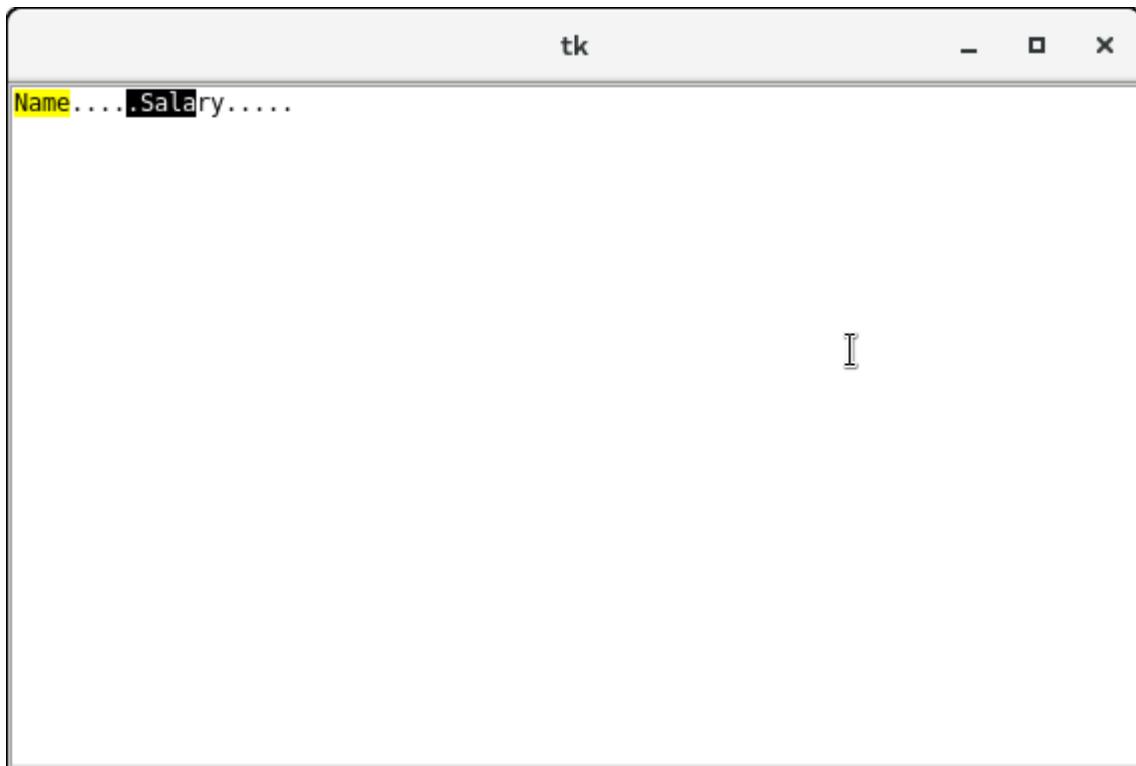
| SN | Method | Description |
|----|---|---|
| 1 | tag_add(tagname, startindex, endindex) | This method is used to tag the string present in the specified range. |
| 2 | tag_config | This method is used to configure the tag properties. |
| 3 | tag_delete(tagname) | This method is used to delete a given tag. |
| 4 | tag_remove(tagname, startindex, endindex) | This method is used to remove a tag from the specified range. |

Example

1. `from tkinter import *`
- 2.
3. `top = Tk()`
4. `text = Text(top)`
5. `text.insert(INSERT, "Name.....")`
6. `text.insert(END, "Salary.....")`
- 7.
8. `text.pack()`
- 9.

```
10. text.tag_add("Write Here", "1.0", "1.4")
11. text.tag_add("Click Here", "1.8", "1.13")
12.
13. text.tag_config("Write Here", background="yellow", foreground="black")
14. text.tag_config("Click Here", background="black", foreground="white")
15.
16. top.mainloop()
```

Output:



Tkinter Toplevel

The Toplevel widget is used to create and display the toplevel windows which are directly managed by the window manager. The toplevel widget may or may not have the parent window on the top of them.

The toplevel widget is used when a python application needs to represent some extra information, pop-up, or the group of widgets on the new window.

The toplevel windows have the title bars, borders, and other window decorations.

The syntax to use the Toplevel widget is given below.

Syntax

1. w = Toplevel(options)

A List of possible options is given below.

| SN | Options | Description |
|-----------|----------------|--|
| 1 | bg | It represents the background color of the window. |
| 2 | bd | It represents the border size of the window. |
| 3 | cursor | The mouse pointer is changed to the cursor type set to the arrow, dot, etc. when the mouse is in the window. |
| 4 | class_ | The text selected in the text widget is exported to be selected to the window manager. We can set this to 0 to make this behavior false. |
| 5 | font | The font type of the text inserted into the widget. |
| 6 | fg | The foreground color of the widget. |
| 7 | height | It represents the height of the window. |
| 8 | relief | It represents the type of the window. |
| 9 | width | It represents the width of the window, |

Methods

The methods associated with the Toplevel widget is given in the following list.

| SN | Method | Description |
|-----------|--------------------------|--|
| 1 | deiconify() | This method is used to display the window. |
| 2 | frame() | It is used to show a system dependent window identifier. |
| 3 | group(window) | It is used to add this window to the specified window group. |
| 4 | iconify() | It is used to convert the toplevel window into an icon. |
| 5 | protocol(name, function) | It is used to mention a function which will be called for the specific protocol. |

| | | |
|----|--------------------------|---|
| 6 | state() | It is used to get the current state of the window. Possible values are normal, iconic, withdrawn, and icon. |
| 7 | transient([master]) | It is used to convert this window to a transient window (temporary). |
| 8 | withdraw() | It is used to delete the window but doesn't destroy it. |
| 9 | maxsize(width, height) | It is used to declare the maximum size for the window. |
| 10 | minsize(width, height) | It is used to declare the minimum size for the window. |
| 11 | positionfrom(who) | It is used to define the position controller. |
| 12 | resizable(width, height) | It is used to control whether the window can be resizable or not. |
| 13 | sizefrom(who) | It is used to define the size controller. |
| 14 | title(string) | It is used to define the title for the window. |

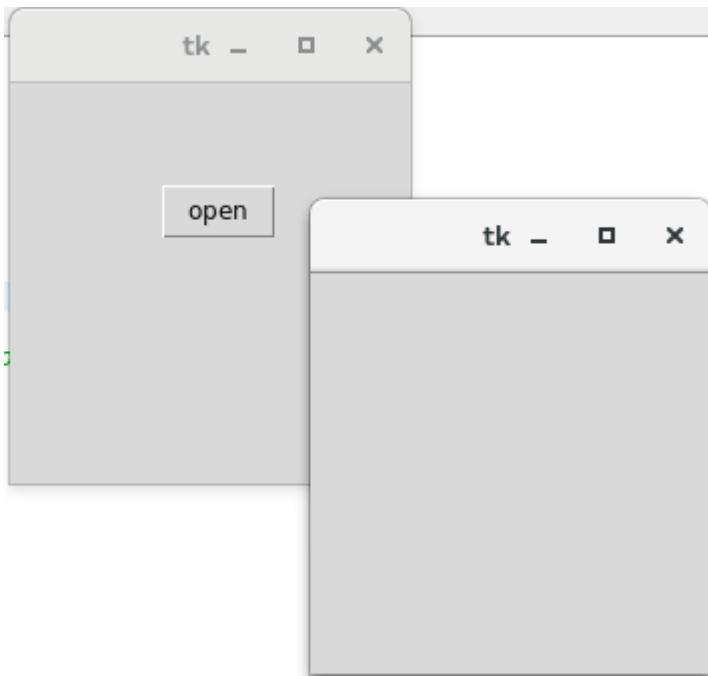
Example

```

1. from tkinter import *
2.
3. root = Tk()
4.
5. root.geometry("200x200")
6.
7. def open():
8.     top = Toplevel(root)
9.     top.mainloop()
10.
11. btn = Button(root, text = "open", command = open)
12.
13. btn.place(x=75,y=50)
14.
15. root.mainloop()

```

Output:



Python Tkinter Spinbox

The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one.

It is used in the case where a user is given some fixed number of values to choose from.

We can use various options with the Spinbox to decorate the widget. The syntax to use the Spinbox is given below.

Syntax

1. `w = Spinbox(top, options)`

A list of possible options is given below.

| SN | Option | Description |
|----|------------------|--|
| 1 | activebackground | The background color of the widget when it has the focus. |
| 2 | bg | The background color of the widget. |
| 3 | bd | The border width of the widget. |
| 4 | command | The associated callback with the widget which is called each time the state of the widget is called. |
| 5 | cursor | The mouse pointer is changed to the cursor type assigned to this option. |

| | | |
|----|--------------------|--|
| 6 | disabledbackground | The background color of the widget when it is disabled. |
| 7 | disabledforeground | The foreground color of the widget when it is disabled. |
| 8 | fg | The normal foreground color of the widget. |
| 9 | font | The font type of the widget content. |
| 10 | format | This option is used for the format string. It has no default value. |
| 11 | from_ | It is used to show the starting range of the widget. |
| 12 | justify | It is used to specify the justification of the multi-line widget content. The default is LEFT. |
| 13 | relief | It is used to specify the type of the border. The default is SUNKEN. |
| 14 | repeatdelay | This option is used to control the button auto repeat. The value is given in milliseconds. |
| 15 | repeatinterval | It is similar to repeatdelay. The value is given in milliseconds. |
| 16 | state | It represents the state of the widget. The default is NORMAL. The possible values are NORMAL, DISABLED, or "readonly". |
| 17 | textvariable | It is like a control variable which is used to control the behaviour of the widget text. |
| 18 | to | It specifies the maximum limit of the widget value. The other is specified by the from_ option. |
| 19 | validate | This option controls how the widget value is validated. |
| 20 | validatecommand | It is associated to the function callback which is used for the validation of the widget content. |
| 21 | values | It represents the tuple containing the values for this widget. |
| 22 | vcmd | It is same as validation command. |
| 23 | width | It represents the width of the widget. |

| | | |
|----|----------------|---|
| 24 | wrap | This option wraps up the up and down button the Spinbox. |
| 25 | xscrollcommand | This options is set to the set() method of scrollbar to make this widget horizontally scrollable. |

Methods

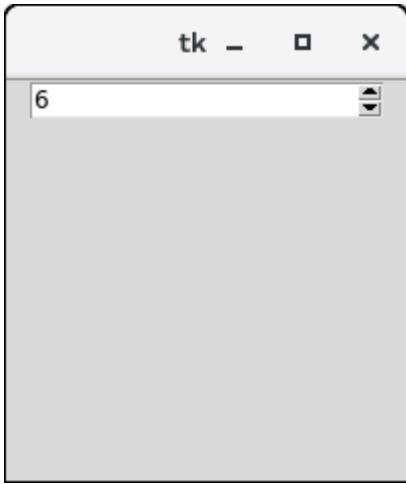
There are the following methods associated with the widget.

| SN | Option | Description |
|----|------------------------------|--|
| 1 | delete(startindex, endindex) | This method is used to delete the characters present at the specified range. |
| 2 | get(startindex, endindex) | It is used to get the characters present in the specified range. |
| 3 | identify(x, y) | It is used to identify the widget's element within the specified range. |
| 4 | index(index) | It is used to get the absolute value of the given index. |
| 5 | insert(index, string) | This method is used to insert the string at the specified index. |
| 6 | invoke(element) | It is used to invoke the callback associated with the widget. |

Example

1. **from** tkinter **import** *
- 2.
3. top = Tk()
- 4.
5. top.geometry("200x200")
- 6.
7. spin = Spinbox(top, from_= 0, to = 25)
- 8.
9. spin.pack()
- 10.
11. top.mainloop()

Output:



Tkinter PanedWindow

The PanedWindow widget acts like a Container widget which contains one or more child widgets (panes) arranged horizontally or vertically. The child panes can be resized by the user, by moving the separator lines known as sashes by using the mouse.

Each pane contains only one widget. The PanedWindow is used to implement the different layouts in the python applications.

The syntax to use the PanedWindow is given below.

Syntax

1. w= PanedWindow(master, options)

A list of possible options is given below.

| SN | Option | Description |
|----|-------------|--|
| 1 | bg | It represents the background color of the widget when it doesn't have the focus. |
| 2 | bd | It represents the 3D border size of the widget. The default option specifies that the trough contains no border whereas the arrowheads and slider contain the 2-pixel border size. |
| 3 | borderwidth | It represents the border width of the widget. The default is 2 pixel. |

| | | |
|----------|------------|---|
| 4 | cursor | The mouse pointer is changed to the specified cursor type when it is over the window. |
| 5 | handlepad | This option represents the distance between the handle and the end of the sash. For the horizontal orientation, it is the distance between the top of the sash and the handle. The default is 8 pixels. |
| 6 | handlesize | It represents the size of the handle. The default size is 8 pixels. However, the handle will always be a square. |
| 7 | height | It represents the height of the widget. If we do not specify the height, it will be calculated by the height of the child window. |
| 8 orient | | The orient will be set to HORIZONTAL if we want to place the child windows side by side. It can be set to VERTICAL if we want to place the child windows from top to bottom. |
| 9 | relief | It represents the type of the border. The default is FLAT. |
| 10 | sashpad | It represents the padding to be done around each sash. The default is 0. |
| 11 | sashrelief | It represents the type of the border around each of the sash. The default is FLAT. |
| 12 | sashwidth | It represents the width of the sash. The default is 2 pixels. |
| 13 | showhandle | It is set to True to display the handles. The default value is false. |
| 14 | Width | It represents the width of the widget. If we don't specify the width of the |

| | | |
|--|--|---|
| | | widget, it will be calculated by the size of the child widgets. |
|--|--|---|

Methods

There are the following methods that are associated with the PanedWindow.

| SN | Method | Description |
|----|---------------------------|---|
| 1 | add(child, options) | It is used to add a window to the parent window. |
| 2 | get(startindex, endindex) | This method is used to get the text present at the specified range. |
| 3 | config(options) | It is used to configure the widget with the specified options. |

Example

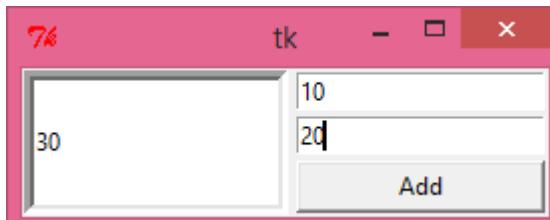
```

1. #!/usr/bin/python3
2. from tkinter import *
3.
4. def add():
5.     a = int(e1.get())
6.     b = int(e2.get())
7.     leftdata = str(a+b)
8.     left.insert(1,leftdata)
9.
10. w1 = PanedWindow()
11. w1.pack(fill = BOTH, expand = 1)
12.
13. left = Entry(w1, bd = 5)
14. w1.add(left)
15.
16. w2 = PanedWindow(w1, orient = VERTICAL)
17. w1.add(w2)
18.
19. e1 = Entry(w2)
20. e2 = Entry(w2)
21.
22. w2.add(e1)
23. w2.add(e2)

```

```
24.  
25. bottom = Button(w2, text = "Add", command = add)  
26. w2.add(bottom)  
27.  
28. mainloop()
```

Output:



Tkinter LabelFrame

The LabelFrame widget is used to draw a border around its child widgets. We can also display the title for the LabelFrame widget. It acts like a container which can be used to group the number of interrelated widgets such as Radiobuttons.

This widget is a variant of the Frame widget which has all the features of a frame. It also can display a label.

The syntax to use the LabelFrame widget is given below.

Syntax

1. w = LabelFrame(top, options)

A list of options is given below.

| SN | Option | Description |
|----|----------|--|
| 1 | bg | The background color of the widget. |
| 2 | bd | It represents the size of the border shown around the indicator. The default is 2 pixels. |
| 3 | Class | The default value of the class is LabelFrame. |
| 4 | colormap | This option is used to specify which colormap is to be used for this widget. By colormap, we mean the 256 colors that are used to form the graphics. With this option, we can reuse the colormap of another window on this widget. |

| | | |
|----|---------------------|---|
| 5 | container | If this is set to true, the LabelFrame becomes the container widget. The default value is false. |
| 6 | cursor | It can be set to a cursor type, i.e. arrow, dot, etc. the mouse pointer is changed to the cursor type when it is over the widget. |
| 7 | fg | It represents the foreground color of the widget. |
| 8 | font | It represents the font type of the widget text. |
| 9 | height | It represents the height of the widget. |
| 10 | labelAnchor | It represents the exact position of the text within the widget. The default is NW(north-west) |
| 11 | labelwidget | It represents the widget to be used for the label. The frame uses the text for the label if no value specified. |
| 12 | highlightbackground | The color of the focus highlight border when the widget doesn't have the focus. |
| 13 | highlightcolor | The color of the focus highlight when the widget has the focus. |
| 14 | highlightthickness | The width of the focus highlight border. |
| 15 | padx | The horizontal padding of the widget. |
| 16 | pady | The vertical padding of the widget. |
| 17 | relief | It represents the border style. The default value is GROOVE. |
| 18 | text | It represents the string containing the label text. |
| 19 | width | It represents the width of the frame. |

Example

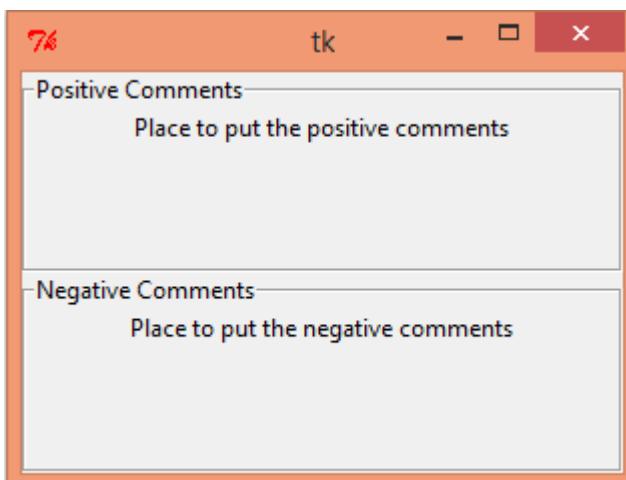
1. `#!/usr/bin/python3`
2. `from tkinter import *`
- 3.
4. `top = Tk()`
5. `top.geometry("300x200")`
- 6.

```

7. labelframe1 = LabelFrame(top, text="Positive Comments")
8. labelframe1.pack(fill="both", expand="yes")
9.
10. toplabel = Label(labelframe1, text="Place to put the positive comments")
11. toplabel.pack()
12.
13. labelframe2 = LabelFrame(top, text = "Negative Comments")
14. labelframe2.pack(fill="both", expand = "yes")
15.
16. bottomlabel = Label(labelframe2, text = "Place to put the negative comments")
17. bottomlabel.pack()
18.
19. top.mainloop()

```

Output:



Tkinter messagebox

The messagebox module is used to display the message boxes in the python applications. There are the various functions which are used to display the relevant messages depending upon the application requirements.

The syntax to use the messagebox is given below.

Syntax

1. messagebox.function_name(title, message [, options])

Parameters

- **function_name:** It represents an appropriate message box function.
- **title:** It is a string which is shown as a title of a message box.
- **message:** It is the string to be displayed as a message on the message box.

- o **options:** There are various options which can be used to configure the message dialog box.

The two options that can be used are default and parent.

1. default

The default option is used to mention the types of the default button, i.e. ABORT, RETRY, or IGNORE in the message box.

2. parent

The parent option specifies the parent window on top of which, the message box is to be displayed.

There is one of the following functions used to show the appropriate message boxes. All the functions are used with the same syntax but have the specific functionalities.

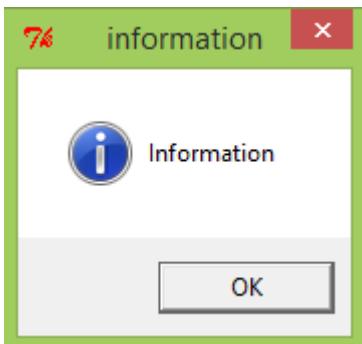
1. showinfo()

The showinfo() messagebox is used where we need to show some relevant information to the user.

Example

```
1. #!/usr/bin/python3
2.
3. from tkinter import *
4.
5. from tkinter import messagebox
6.
7. top = Tk()
8.
9. top.geometry("100x100")
10.
11. messagebox.showinfo("information","Information")
12.
13. top.mainloop()
```

Output:



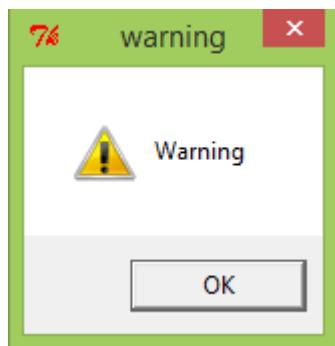
2. showwarning()

This method is used to display the warning to the user. Consider the following example.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3.
4. from tkinter import messagebox
5.
6. top = Tk()
7. top.geometry("100x100")
8. messagebox.showwarning("warning","Warning")
9.
10. top.mainloop()
```

Output:



3. showerror()

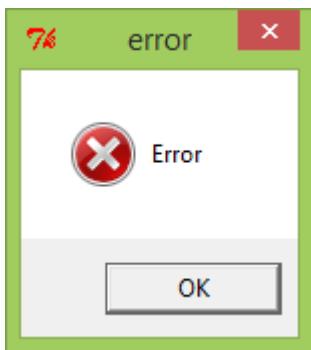
This method is used to display the error message to the user. Consider the following example.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3. from tkinter import messagebox
```

```
4.  
5. top = Tk()  
6. top.geometry("100x100")  
7. messagebox.showerror("error","Error")  
8. top.mainloop()
```

Output:



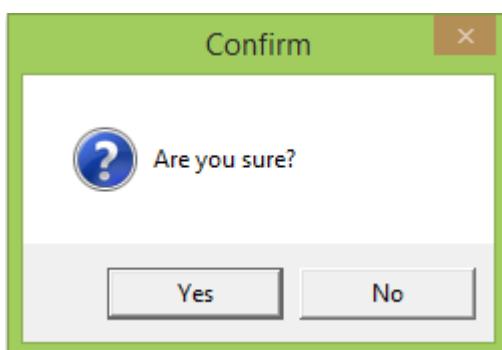
4. askquestion()

This method is used to ask some question to the user which can be answered in yes or no. Consider the following example.

Example

```
1. #!/usr/bin/python3  
2. from tkinter import *  
3. from tkinter import messagebox  
4.  
5. top = Tk()  
6. top.geometry("100x100")  
7. messagebox.askquestion("Confirm","Are you sure?")  
8. top.mainloop()
```

Output:



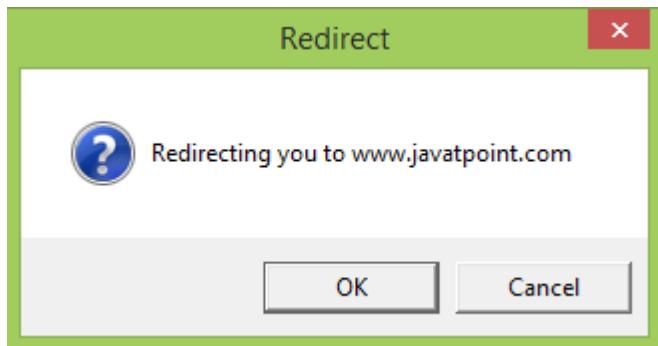
5. askokcancel()

This method is used to confirm the user's action regarding some application activity. Consider the following example.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3. from tkinter import messagebox
4.
5. top = Tk()
6. top.geometry("100x100")
7. messagebox.askokcancel("Redirect","Redirecting you to www.javatpoint.com")
8. top.mainloop()
```

Output:



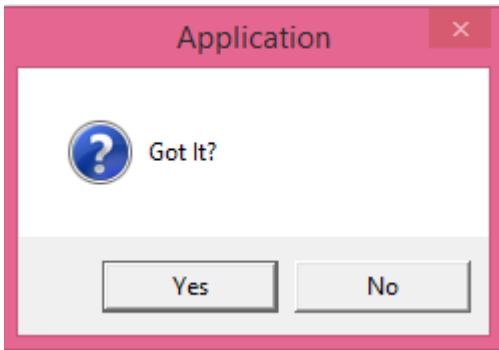
6. askyesno()

This method is used to ask the user about some action to which, the user can answer in yes or no. Consider the following example.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3. from tkinter import messagebox
4.
5. top = Tk()
6. top.geometry("100x100")
7. messagebox.askyesno("Application","Got It?")
8. top.mainloop()
```

Output:



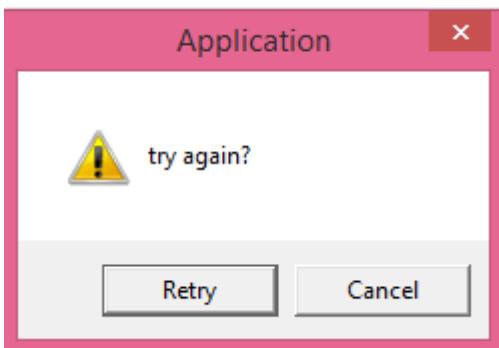
7. askretrycancel()

This method is used to ask the user about doing a particular task again or not. Consider the following example.

Example

```
1. #!/usr/bin/python3
2. from tkinter import *
3. from tkinter import messagebox
4.
5. top = Tk()
6. top.geometry("100x100")
7. messagebox.askretrycancel("Application","try again?")
8.
9. top.mainloop()
```

Output:



Introduction

In this section of the tutorial, we are going to build a real-time most popular python application known as website blocker.

This application can be used to block the websites so that the user can not open them during the specific period.

Let's discuss how can we build such an application by using python.

Objective

The objective of Python website blocker is to block some certain websites which can distract the user during the specified amount of time.

In this, we will block the access to the list of some particular websites during the working hours so that the user can only access those websites during the free time only.

The working time in this python application is considered from 9 AM to 5 PM. The time period except that time will be considered as free time.

Process

To block the access to a specific website on the computer, we need to configure the **hosts** file.

The hosts file

The hosts file is a local file which was used to map hostnames to IP addresses in the old days. Although the DNS service is used nowadays to map the hostnames to the IP addresses, the host file is still very complex and can be used to configure the mapping of the IP addresses locally.

Location of hosts file

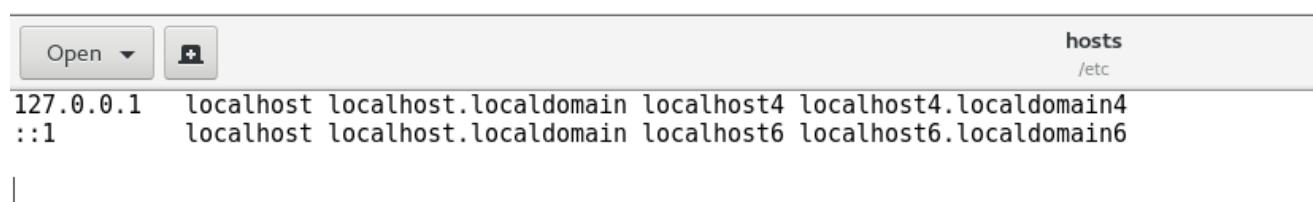
The location of the hosts file varies from operating system to operating system.

Windows: C:\Windows\System32\drivers\etc

mac and Linux: /etc/hosts

Configuration

Since the hosts file contains the mapping between the host names and IP addresses, let's look at the content of our hosts file first stored as **/etc/hosts** as we use CentOS 7 (Linux).



A screenshot of a text editor window showing the contents of the /etc/hosts file. The window has a toolbar with 'Open' and a plus icon. The file is named 'hosts' and is located in '/etc'. The content of the file is as follows:

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1      localhost localhost.localdomain localhost6 localhost6.localdomain6
```

As we can see in the above image, the mappings are saved for the localhost with the IP address 127.0.0.1.

Similarly, we can redirect any hostname back to the localhost IP (127.0.0.1). It will block the access to that hostname and redirect that to localhost on each request.

For testimony, let's edit the content of hosts file and add some of the following lines to it. To make changes to the **/etc/hosts** file, we will need to change its permissions.

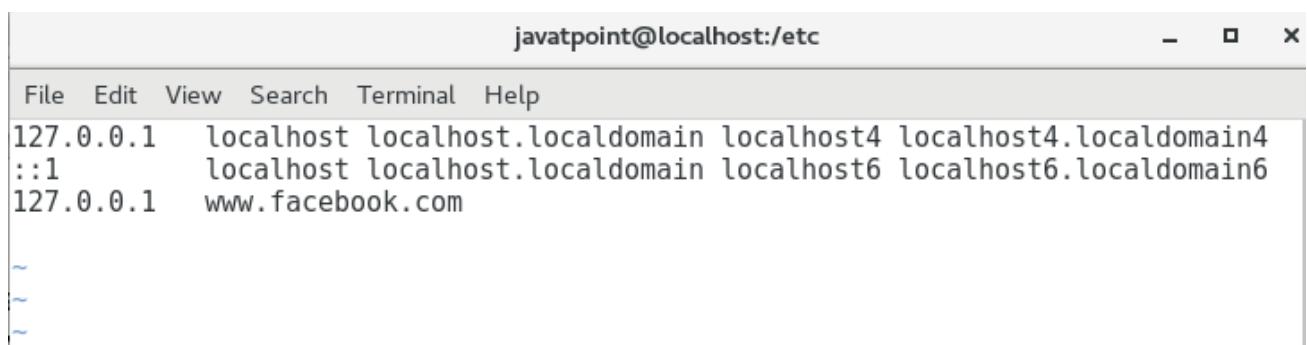
Run the following command on the terminal to change the permissions.

1. \$ sudo chmod 777 /etc/hosts

Now add the following line to the file to block the access to facebook.com (for example).

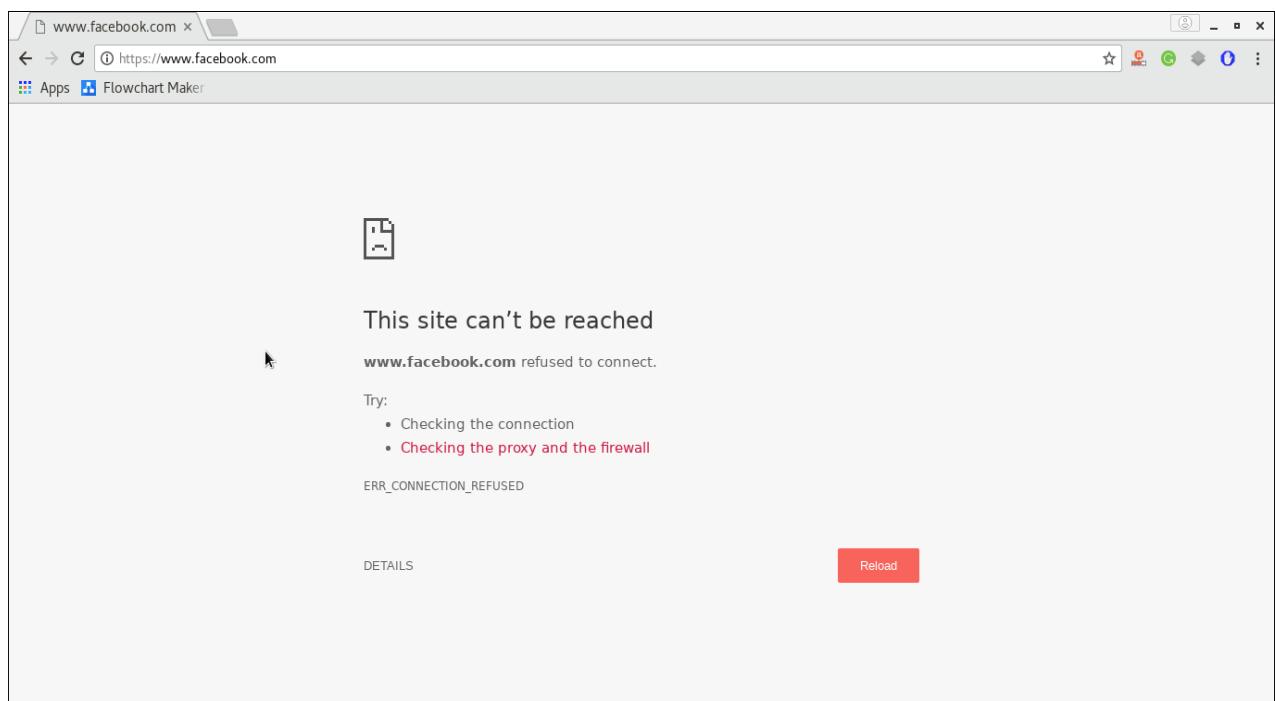
1. 127.0.0.1 www.facebook.com

As shown in the image below.



```
javatpoint@localhost:/etc
File Edit View Search Terminal Help
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
127.0.0.1 www.facebook.com
```

Save the file and try to open the www.facebook.com using the browser.



As shown in the above figure, it is refused to connect.

We have completed our task by manually editing the hosts file. We haven't achieved our objective yet. Our objective is to block access to some particular websites (for example, facebook) during working hours only (9 AM to 5 PM).

It needs a python script which keeps adding the necessary lines to the hosts file during a particular period.

In this section of the tutorial, we will build such python script which keeps editing the hosts file during the working hours. We will also deploy that script at the OS startup so that it doesn't need any external execution.

Requirements

We need to know the following python modules to build the python website blocker.

1. **file handling:** file handling is used to do the modifications to the hosts file.
2. **time:** The time module is used to control the frequency of the modifications to the hosts file.
3. **datetime:** The datetime module is used to keep track of the free time and working time.

Building python script

Let's start building the python script that can run at system startup to block the access to the particular websites. Open PyCharm to edit the code or you can use any IDE you want.

Create a new Python Script named as **web-blocker.py**. To make the process understandable to you, we will build this script step by step. So let's start coding by setting up all the required variables.

Setting up the variables

This step initializes all the required variables that will be used in the script. Here, the host_path is set to the path to the hosts file. In our case, it is located under **/etc**. In python, r is used to represent the raw string.

The **redirect** is assigned to the localhost address, i.e. 127.0.0.1. The **websites** is a list which contains the website lists that are to be blocked.

1. host_path = r"/etc/hosts"
2. redirect = "127.0.0.1"
3. websites = ["www.facebook.com", "https://www.facebook.com"]

Setting up the infinite loop

We need to have a while loop in the python script to make sure that our script will run at every 5 seconds.

To make this happen, we will use the sleep() method of the time module.

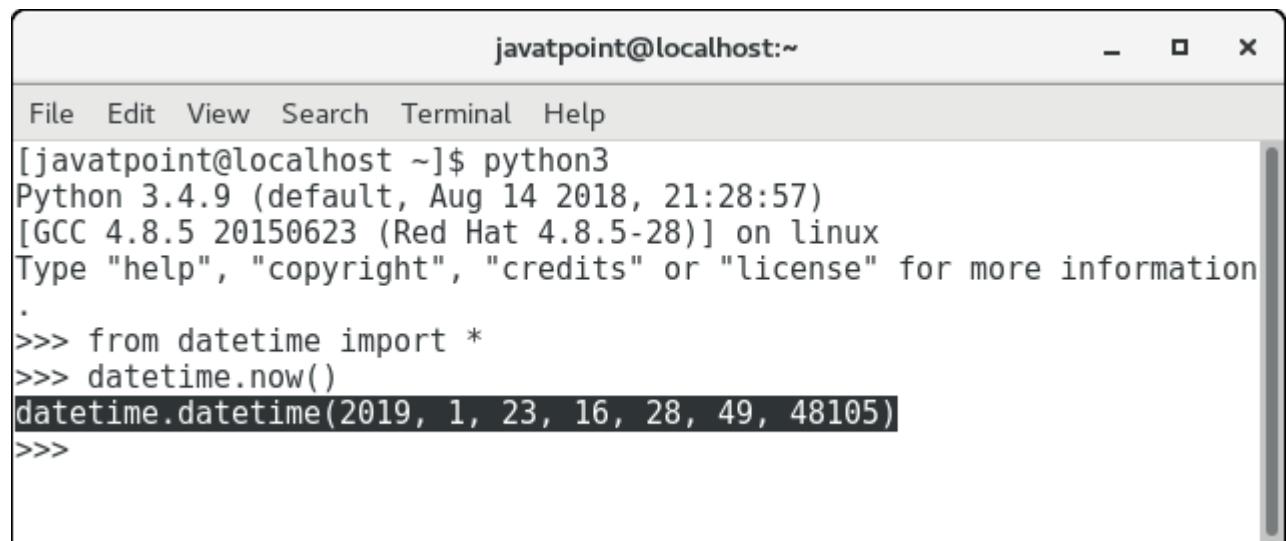
```
1. import time  
2.  
3. host_path = r"/etc/hosts"  
4. redirect = "127.0.0.1"  
5. websites = ["www.facebook.com", "https://www.facebook.com"]  
6.  
7. while True:  
8.     time.sleep(5)
```

Determining the time

In the process to build our desired python script, we need to check the current time whether it is working time or fun time since the application will block the website access during the working time.

To check the current time, we will use the datetime module. We will check whether the datetime.now() is greater than the datetime object for 9 AM of the current date and is lesser than the datetime object for 5 PM of the current date.

Let's discuss more the output of datetime.now().



```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python3  
Python 3.4.9 (default, Aug 14 2018, 21:28:57)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux  
Type "help", "copyright", "credits" or "license" for more information  
. .  
>>> from datetime import *  
>>> datetime.now()  
datetime.datetime(2019, 1, 23, 16, 28, 49, 48105)  
>>>
```

It returns a datetime object containing current time including year (2019), month(January 1st), date(23rd), time (hour, minutes, seconds). We can compare this value and check whether this value exists between the 9 AM for the current date and 5 PM for a current date using if statement.

The script will now contain the following code.

```

1. from time import *
2. from datetime import *
3.
4. host_path = r"/etc/hosts"
5. redirect = "127.0.0.1"
6. websites = ["www.facebook.com", "https://www.facebook.com"]
7.
8. while True:
9.     if datetime(datetime.now().year,datetime.now().month,datetime.now().day,9) < datetime.now() < datetime(datetime.now().year,datetime.now().month,datetime.now().day,17):
10.         print("Working hours")
11.
12.     else:
13.         print("Fun hours")
14. sleep(5)

```

Writing to the hosts file

The main objective of the script is to keep modifying the hosts file at regular intervals. To let the script configure the hosts file, we need to implement the file handling methods here.

The following code is added to the hosts file.

```

1. with open(host_path,"r+") as fileptr:
2.     content = fileptr.read()
3.     for website in websites:
4.         if website in content:
5.             pass
6.         else:
7.             fileptr.write(redirect+
8.                         "+website+"\n")

```

The open() method opens the file stored as host_path in r+ mode. First of all, we read all the content of the file by using the read() method and store it to a variable named content.

The for loop iterates over the website list (websites) and we will check for each item of the list whether it is already present in the content.

If it is present in the hosts file content, then we must pass. Otherwise, we must write the redirect-website mapping to the hosts file so that the website hostname will be redirected to the localhost.

The hosts file will now contain the following python code.

```

1. from time import *
2. from datetime import *
3. host_path = r"/etc/hosts"
4. redirect = "127.0.0.1"
5. websites = ["www.facebook.com", "https://www.facebook.com"]
6. while True:
7.     if datetime(datetime.now().year,datetime.now().month,datetime.now().day,9) < datetime.now() < datetime(datetime.now().year,datetime.now().month,datetime.now().day,17):
8.         print("working hours")
9.         with open(host_path,"r+") as fileptr:
10.             content = fileptr.read()
11.             for website in websites:
12.                 if website in content:
13.                     pass
14.                 else:
15.                     fileptr.write(redirect+" "+website+"\n")
16.     else:
17.         print("Fun hours")
18.     sleep(5)

```

Now, let's run this python script and check whether it has modified the hosts file or not.

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** PythonTest [~/PycharmProjects/PythonTest] - .../web-blocker.py [PythonTest] - PyCharm
- File Structure:** PythonTest > web-blocker.py
- Code Editor:** The code editor displays the Python script 'web-blocker.py' with syntax highlighting. The line `print("working hours")` is highlighted in yellow.
- Run History:** The bottom section shows the run history for the 'web-blocker' configuration, which runs the command `/root/PycharmProjects/PythonTest/venv/bin/python /root/PycharmProjects/PythonTest/web-blocker.py`. The output shows six consecutive lines of "working hours".

As we can see, It keeps printing working hours on the console as we are in working hours. Now, let's check the content of the hosts file.

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1      localhost localhost.localdomain localhost6 localhost6.localdomain6
#
127.0.0.1 www.facebook.com
127.0.0.1 https://www.facebook.com
~  
~  
~  
~  
~
```

As we can see, the two lines have been added to the hosts file. It will redirect the access of Facebook to the localhost.

Removing from the hosts file

Our script is working fine for the working hours, now lets add some features for the fun hours also. In the fun hours (not working hours) we must remove the added lines from the hosts file so that the access to the blocked websites will be granted.

The following code is added to the else part (fun hours case) of the script.

1. with open(host_path,'r+') as file:
2. content = file.readlines();
3. file.seek(0)
4. for line in content:
5. if not any(website in line for website in websites):
6. file.write(line)
7. file.truncate()
8. print("fun hours")

The else part will be executed during the fun hours, and it removes all the mappings that block the access to some specific websites on the computer.

Let's check the content of hosts file on the execution of the python script during the fun hours.

```
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1      localhost localhost.localdomain localhost6 localhost6.localdomain6
~  
~  
~  
~  
~  
~
```

The final script

Now, we have a python script which is running fine to block the access of some particular websites during working hours (9 AM to 5 PM) and provide the access during the fun hours.

The script **web-blocker.py** is given below.

web-blocker.py

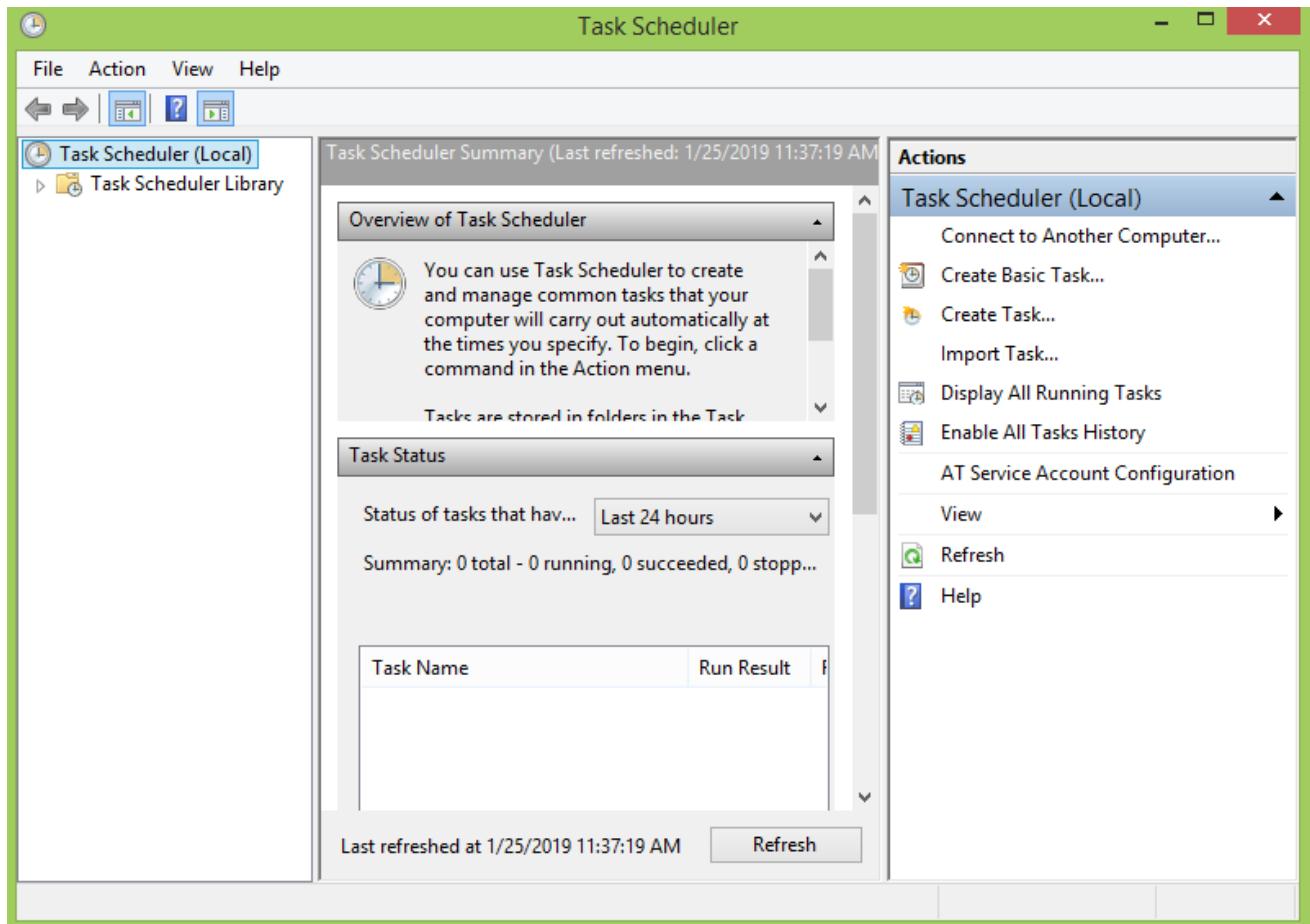
```
1. from time import *
2. from datetime import *
3.
4. host_path = r"/etc/hosts"
5. redirect = "127.0.0.1"
6. websites = ["www.facebook.com", "https://www.facebook.com"]
7.
8. while True:
9.     if datetime(datetime.now().year,datetime.now().month,datetime.now().day,9)<datetime.now()<datetime(datetime.now().year,datetime.now().month,datetime.now().day,17):
10.         with open(host_path,"r+") as fileptr:
11.             content = fileptr.read()
12.             for website in websites:
13.                 if website in content:
14.                     pass
15.                 else:
16.                     fileptr.write(redirect+"      "+website+"\n")
17.             else:
18.                 with open(host_path,'r+') as file:
19.                     content = file.readlines();
20.                     file.seek(0)
21.                     for line in content:
22.                         if not any(website in line for website in websites):
23.                             file.write(line)
24.                     file.truncate()
25.                     sleep(5)
```

Script Deployment on Windows

This section of the tutorial illustrates how the python script will be deployed at start-up so that we don't need to open the terminal all the time to run the script.

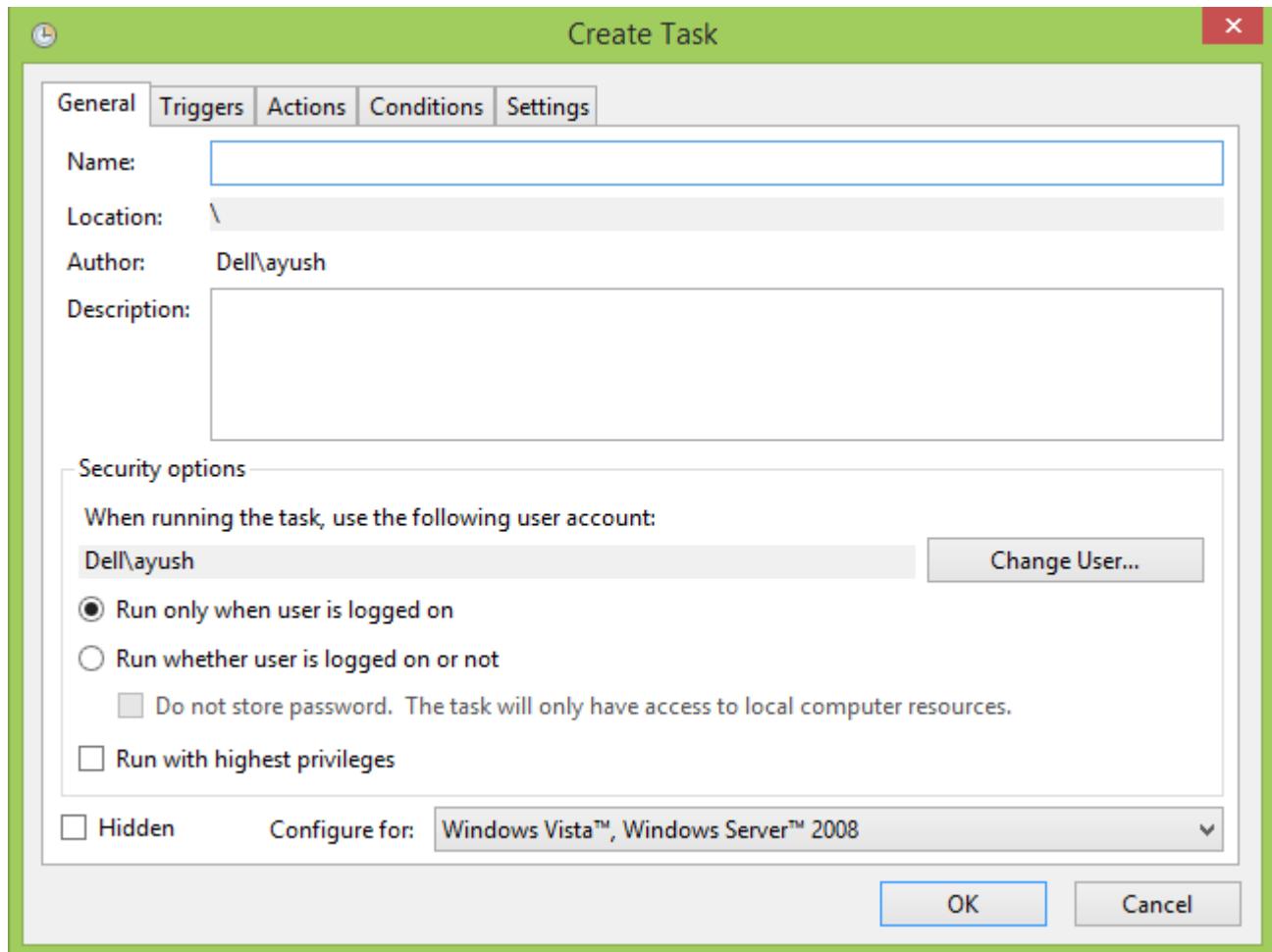
Lets first change our hosts file path from "/etc/hosts" to "C:\System32\drivers\etc\hosts" as the hosts file is stored at this location on windows.

To schedule the tasks on the windows, we need to open the task scheduler as shown in the below image.

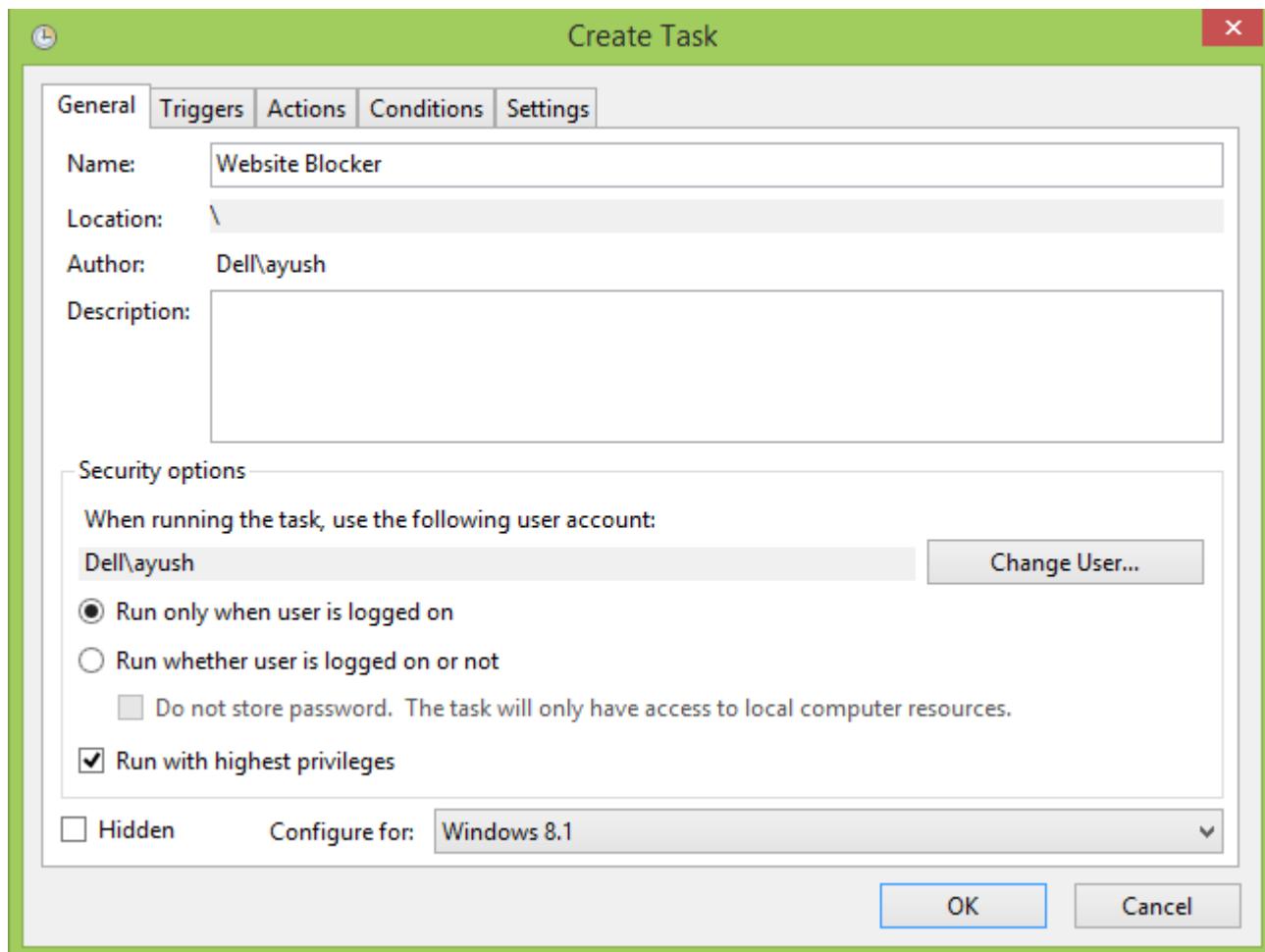


Click on **Create Task..** given in the right pane of the application.

The following window will open.

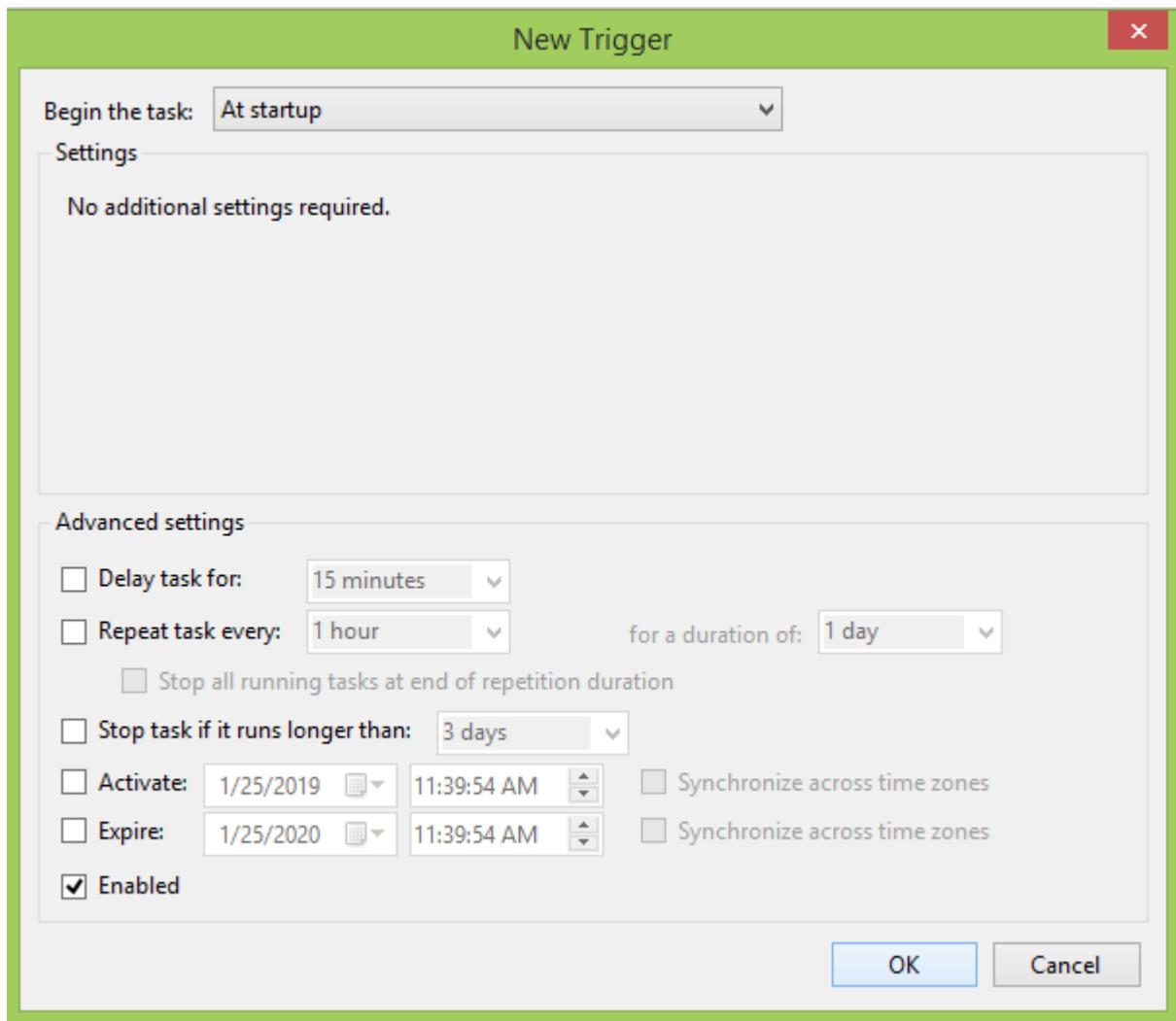


Configure the properties and give the name and other required properties for your script.
Do check the Checkbox as "**Run with highest privileges**".

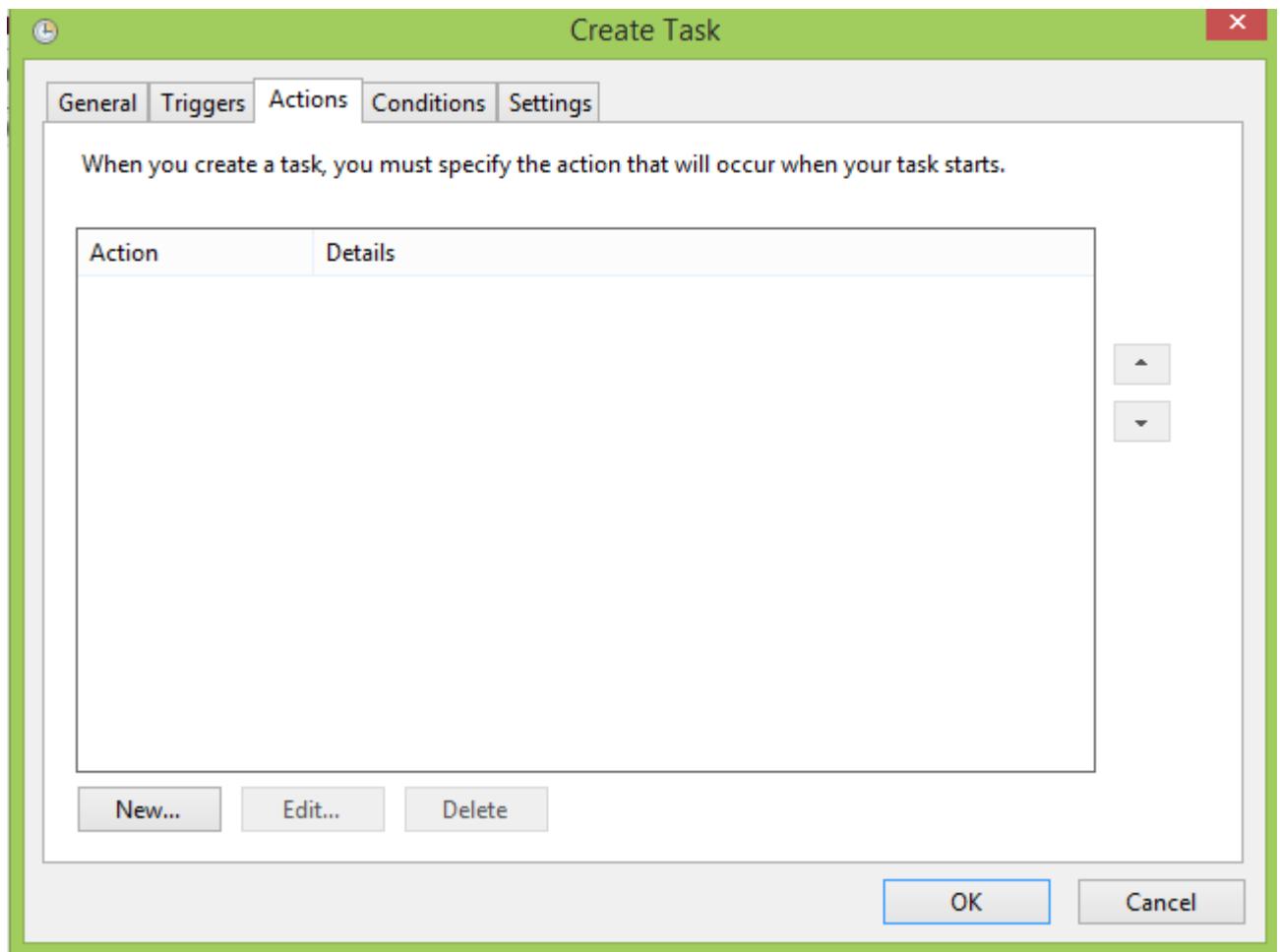


Go to Triggers and create a new trigger as shown in the below image.

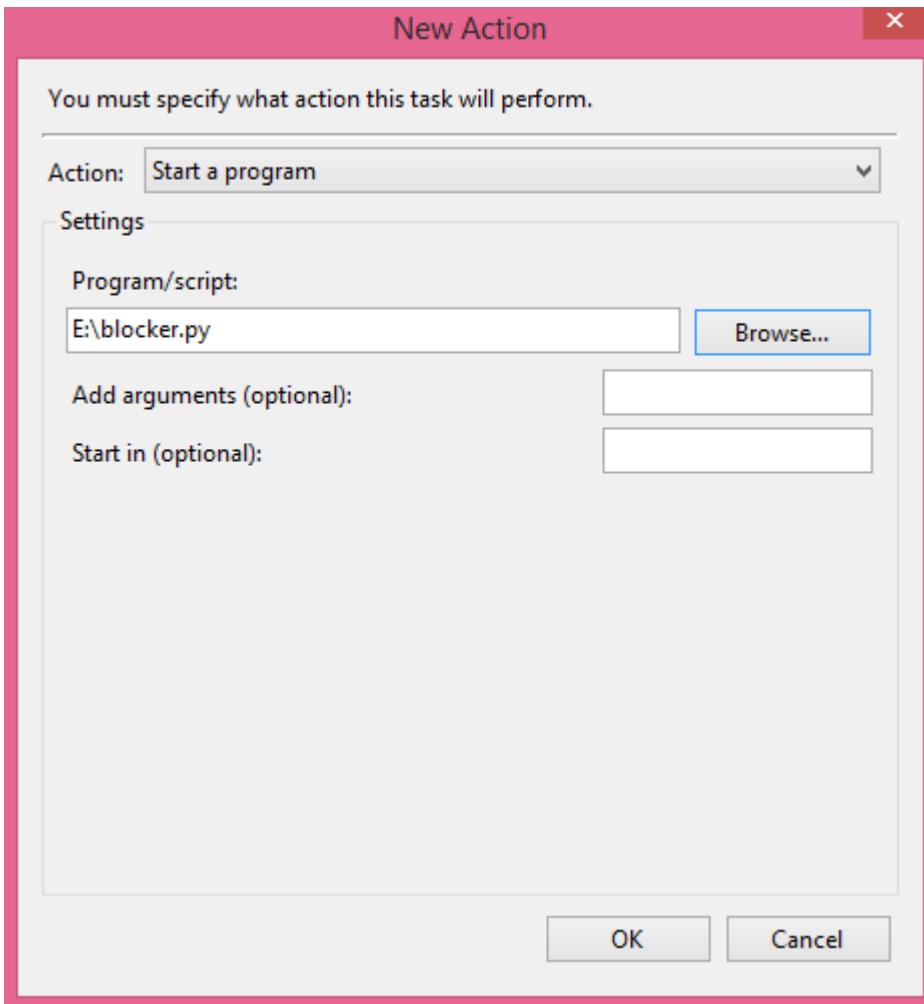
Select the "At startup" option from the drop-down list so that the script can run at start-up.



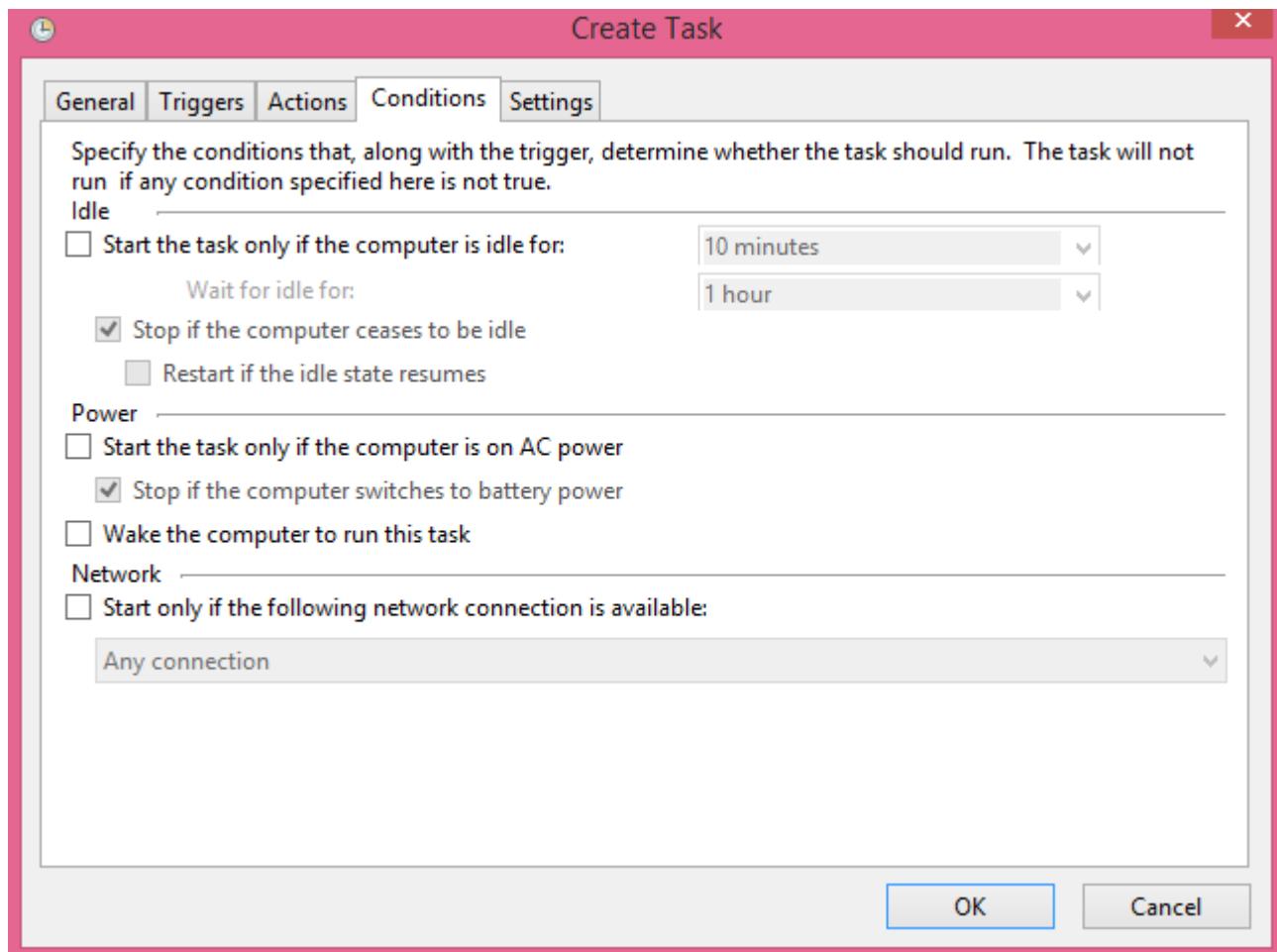
Now, go to Actions and create a new action by clicking on new.



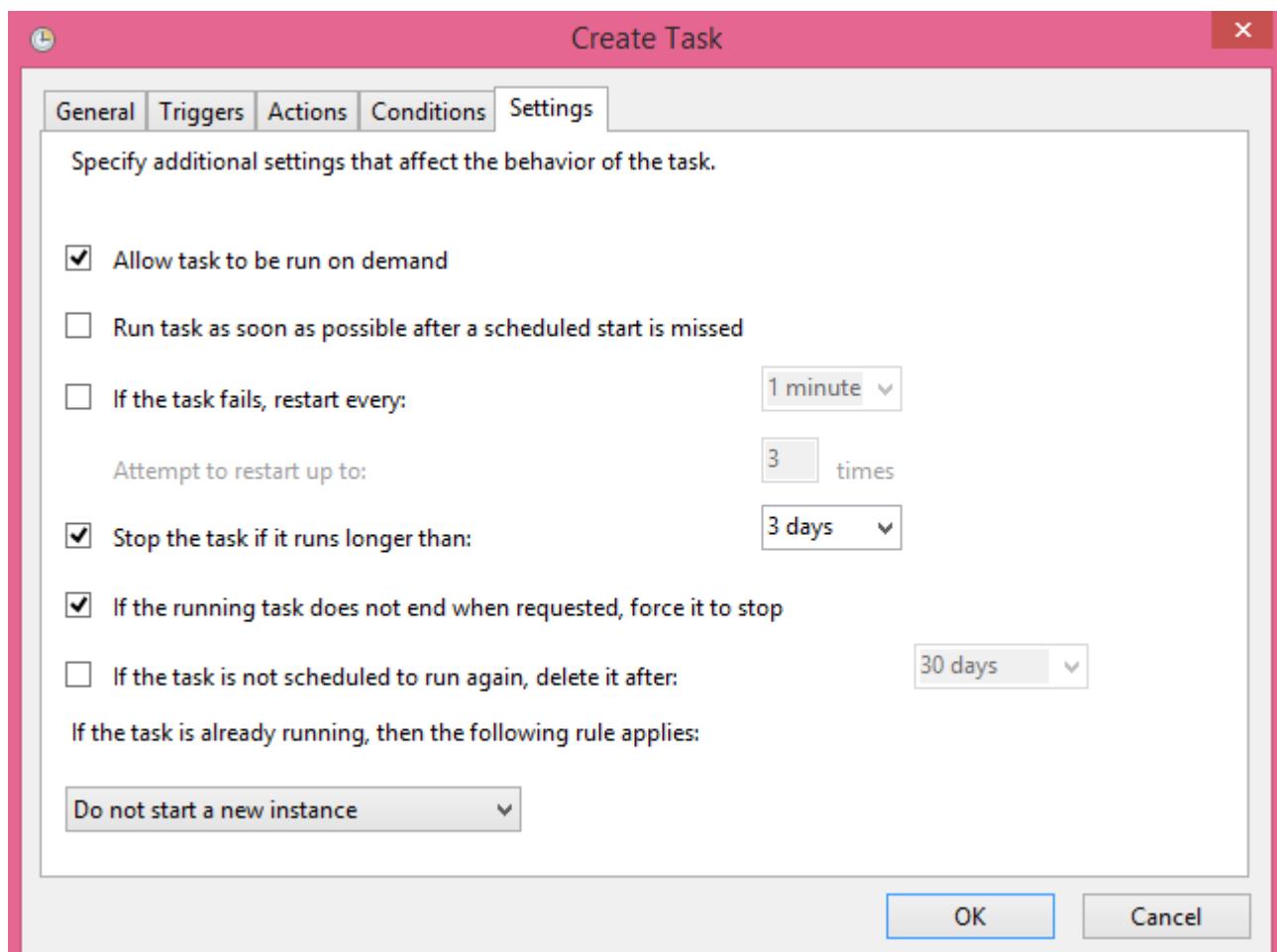
The following window will open. Select the "**Start a program**" action from the drop-down list and browse the path to the script, i.e. blocker.py (in my case it is E:\blocker.py) and click OK.



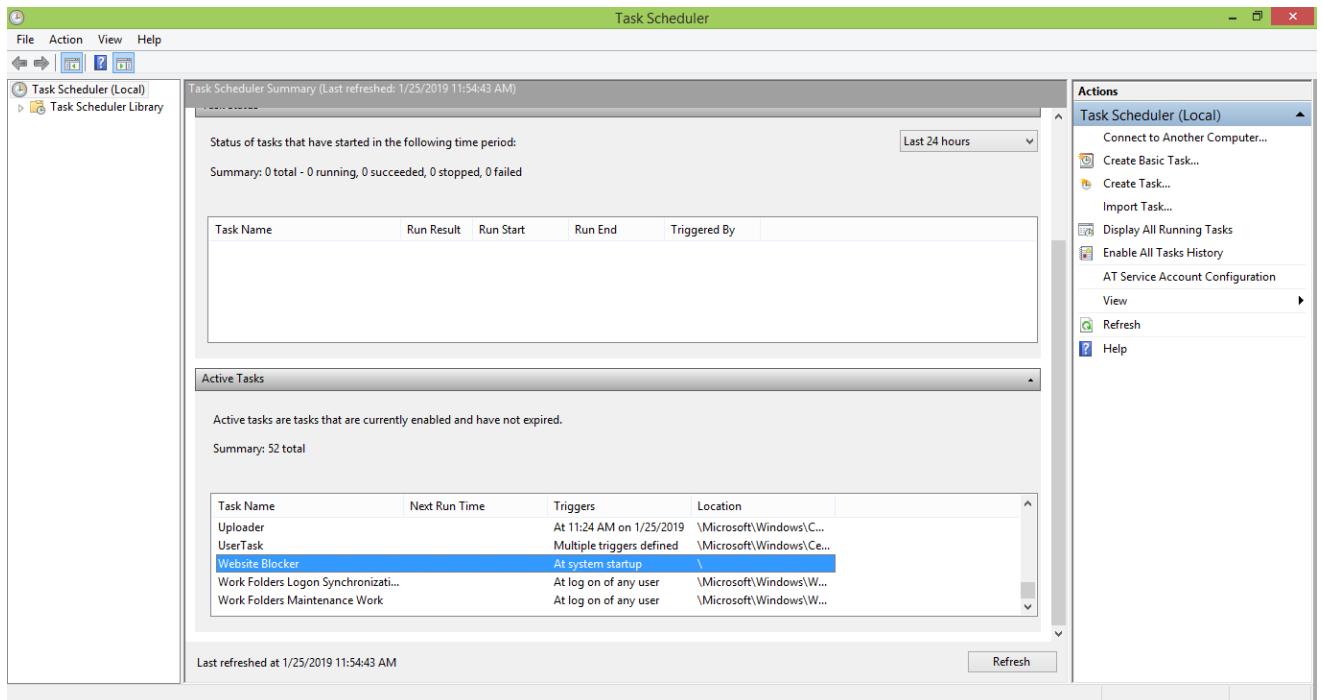
Now, click on Conditions and deselect the 2nd option which says "**Start the task only if the computer is on AC power.**"



Now, go to settings and click OK as shown in the following image.



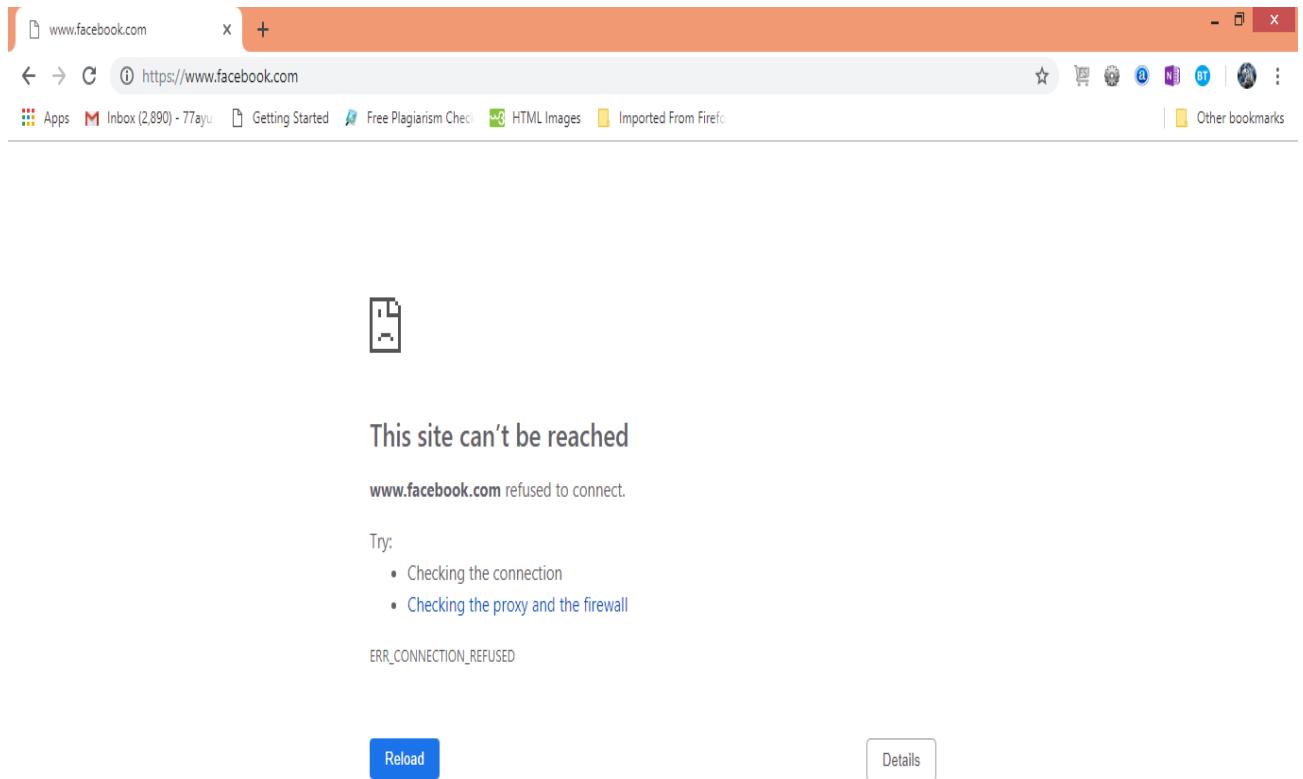
Now, we got our task Website-blocker scheduled at system start-up. We can check this in the task list as shown in the following image.



Now, we need to restart our system to make the script active on system start-up.

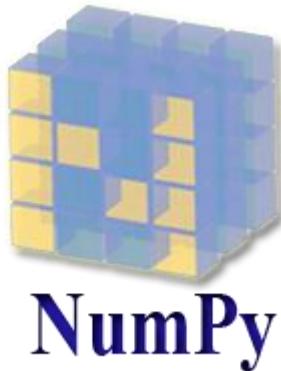
Restart the system now, and try to access the blocked website www.facebook.com as we are in working hours now.

It will show the display which looks like following.



Hence, we have got our script working fine on system start-up and block the access to www.facebook.com (or any website you want) automatically.

Python NumPy Tutorial



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

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

What is NumPy

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements.

Travis Oliphant created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.

It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.

NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.

In this tutorial, we will go through the numeric python library NumPy.

The need of NumPy

With the revolution of data science, data analysis libraries like NumPy, SciPy, Pandas, etc. have seen a lot of growth. With a much easier syntax than other programming languages, python is the first choice language for the data scientist.

NumPy provides a convenient and efficient way to handle the vast amount of data. NumPy is also very convenient with Matrix multiplication and data reshaping. NumPy is fast which makes it reasonable to work with a large set of data.

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
5. NumPy provides the in-built functions for linear algebra and random number generation.

Nowadays, NumPy in combination with SciPy and Matplotlib is used as the replacement to MATLAB as Python is more complete and easier programming language than MATLAB.

Prerequisite

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

Audience

Our Numpy tutorial is designed to help beginners and professionals.

Problem

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

NumPy Environment Setup

NumPy doesn't come bundled with Python. We have to install it using **the python pip** installer. Execute the following command.

1. \$ pip install numpy

It is best practice to install NumPy with the full SciPy stack. The binary distribution of the SciPy stack is specific to the operating systems.

Windows

On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.

It can be downloaded from the official website: <https://www.anaconda.com/>. It is also available for Linux and Mac.

The CanoPy also comes with the full SciPy stack which is available as free as well as commercial license. We can download it by visiting the link: <https://www.enthought.com/products/canopy/>

The Python (x, y) is also available free with the full SciPy distribution. Download it by visiting the link: <https://python-xy.github.io/>

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.

Ubuntu

Execute the following command on the terminal.

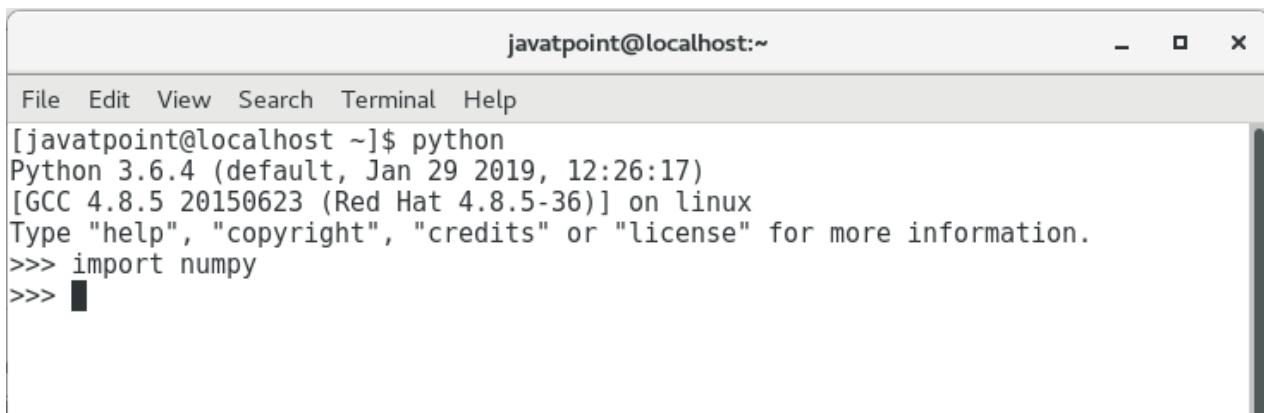
1. \$ sudo apt-get install python-numpy
- 2.
3. \$ python-scipy python-matplotlib python-ipython notebook python-pandas
- 4.
5. \$ python-sympy python-nose

Redhat

On Redhat, the following commands are executed to install the Python SciPy package stack.

1. \$ sudo yum install numpyscipy python-matplotlibpython
- 2.
3. \$ python-pandas sympy python-nose atlas-devel

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy as shown in the below image. If it doesn't give the error, then it is installed successfully.



```
javatpoint@localhost:~
```

```
File Edit View Search Terminal Help
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> 
```

NumPy Ndarray

Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define a ndarray as the collection of the data type (dtype) objects.

The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

1. >>> a = numpy.array

Consider the below image.

A screenshot of a terminal window titled "javatpoint@localhost:~". The window shows the following Python session:

```
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a = np.array
>>> print(a)
<built-in function array>
>>>
```

We can also pass a collection object into the array routine to create the equivalent n-dimensional array. The syntax is given below.

1. >>> numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

The parameters are described in the following table.

| SN | Parameter | Description |
|----|-----------|--|
| 1 | object | It represents the collection object. It can be a list, tuple, dictionary, set, etc. |
| 2 | dtype | We can change the data type of the array elements by changing this option to the specified type. The default is none. |
| 3 | copy | It is optional. By default, it is true which means the object is copied. |
| 4 | order | There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any) |
| 5 | subok | The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true. |

| | | |
|---|-------|--|
| 6 | ndmin | It represents the minimum dimensions of the resultant array. |
|---|-------|--|

To create an array using the list, use the following syntax.

1. `>>> a = numpy.array([1, 2, 3])`

```
javatpoint@localhost:~ - □
File Edit View Search Terminal Help
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([1,2,3])
>>> print(a)
[1 2 3]
>>> print(type(a))
<class 'numpy.ndarray'>
>>> █
```

To create a multi-dimensional array object, use the following syntax.

1. `>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])`

```
javatpoint@localhost:~ - □
File Edit View Search Terminal Help
[javatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> a = numpy.array([[1,2,3],[4,5,6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>> █
```

To change the data type of the array elements, mention the name of the data type along with the collection.

1. `>>> a = numpy.array([1, 3, 5, 7], complex)`

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy  
>>> a = numpy.array([1, 3, 5, 7], dtype = complex)  
>>> print(a)  
[1.+0.j 3.+0.j 5.+0.j 7.+0.j]  
>>> █
```

Finding the dimensions of the Array

The **ndim** function can be used to find the dimensions of the array.

1. >>> **import** numpy as np
2. >>> arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])
- 3.
4. >>> **print**(arr.ndim)

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy as np  
>>> arr = np.array([[1,2,3,4],[4,5,6,7],[9,10,11,23]])  
>>> print(arr.ndim)  
2  
>>> print(arr)  
[[ 1  2  3  4]  
 [ 4  5  6  7]  
 [ 9 10 11 23]]  
>>> █
```

Finding the size of each array element

The **itemsize** function is used to get the size of each array item. It returns the number of bytes taken by each array element.

Consider the following example.

Example

1. #finding the size of each item in the array
2. **import** numpy as np
3. a = np.array([[1,2,3]])

```
4. print("Each item contains",a.itemsize,"bytes")
```

Output:

```
Each item contains 8 bytes.
```

Finding the data type of each array item

To check the data type of each array item, the `dtype` function is used. Consider the following example to check the data type of the array items.

Example

```
1. #finding the data type of each array item
2. import numpy as np
3. a = np.array([[1,2,3]])
4. print("Each item is of the type",a.dtype)
```

Output:

```
Each item is of the type int64
```

Finding the shape and size of the array

To get the shape and size of the array, the `size` and `shape` function associated with the numpy array is used.

Consider the following example.

Example

```
1. import numpy as np
2. a = np.array([[1,2,3,4,5,6,7]])
3. print("Array Size:",a.size)
4. print("Shape:",a.shape)
```

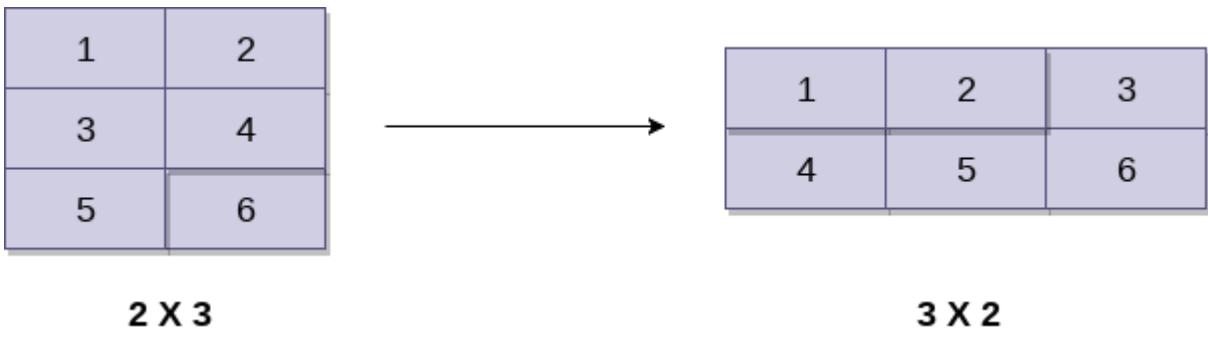
Output:

```
Array Size: 7
Shape: (1, 7)
```

Reshaping the array objects

By the shape of the array, we mean the number of rows and columns of a multi-dimensional array. However, the numpy module provides us the way to reshape the array by changing the number of rows and columns of the multi-dimensional array.

The `reshape()` function associated with the `ndarray` object is used to reshape the array. It accepts the two parameters indicating the row and columns of the new shape of the array.



Example

1. **import** numpy as np
2. a = np.array([[1,2],[3,4],[5,6]])
3. **print**("printing the original array..")
4. **print**(a)
5. a=a.reshape(2,3)
6. **print**("printing the reshaped array..")
7. **print**(a)

Output:

```
printing the original array..
[[1 2]
 [3 4]
 [5 6]]
printing the reshaped array..
[[1 2 3]
 [4 5 6]]
```

Slicing in the Array

Slicing in the NumPy array is the way to extract a range of elements from an array. Slicing in the array is performed in the same way as it is performed in the python list.

Consider the following example to print a particular element of the array.

Example

1. **import** numpy as np
2. a = np.array([[1,2],[3,4],[5,6]])
3. **print**(a[0,1])
4. **print**(a[2,0])

Output:

```
2
5
```

The above program prints the 2nd element from the 0th index and 0th element from the 2nd index of the array.

Linspace

The linspace() function returns the evenly spaced values over the given interval. The following example returns the 10 evenly separated values over the given interval 5-15

Example

1. `import numpy as np`
2. `a=np.linspace(5,15,10) #prints 10 values which are evenly spaced over the given interval 5-15`
3. `print(a)`

Output:

```
[ 5.           6.11111111  7.22222222  8.33333333  9.44444444 10.55555556
 11.66666667 12.77777778 13.88888889 15.         ]
```

Finding the maximum, minimum, and sum of the array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum, minimum, and sum of the array elements respectively.

Consider the following example.

Example

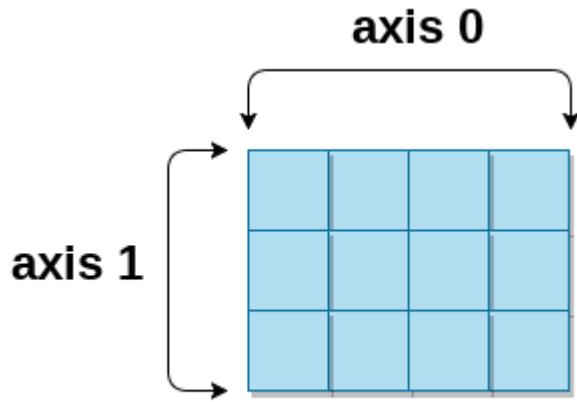
1. `import numpy as np`
2. `a = np.array([1,2,3,10,15,4])`
3. `print("The array:",a)`
4. `print("The maximum element:",a.max())`
5. `print("The minimum element:",a.min())`
6. `print("The sum of the elements:",a.sum())`

Output:

```
The array: [ 1  2  3 10 15  4]
The maximum element: 15
The minimum element: 1
The sum of the elements: 35
```

NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where axis-0 represents the columns and axis-1 represents the rows. We can mention the axis to perform row-level or column-level calculations like the addition of row or column elements.



NumPy Array

To calculate the maximum element among each column, the minimum element among each row, and the addition of all the row elements, consider the following example.

Example

1. `import numpy as np`
2. `a = np.array([[1,2,30],[10,15,4]])`
3. `print("The array:",a)`
4. `print("The maximum elements of columns:",a.max(axis = 0))`
5. `print("The minimum element of rows:",a.min(axis = 1))`
6. `print("The sum of all rows",a.sum(axis = 1))`

Output:

```
The array: [[1  2  30]
           [10 15  4]]
The maximum elements of columns: [10 15 30]
The minimum element of rows [1 4]
The sum of all rows [33 29]
```

Finding square root and standard deviation

The `sqrt()` and `std()` functions associated with the numpy array are used to find the square root and standard deviation of the array elements respectively.

Standard deviation means how much each element of the array varies from the mean value of the numpy array.

Consider the following example.

Example

1. `import numpy as np`
2. `a = np.array([[1,2,30],[10,15,4]])`
3. `print(np.sqrt(a))`
4. `print(np.std(a))`

Output:

```
[[1.          1.41421356 5.47722558]
 [3.16227766 3.87298335 2.          ]
 10.044346115546242
```

Arithmetic operations on the array

The numpy module allows us to perform the arithmetic operations on multi-dimensional arrays directly.

In the following example, the arithmetic operations are performed on the two multi-dimensional arrays a and b.

Example

1. `import` numpy as np
2. `a = np.array([[1,2,30],[10,15,4]])`
3. `b = np.array([[1,2,3],[12, 19, 29]])`
4. `print("Sum of array a and b\n",a+b)`
5. `print("Product of array a and b\n",a*b)`
6. `print("Division of array a and b\n",a/b)`

Array Concatenation

The numpy provides us with the vertical stacking and horizontal stacking which allows us to concatenate two multi-dimensional arrays vertically or horizontally.

Consider the following example.

Example

1. `import` numpy as np
2. `a = np.array([[1,2,30],[10,15,4]])`
3. `b = np.array([[1,2,3],[12, 19, 29]])`
4. `print("Arrays vertically concatenated\n",np.vstack((a,b)))`
5. `print("Arrays horizontally concatenated\n",np.hstack((a,b)))`

Output:

```
Arrays vertically concatenated
 [[ 1  2 30]
 [10 15  4]
 [ 1  2  3]
 [12 19 29]]
Arrays horizontally concatenated
 [[ 1  2 30  1  2  3]
 [10 15  4 12 19 29]]
```

NumPy Datatypes

The NumPy provides a higher range of numeric data types than that provided by the Python. A list of numeric data types is given in the following table.

| SN | Data type | Description |
|----|-----------|--|
| 1 | bool_ | It represents the boolean value indicating true or false. It is stored as a byte. |
| 2 | int_ | It is the default type of integer. It is identical to long type in C that contains 64 bit or 32-bit integer. |
| 3 | intc | It is similar to the C integer (c int) as it represents 32 or 64-bit int. |
| 4 | intp | It represents the integers which are used for indexing. |
| 5 | int8 | It is the 8-bit integer identical to a byte. The range of the value is -128 to 127. |
| 6 | int16 | It is the 2-byte (16-bit) integer. The range is -32768 to 32767. |
| 7 | int32 | It is the 4-byte (32-bit) integer. The range is -2147483648 to 2147483647. |
| 8 | int64 | It is the 8-byte (64-bit) integer. The range is -9223372036854775808 to 9223372036854775807. |
| 9 | uint8 | It is the 1-byte (8-bit) unsigned integer. |
| 10 | uint16 | It is the 2-byte (16-bit) unsigned integer. |
| 11 | uint32 | It is the 4-byte (32-bit) unsigned integer. |
| 12 | uint64 | It is the 8 bytes (64-bit) unsigned integer. |
| 13 | float_ | It is identical to float64. |
| 14 | float16 | It is the half-precision float. 5 bits are reserved for the exponent. 10 bits are reserved for mantissa, and 1 bit is reserved for the sign. |
| 15 | float32 | It is a single precision float. 8 bits are reserved for the exponent, 23 bits are reserved for mantissa, and 1 bit is reserved for the sign. |

| | | |
|----|------------|---|
| 16 | float64 | It is the double precision float. 11 bits are reserved for the exponent, 52 bits are reserved for mantissa, 1 bit is used for the sign. |
| 17 | complex_ | It is identical to complex128. |
| 18 | complex64 | It is used to represent the complex number where real and imaginary part shares 32 bits each. |
| 19 | complex128 | It is used to represent the complex number where real and imaginary part shares 64 bits each. |

NumPy dtype

All the items of a numpy array are data type objects also known as numpy dtypes. A data type object implements the fixed size of memory corresponding to an array.

We can create a dtype object by using the following syntax.

1. `numpy.dtype(object, align, copy)`

The constructor accepts the following object.

Object: It represents the object which is to be converted to the data type.

Align: It can be set to any boolean value. If true, then it adds extra padding to make it equivalent to a C struct.

Copy: It creates another copy of the dtype object.

Example 1

1. `import numpy as np`
2. `d = np.dtype(np.int32)`
3. `print(d)`

Output:

```
int32
```

Example 2

1. `import numpy as np`
2. `d = np.int32(i4)`
3. `print(d)`

Output:

```
int32
```

Creating a Structured data type

We can create a map-like (dictionary) data type which contains the mapping between the values. For example, it can contain the mapping between employees and salaries or the students and the age, etc.

Consider the following example.

Example 1

1. `import numpy as np`
2. `d = np.dtype([('salary',np.float)])`
3. `print(d)`

Output:

```
[('salary', '<=' pre="">
```

Example 2

1. `import numpy as np`
2. `d=np.dtype([('salary',np.float)])`
3. `arr = np.array([(10000.12),(20000.50)],dtype=d)`
4. `print(arr['salary'])`

Output:

```
[(10000.12,) (20000.5 ,)]
```

Numpy Array Creation

The ndarray object can be constructed by using the following routines.

Numpy.empty

As the name specifies, The empty routine is used to create an uninitialized array of specified shape and data type.

The syntax is given below.

1. `numpy.empty(shape, dtype = float, order = 'C')`

It accepts the following parameters.

- **Shape:** The desired shape of the specified array.
- **dtype:** The data type of the array items. The default is the float.
- **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

1. `import numpy as np`
2. `arr = np.empty((3,2), dtype = int)`
3. `print(arr)`

Output:

```
[ [ 140482883954664      36917984]
  [ 140482883954648      140482883954648]
  [6497921830368665435  172026472699604272] ]
```

NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 0.

The syntax is given below.

1. `numpy.zeros(shape, dtype = float, order = 'C')`

It accepts the following parameters.

- o **Shape:** The desired shape of the specified array.
- o **dtype:** The data type of the array items. The default is the float.
- o **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

1. `import numpy as np`
2. `arr = np.zeros((3,2), dtype = int)`
3. `print(arr)`

Output:

```
[ [0 0]
  [0 0]
  [0 0]]
```

NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array item is initialized to 1.

The syntax to use this module is given below.

1. `numpy.ones(shape, dtype = none, order = 'C')`

It accepts the following parameters.

- **Shape:** The desired shape of the specified array.
- **dtype:** The data type of the array items.
- **Order:** The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

1. `import numpy as np`
2. `arr = np.ones((3,2), dtype = int)`
3. `print(arr)`

Output:

```
[[1 1]
 [1 1]
 [1 1]]
```

Numpy array from existing data

NumPy provides us the way to create an array by using the existing data.

`numpy.asarray`

This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

The syntax to use the `asarray()` routine is given below.

1. `numpy.asarray(sequence, dtype = None, order = None)`

It accepts the following parameters.

1. **sequence:** It is the python sequence which is to be converted into the python array.
2. **dtype:** It is the data type of each item of the array.
3. **order:** It can be set to C or F. The default is C.

Example: creating numpy array using the list

1. `import numpy as np`
2. `l=[1,2,3,4,5,6,7]`
3. `a = np.asarray(l);`
4. `print(type(a))`
5. `print(a)`

Output:

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Example: creating a numpy array using Tuple

1. `import` numpy as np
2. `l=(1,2,3,4,5,6,7)`
3. `a = np.asarray(l);`
4. `print(type(a))`
5. `print(a)`

Output:

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Example: creating a numpy array using more than one list

1. `import` numpy as np
2. `l=[[1,2,3,4,5,6,7],[8,9]]`
3. `a = np.asarray(l);`
4. `print(type(a))`
5. `print(a)`

Output:

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
```

numpy.frombuffer

This function is used to create an array by using the specified buffer. The syntax to use this buffer is given below.

1. `numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)`

It accepts the following parameters.

- **buffer:** It represents an object that exposes a buffer interface.
- **dtype:** It represents the data type of the returned data type array. The default value is 0.
- **count:** It represents the length of the returned ndarray. The default value is -1.
- **offset:** It represents the starting position to read from. The default value is 0.

Example

1. `import` numpy as np
2. `l = b'hello world'`
3. `print(type(l))`
4. `a = np.frombuffer(l, dtype = "S1")`
5. `print(a)`
6. `print(type(a))`

Output:

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

numpy.fromiter

This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarray object.

The syntax is given below.

1. `numpy.fromiter(iterable, dtype, count = -1)`

It accepts the following parameters.

1. **Iterable:** It represents an iterable object.
2. **dtype:** It represents the data type of the resultant array items.
3. **count:** It represents the number of items to read from the buffer in the array.

Example

1. `import numpy as np`
2. `list = [0,2,4,6]`
3. `it = iter(list)`
4. `x = np.fromiter(it, dtype = float)`
5. `print(x)`
6. `print(type(x))`

Output:

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

Numpy Arrays within the numerical range

This section of the tutorial illustrates how the numpy arrays can be created using some given specified range.

Numpy.arange

It creates an array by using the evenly spaced values over the given interval. The syntax to use the function is given below.

1. `numpy.arange(start, stop, step, dtype)`

It accepts the following parameters.

1. **start:** The starting of an interval. The default is 0.
2. **stop:** represents the value at which the interval ends excluding this value.
3. **step:** The number by which the interval values change.
4. **dtype:** the data type of the numpy array items.

group

Example

1. **import** numpy as np
2. arr = np.arange(0,10,2,float)
3. **print**(arr)

Output:

```
[0.  2.  4.  6.  8.]
```

Example

1. **import** numpy as np
2. arr = np.arange(10,100,5,int)
3. **print**("The array over the given range is ",arr)

Output:

```
The array over the given range is [10 15 20 25 30 35 40 45 50 55 60 65 70 75  
80 85 90 95]
```

NumPy.linspace

It is similar to the arrange function. However, it doesn't allow us to specify the step size in the syntax.

Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size.

The syntax is given below.

1. numpy.linspace(start, stop, num, endpoint, retstep, dtype)

It accepts the following parameters.

1. **start:** It represents the starting value of the interval.
2. **stop:** It represents the stopping value of the interval.
3. **num:** The amount of evenly spaced samples over the interval to be generated. The default is 50.
4. **endpoint:** Its true value indicates that the stopping value is included in the interval.

5. **retstep:** This has to be a boolean value. Represents the steps and samples between the consecutive numbers.
6. **dtype:** It represents the data type of the array items.

Example

1. **import** numpy as np
2. arr = np.linspace(10, 20, 5)
3. **print**("The array over the given range is ",arr)

Output:

```
The array over the given range is [10. 12.5 15. 17.5 20.]
```

Example

1. **import** numpy as np
2. arr = np.linspace(10, 20, 5, endpoint = False)
3. **print**("The array over the given range is ",arr)

Output:

```
The array over the given range is [10. 12. 14. 16. 18.]
```

numpy.logspace

It creates an array by using the numbers that are evenly separated on a log scale.

The syntax is given below.

1. numpy.logspace(start, stop, num, endpoint, base, dtype)

It accepts the following parameters.

1. **start:** It represents the starting value of the interval in the base.
2. **stop:** It represents the stopping value of the interval in the base.
3. **num:** The number of values between the range.
4. **endpoint:** It is a boolean type value. It makes the value represented by stop as the last value of the interval.
5. **base:** It represents the base of the log space.
6. **dtype:** It represents the data type of the array items.

Example

1. **import** numpy as np
2. arr = np.logspace(10, 20, num = 5, endpoint = True)
3. **print**("The array over the given range is ",arr)

Output:

```
The array over the given range is [1.0000000e+10 3.16227766e+12 1.00000000e+15  
3.16227766e+17  
1.00000000e+20]
```

Example

1. **import** numpy as np
2. arr = np.logspace(10, 20, num = 5, base = 2, endpoint = True)
3. **print**("The array over the given range is ",arr)

Output:

```
The array over the given range is [1.0240000e+03 5.79261875e+03  
3.27680000e+04 1.85363800e+05  
1.04857600e+06]
```

NumPy Broadcasting

In Mathematical operations, we may need to consider the arrays of different shapes. NumPy can perform such operations where the array of different shapes are involved.

For example, if we consider the matrix multiplication operation, if the shape of the two matrices is the same then this operation will be easily performed. However, we may also need to operate if the shape is not similar.

Consider the following example to multiply two arrays.

Example

1. **import** numpy as np
2. a = np.array([1,2,3,4,5,6,7])
3. b = np.array([2,4,6,8,10,12,14])
4. c = a*b;
5. **print**(c)

Output:

```
[ 2   8  18  32  50  72  98]
```

However, in the above example, if we consider arrays of different shapes, we will get the errors as shown below.

Example

1. **import** numpy as np
2. a = np.array([1,2,3,4,5,6,7])
3. b = np.array([2,4,6,8,10,12,14,19])
4. c = a*b;
5. **print**(c)

Output:

```
ValueError: operands could not be broadcast together with shapes (7,) (8,)
```

In the above example, we can see that the shapes of the two arrays are not similar and therefore they cannot be multiplied together. NumPy can perform such operation by using the concept of broadcasting.

In broadcasting, the smaller array is broadcast to the larger array to make their shapes compatible with each other.

Broadcasting Rules

Broadcasting is possible if the following cases are satisfied.

1. The smaller dimension array can be appended with '1' in its shape.
2. Size of each output dimension is the maximum of the input sizes in the dimension.
3. An input can be used in the calculation if its size in a particular dimension matches the output size or its value is exactly 1.
4. If the input size is 1, then the first data entry is used for the calculation along the dimension.

Broadcasting can be applied to the arrays if the following rules are satisfied.

1. All the input arrays have the same shape.
2. Arrays have the same number of dimensions, and the length of each dimension is either a common length or 1.
3. Array with the fewer dimension can be appended with '1' in its shape.

Let's see an example of broadcasting.

Example

1. **import** numpy as np
2. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
3. b = np.array([2,4,6,8])
4. **print**("\nprinting array a..")
5. **print**(a)
6. **print**("\nprinting array b..")
7. **print**(b)
8. **print**("\nAdding arrays a and b ..")
9. c = a + b;
10. **print**(c)

Output:

```
printing array a..
```

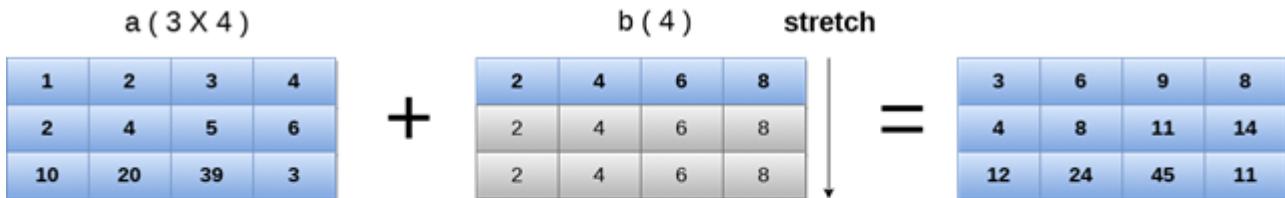
```

[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]

printing array b..
[2 4 6 8]

Adding arrays a and b ..
[[ 3  6  9 12]
 [ 4  8 11 14]
 [12 24 45 11]]

```



```

a = [[1, 2, 3, 4], [2, 4, 5, 6], [10, 20, 39, 3]]
b = [[2, 4, 6, 8]]

```

NumPy Array Iteration

NumPy provides an iterator object, i.e., `nditer` which can be used to iterate over the given array using python standard Iterator interface.

Consider the following example.

Example

1. `import numpy as np`
2. `a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])`
3. `print("Printing array:")`
4. `print(a);`
5. `print("Iterating over the array:")`
6. `for x in np.nditer(a):`
7. `print(x,end=' ')`

Output:

```

Printing array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
Iterating over the array:
1 2 3 4 2 4 5 6 10 20 39 3

```

Order of the iteration doesn't follow any special ordering like row-major or column-order. However, it is intended to match the memory layout of the array.

Let's iterate over the transpose of the array given in the above example.

Example

```
1. import numpy as np
2. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
3. print("Printing the array:")
4. print(a)
5. print("Printing the transpose of the array:")
6. at = a.T
7. print(at)
8.
9. #this will be same as previous
10. for x in np.nditer(at):
11.     print(print("Iterating over the array:"))
12. for x in np.nditer(a):
13.     print(x,end=' ')
```

Output:

```
Printing the array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]
Printing the transpose of the array:
[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]
1 2 3 4 2 4 5 6 10 20 39 3
```

Order of Iteration

As we know, there are two ways of storing values into the numpy arrays:

1. F-style order
2. C-style order

Let's see an example of how the numpy Iterator treats the specific orders (F or C).

Example

```
1. import numpy as np
2.
3. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
4.
5. print("\nPrinting the array:\n")
6.
7. print(a)
8.
```

```

9. print("\nPrinting the transpose of the array:\n")
10. at = a.T
11.
12. print(at)
13.
14. print("\nIterating over the transposed array\n")
15.
16. for x in np.nditer(at):
17.     print(x, end= ' ')
18.
19. print("\nSorting the transposed array in C-style:\n")
20.
21. c = at.copy(order = 'C')
22.
23. print(c)
24.
25. print("\nIterating over the C-style array:\n")
26. for x in np.nditer(c):
27.     print(x,end=' ')
28.
29.
30. d = at.copy(order = 'F')
31.
32. print(d)
33. print("Iterating over the F-style array:\n")
34. for x in np.nditer(d):
35.     print(x,end=' ')

```

Output:

```

Printing the array:
[[ 1  2  3  4]
 [ 2  4  5  6]
 [10 20 39  3]]

Printing the transpose of the array:
[[ 1  2 10]
 [ 2  4 20]
 [ 3  5 39]
 [ 4  6  3]]

Iterating over the transposed array
1 2 3 4 2 4 5 6 10 20 39 3
Sorting the transposed array in C-style:
[[ 1  2 10]]

```

```
[ 2  4 20]
[ 3  5 39]
[ 4  6  3]]
```

Iterating over the C-style array:

```
1 2 10 2 4 20 3 5 39 4 6 3 [[ 1  2 10]
[ 2  4 20]
[ 3  5 39]
[ 4  6  3]]
```

Iterating over the F-style array:

```
1 2 3 4 2 4 5 6 10 20 39 3
```

We can mention the order 'C' or 'F' while defining the Iterator object itself. Consider the following example.

Example

```
1. import numpy as np
2.
3. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
4.
5. print("\nPrinting the array:\n")
6.
7. print(a)
8.
9. print("\nPrinting the transpose of the array:\n")
10. at = a.T
11.
12. print(at)
13.
14. print("\nIterating over the transposed array\n")
15.
16. for x in np.nditer(at):
17.     print(x, end= ' ')
18.
19. print("\nSorting the transposed array in C-style:\n")
20.
21. print("\nIterating over the C-style array:\n")
22. for x in np.nditer(at, order = 'C'):
23.     print(x,end=' ')
```

Output:

```
Iterating over the transposed array
1 2 3 4 2 4 5 6 10 20 39 3
Sorting the transposed array in C-style:
```

```
Iterating over the C-style array:
```

```
1 2 10 2 4 20 3 5 39 4 6 3
```

Array Values Modification

We can not modify the array elements during the iteration since the op-flag associated with the Iterator object is set to readonly.

However, we can set this flag to readwrite or write only to modify the array values. Consider the following example.

Example

```
1. import numpy as np
2.
3. a = np.array([[1,2,3,4],[2,4,5,6],[10,20,39,3]])
4.
5. print("\nPrinting the original array:\n")
6.
7. print(a)
8.
9. print("\nIterating over the modified array\n")
10.
11. for x in np.nditer(a, op_flags = ['readwrite']):
12.     x[...] = 3 * x;
13.     print(x,end = ' ')
```

Output:

```
Printing the original array:
```

```
[[ 1   2   3   4]
 [ 2   4   5   6]
 [10  20  39  3]]
```

```
Iterating over the modified array
```

```
3 6 9 12 6 12 15 18 30 60 117 9
```

NumPy Bitwise Operators

Numpy provides the following bitwise operators.

| SN | Operator | Description |
|----|-------------|---|
| 1 | bitwise_and | It is used to calculate the bitwise and operation between the corresponding array elements. |

| | | |
|---|-------------|--|
| 2 | bitwise_or | It is used to calculate the bitwise or operation between the corresponding array elements. |
| 3 | invert | It is used to calculate the bitwise not the operation of the array elements. |
| 4 | left_shift | It is used to shift the bits of the binary representation of the elements to the left. |
| 5 | right_shift | It is used to shift the bits of the binary representation of the elements to the right. |

bitwise_and Operation

The NumPy provides the `bitwise_and()` function which is used to calculate the `bitwise_and` operation of the two operands.

The bitwise and operation is performed on the corresponding bits of the binary representation of the operands. If both the corresponding bit in the operands is set to 1, then only the resultant bit in the AND result will be set to 1 otherwise it will be set to 0.

Example

1. `import numpy as np`
- 2.
3. `a = 10`
4. `b = 12`
- 5.
6. `print("binary representation of a:",bin(a))`
7. `print("binary representation of b:",bin(b))`
8. `print("Bitwise-and of a and b: ",np.bitwise_and(a,b))`

Output:

```
binary representation of a: 0b1010
binary representation of b: 0b1100
Bitwise-and of a and b:  8
```

AND Truth Table

The output of the AND result of the two bits is 1 if and only if both the bits are 1 otherwise it will be 0.

| A | B | AND (A, B) |
|---|---|------------|
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

bitwise_or Operator

The NumPy provides the `bitwise_or()` function which is used to calculate the bitwise or operation of the two operands.

The bitwise or operation is performed on the corresponding bits of the binary representation of the operands. If one of the corresponding bit in the operands is set to 1 then the resultant bit in the OR result will be set to 1; otherwise it will be set to 0.

Example

1. `import numpy as np`
- 2.
3. `a = 50`
4. `b = 90`
5. `print("binary representation of a:",bin(a))`
6. `print("binary representation of b:",bin(b))`
7. `print("Bitwise-or of a and b: ",np.bitwise_or(a,b))`

Output:

```
binary representation of a: 0b110010
binary representation of b: 0b1011010
Bitwise-or of a and b:  122
```

Or truth table

The output of the OR result of the two bits is 1 if one of the bits are 1 otherwise it will be 0.

| A | B | Or (A, B) |
|---|---|-----------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

Invert operation

It is used to calculate the bitwise not the operation of the given operand. The 2's complement is returned if the signed integer is passed in the function.

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `arr = np.array([20],dtype = np.uint8)`
4. `print("Binary representation:",np.binary_repr(20,8))`
- 5.
6. `print(np.invert(arr))`
- 7.
8. `print("Binary representation: ", np.binary_repr(235,8))`

Output:

```
Binary representation: 00010100
[235]
Binary representation: 11101011
```

It shifts the bits in the binary representation of the operand to the left by the specified position. An equal number of 0s are appended from the right. Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `print("left shift of 20 by 3 bits",np.left_shift(20, 3))`
- 4.
5. `print("Binary representation of 20 in 8 bits",np.binary_repr(20, 8))`
- 6.
7. `print("Binary representation of 160 in 8 bits",np.binary_repr(160,8))`

Output:

```
left shift of 20 by 3 bits 160
Binary representation of 20 in 8 bits 00010100
Binary representation of 160 in 8 bits 10100000
```

Right Shift Operation

It shifts the bits in the binary representation of the operand to the right by the specified position. An equal number of 0s are appended from the left. Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `print("left shift of 20 by 3 bits",np.right_shift(20, 3))`
- 4.
5. `print("Binary representation of 20 in 8 bits",np.binary_repr(20, 8))`
- 6.
7. `print("Binary representation of 160 in 8 bits",np.binary_repr(160,8))`

Output:

```
left shift of 20 by 3 bits 2
Binary representation of 20 in 8 bits 00010100
Binary representation of 160 in 8 bits 10100000
```

NumPy String Functions

NumPy contains the following functions for the operations on the arrays of dtype string.

| SN | Function | Description |
|----|---------------------------|--|
| 1 | <code>add()</code> | It is used to concatenate the corresponding array elements (strings). |
| 2 | <code>multiply()</code> | It returns the multiple copies of the specified string, i.e., if a string 'hello' is multiplied by 3 then, a string 'hello hello' is returned. |
| 3 | <code>center()</code> | It returns the copy of the string where the original string is centered with the left and right padding filled with the specified number of fill characters. |
| 4 | <code>capitalize()</code> | It returns a copy of the original string in which the first letter of the original string is converted to the Upper Case. |
| 5 | <code>title()</code> | It returns the title cased version of the string, i.e., the first letter of each word of the string is converted into the upper case. |
| 6 | <code>lower()</code> | It returns a copy of the string in which all the letters are converted into the lower case. |
| 7 | <code>upper()</code> | It returns a copy of the string in which all the letters are converted into the upper case. |
| 9 | <code>split()</code> | It returns a list of words in the string. |

| | | |
|----|--------------|--|
| 9 | splitlines() | It returns the list of lines in the string, breaking at line boundaries. |
| 10 | strip() | Returns a copy of the string with the leading and trailing white spaces removed. |
| 11 | join() | It returns a string which is the concatenation of all the strings specified in the given sequence. |
| 12 | replace() | It returns a copy of the string by replacing all occurrences of a particular substring with the specified one. |
| 13 | decode() | It is used to decode the specified string element-wise using the specified codec. |
| 14 | encode() | It is used to encode the decoded string element-wise. |

numpy.char.add() method example

1. `import numpy as np`
2. `print("Concatenating two string arrays:")`
3. `print(np.char.add(['welcome','Hi'], [' to Javatpoint', ' read python']))`

Output:

```
Concatenating two string arrays:
['welcome to Javatpoint' 'Hi read python']
```

numpy.char.multiply() method example

1. `import numpy as np`
2. `print("Printing a string multiple times:")`
3. `print(np.char.multiply("hello ",3))`

Output:

```
Printing a string multiple times:
hello hello hello
```

numpy.char.center() method example

1. `import numpy as np`
2. `print("Padding the string through left and right with the fill char *");`
3. `#np.char.center(string, width, fillchar)`
4. `print(np.char.center("Javatpoint", 20, '*'))`

Output:

```
Padding the string through left and right with the fill char *
*****Javatpoint*****
```

numpy.char.capitalize() method example

1. `import numpy as np`
2. `print("Capitalizing the string using capitalize()...")`

```
3. print(np.char.capitalize("welcome to javatpoint"))
```

Output:

```
Capitalizing the string using capitalize()...
Welcome to javatpoint
```

numpy.char.title() method example

```
1. import numpy as np
2. print("Converting string into title cased version...")
3. print(np.char.title("welcome to javatpoint"))
```

Output:

```
Converting string into title cased version...
Welcome To Javatpoint
```

numpy.char.lower() method example

```
1. import numpy as np
2. print("Converting all the characters of the string into lowercase...")
3. print(np.char.lower("WELCOME TO JAVATPOINT"))
```

Output:

```
Converting all the characters of the string into lowercase...
welcome to javatpoint
```

numpy.char.upper() method example

```
1. import numpy as np
2. print("Converting all the characters of the string into uppercase...")
3. print(np.char.upper("Welcome To Javatpoint"))
```

Output:

```
Converting all the characters of the string into uppercase...
WELCOME TO JAVATPOINT
```

numpy.char.split() method example

```
1. import numpy as np
2. print("Splitting the String word by word..")
3. print(np.char.split("Welcome To Javatpoint"),sep = " ")
```

Output:

```
Splitting the String word by word..
['Welcome', 'To', 'Javatpoint']
```

numpy.char.splitlines() method example

```
1. import numpy as np
2. print("Splitting the String line by line..")
3. print(np.char.splitlines("Welcome\nTo\nJavatpoint"))
```

Output:

```
Splitting the String line by line..
['Welcome', 'To', 'Javatpoint']
```

numpy.char.strip() method example

1. **import** numpy as np
2. str = " welcome to javatpoint "
3. **print**("Original String:",str)
4. **print**("Removing the leading and trailing whitespaces from the string")
5. **print**(np.char.strip(str))

Output:

```
Original String:      welcome to javatpoint
Removing the leading and trailing whitespaces from the string
welcome to javatpoint
```

numpy.char.join() method example

1. **import** numpy as np
2. **print**(np.char.join(':', 'HM'))

Output:

```
H:M
```

numpy.char.replace() method example

1. **import** numpy as np
2. str = "Welcome to Javatpoint"
3. **print**("Original String:",str)
4. **print**("Modified String:",end=" ")
5. **print**(np.char.replace(str, "Welcome to", "www."))

Output:

```
Original String: Welcome to Javatpoint
Modified String: www. Javatpoint
```

numpy.char.encode() and decode() method example

1. **import** numpy as np
2. enstr = np.char.encode("welcome to javatpoint", 'cp500')
3. dstr = np.char.decode(enstr, 'cp500')
4. **print**(enstr)
5. **print**(dstr)

Output:

```
b'\xa6\x85\x93\x83\x96\x94\x85@\xa3\x96@\x91\x81\xa5\x81\xa3\x97\x96\x89\x95\x
a3'
welcome to javatpoint
```

NumPy Mathematical Functions

Numpy contains a large number of mathematical functions which can be used to perform various mathematical operations. The mathematical functions include trigonometric functions, arithmetic functions, and functions for handling complex numbers. Let's discuss the mathematical functions.

Trigonometric functions

Numpy contains the trigonometric functions which are used to calculate the sine, cosine, and tangent of the different angles in radian.

The sin, cos, and tan functions return the trigonometric ratio for the specified angles. Consider the following example.

Example

1. `import numpy as np`
2. `arr = np.array([0, 30, 60, 90, 120, 150, 180])`
3. `print("\nThe sin value of the angles",end = " ")`
4. `print(np.sin(arr * np.pi/180))`
5. `print("\nThe cosine value of the angles",end = " ")`
6. `print(np.cos(arr * np.pi/180))`
7. `print("\nThe tangent value of the angles",end = " ")`
8. `print(np.tan(arr * np.pi/180))`

Output:

```
The sin value of the angles
[0.00000000e+00 5.00000000e-01 8.66025404e-01 1.00000000e+00
 8.66025404e-01 5.00000000e-01 1.22464680e-16]

The cosine value of the angles
[ 1.00000000e+00 8.66025404e-01 5.00000000e-01 6.12323400e-17
 -5.00000000e-01 -8.66025404e-01 -1.00000000e+00]

The tangent value of the angles [ 0.00000000e+00 5.77350269e-01 1.73205081e+00
 1.63312394e+16
 -1.73205081e+00 -5.77350269e-01 -1.22464680e-16]
```

On the other hand, arcsin(), arccos(), and arctan() functions return the trigonometric inverse of the specified angles.

The numpy.degrees() function can be used to verify the result of these trigonometric functions. Consider the following example.

Example

1. `import numpy as np`
2. `arr = np.array([0, 30, 60, 90])`
3. `print("printing the sin values of different angles")`

```

4.
5. sinval = np.sin(arr*np.pi/180)
6.
7. print(sinval)
8. print("printing the inverse of the sin")
9. cosec = np.arcsin(sinval)
10.
11. print(cosec)
12.
13. print("printing the values in degrees")
14. print(np.degrees(cosec))
15.
16. print("\nprinting the cos values of different angles")
17. cosval = np.cos(arr*np.pi/180)
18.
19. print(cosval)
20. print("printing the inverse of the cos")
21. sec = np.acos(cosval)
22. print(sec)
23.
24. print("\nprinting the values in degrees")
25. print(np.degrees(sec))
26.
27. print("\nprinting the tan values of different angles")
28. tanval = np.tan(arr*np.pi/180)
29.
30. print(tanval)
31. print("printing the inverse of the tan")
32. cot = np.arctan(tanval)
33. print(cot)
34.
35. print("\nprinting the values in degrees")
36. print(np.degrees(cot))

```

Output:

```

printing the sin values of different angles
[0.          0.5          0.8660254  1.          ]
printing the inverse of the sin
[0.          0.52359878  1.04719755  1.57079633]
printing the values in degrees
[ 0.  30.  60.  90.]

printing the cos values of different angles
[1.00000000e+00  8.66025404e-01  5.00000000e-01  6.12323400e-17]

```

```

printing the inverse of the cos
[0.          0.52359878 1.04719755 1.57079633]

printing the values in degrees
[ 0. 30. 60. 90.]

printing the tan values of different angles
[0.00000000e+00 5.77350269e-01 1.73205081e+00 1.63312394e+16]
printing the inverse of the tan
[0.          0.52359878 1.04719755 1.57079633]

printing the values in degrees
[ 0. 30. 60. 90.]

```

Rounding Functions

The numpy provides various functions that can be used to truncate the value of a decimal float number rounded to a particular precision of decimal numbers. Let's discuss the rounding functions.

The `numpy.around()` function

This function returns a decimal value rounded to a desired position of the decimal. The syntax of the function is given below.

1. `numpy.around(num, decimals)`

It accepts the following parameters.

| SN | Parameter | Description |
|----|-----------|---|
| 1 | num | It is the input number. |
| 2 | decimals | It is the number of decimals which to which the number is to be rounded. The default value is 0. If this value is negative, then the decimal will be moved to the left. |

Consider the following example.

Example

1. `import numpy as np`
2. `arr = np.array([12.202, 90.23120, 123.020, 23.202])`
3. `print("printing the original array values:",end = " ")`
4. `print(arr)`
5. `print("Array values rounded off to 2 decimal position",np.around(arr, 2))`
6. `print("Array values rounded off to -1 decimal position",np.around(arr, -1))`

Output:

```
printing the original array values: [ 12.202   90.2312 123.02   23.202 ]
Array values rounded off to 2 decimal position [ 12.2    90.23 123.02  23.2 ]
Array values rounded off to -2 decimal position [ 10.   90. 120.  20.]
```

The numpy.floor() function

This function is used to return the floor value of the input data which is the largest integer not greater than the input value. Consider the following example.

Example

1. **import** numpy as np
2. arr = np.array([12.202, 90.23120, 123.020, 23.202])
3. **print**(np.floor(arr))

Output:

```
[ 12.  90. 123.  23.]
```

The numpy.ceil() function

This function is used to return the ceiling value of the array values which is the smallest integer value greater than the array element. Consider the following example.

Example

1. **import** numpy as np
2. arr = np.array([12.202, 90.23120, 123.020, 23.202])
3. **print**(np.ceil(arr))

Output:

```
[ 13.  91. 124.  24.]
```

Numpy statistical functions

Numpy provides various statistical functions which are used to perform some statistical data analysis. In this section of the tutorial, we will discuss the statistical functions provided by the numpy.

Finding the minimum and maximum elements from the array

The numpy.amin() and numpy.amax() functions are used to find the minimum and maximum of the array elements along the specified axis respectively.

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. a = np.array([[2,10,20],[80,43,31],[22,43,10]])

```
4.  
5. print("The original array:\n")  
6. print(a)  
7.  
8.  
9. print("\nThe minimum element among the array:",np.amin(a))  
10. print("The maximum element among the array:",np.amax(a))  
11.  
12. print("\nThe minimum element among the rows of array",np.amin(a,0))  
13. print("The maximum element among the rows of array",np.amax(a,0))  
14.  
15. print("\nThe minimum element among the columns of array",np.amin(a,1))  
16. print("The maximum element among the columns of array",np.amax(a,1))
```

Output:

```
The original array:  
[[ 2 10 20]  
 [80 43 31]  
 [22 43 10]]  
  
The minimum element among the array: 2  
The maximum element among the array: 80  
  
The minimum element among the rows of array [ 2 10 10]  
The maximum element among the rows of array [80 43 31]  
  
The minimum element among the columns of array [ 2 31 10]  
The maximum element among the columns of array [20 80 43]
```

numpy.ptp() function

The name of the function numpy.ptp() is derived from the name peak-to-peak. It is used to return the range of values along an axis. Consider the following example.

Example

```
1. import numpy as np  
2.  
3. a = np.array([[2,10,20],[80,43,31],[22,43,10]])  
4.  
5. print("Original array:\n",a)  
6.  
7. print("\nptp value along axis 1:",np.ptp(a,1))  
8.  
9. print("ptp value along axis 0:",np.ptp(a,0))
```

Output:

```
Original array:  
[[ 2 10 20]  
[80 43 31]  
[22 43 10]]  
  
ptp value along axis 1: [18 49 33]  
ptp value along axis 0: [78 33 21]
```

numpy.percentile() function

The syntax to use the function is given below.

1. `numpy.percentile(input, q, axis)`

It accepts the following parameters.

1. **input:** It is the input array.
2. **q:** It is the percentile (1-100) which is calculated of the array element.
3. **axis:** It is the axis along which the percentile is to be calculated.

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `a = np.array([[2,10,20],[80,43,31],[22,43,10]])`
- 4.
5. `print("Array:\n",a)`
- 6.
7. `print("\nPercentile along axis 0",np.percentile(a, 10,0))`
- 8.
9. `print("Percentile along axis 1",np.percentile(a, 10, 1))`

Output:

```
Array:  
[[ 2 10 20]  
[80 43 31]  
[22 43 10]]  
  
Percentile along axis 0 [ 6. 16.6 12. ]  
Percentile along axis 1 [ 3.6 33.4 12.4]
```

Calculating median, mean, and average of array items

The numpy.median() function:

Median is defined as the value that is used to separate the higher range of data sample with a lower range of data sample. The function `numpy.median()` is used to calculate the median of the multi-dimensional or one-dimensional arrays.

The `numpy.mean()` function:

The mean can be calculated by adding all the items of the arrays dividing by the number of array elements. We can also mention the axis along which the mean can be calculated.

The `numpy.average()` function:

The `numpy.average()` function is used to find the weighted average along the axis of the multi-dimensional arrays where their weights are given in another array.

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `a = np.array([[1,2,3],[4,5,6],[7,8,9]])`
- 4.
5. `print("Array:\n",a)`
- 6.
7. `print("\nMedian of array along axis 0:",np.median(a,0))`
8. `print("Mean of array along axis 0:",np.mean(a,0))`
9. `print("Average of array along axis 1:",np.average(a,1))`

NumPy Sorting and Searching

Numpy provides a variety of functions for sorting and searching. There are various sorting algorithms like quicksort, merge sort and heapsort which is implemented using the `numpy.sort()` function.

The kind of the sorting algorithm to be used in the sort operation must be mentioned in the function call.

Let's discuss the sorting algorithm which is implemented in `numpy.sort()`

| SN | Algorithm | Worst case complexity |
|----|------------|-----------------------|
| 1 | Quick Sort | $O(n^2)$ |
| 2 | Merge Sort | $O(n * \log(n))$ |
| 3 | Heap Sort | $O(n * \log(n))$ |

The syntax to use the numpy.sort() function is given below.

1. `numpy.sort(a, axis, kind, order)`

It accepts the following parameters.

| SN | Parameter | Description |
|----|-----------|---|
| 1 | input | It represents the input array which is to be sorted. |
| 2 | axis | It represents the axis along which the array is to be sorted. If the axis is not mentioned, then the sorting is done along the last available axis. |
| 3 | kind | It represents the type of sorting algorithm which is to be used while sorting. The default is quick sort. |
| 4 | order | It represents the filed according to which the array is to be sorted in the case if the array contains the fields. |

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `a = np.array([[10,2,3],[4,5,6],[7,8,9]])`
- 4.
5. `print("Sorting along the columns:")`
6. `print(np.sort(a))`
- 7.
8. `print("Sorting along the rows:")`
9. `print(np.sort(a, 0))`
- 10.
11. `data_type = np.dtype([('name', 'S10'),('marks',int)])`
- 12.
13. `arr = np.array([('Mukesh',200),('John',251)],dtype = data_type)`
- 14.
15. `print("Sorting data ordered by name")`
- 16.
17. `print(np.sort(arr,order = 'name'))`

Output:

```
Sorting along the columns:
```

```
[[ 2  3 10]
 [ 4  5  6]
 [ 7  8  9]]
Sorting along the rows:
[[ 4  2  3]
 [ 7  5  6]
 [10  8  9]]
Sorting data ordered by name
[(b'John', 251), (b'Mukesh', 200)]
```

numpy.argsort() function

This function is used to perform an indirect sort on an input array that is, it returns an array of indices of data which is used to construct the array of sorted data.

Consider the following example.

Example

```
1. import numpy as np
2.
3. a = np.array([90, 29, 89, 12])
4.
5. print("Original array:\n",a)
6.
7. sort_ind = np.argsort(a)
8.
9. print("Printing indices of sorted data\n",sort_ind)
10.
11.sort_a = a[sort_ind]
12.
13.print("printing sorted array")
14.
15.for i in sort_ind:
16.    print(a[i],end = " ")
```

Output:

```
Original array:
[90 29 89 12]
Printing indices of sorted data
[3 1 2 0]
printing sorted array
12 29 89 90
```

numpy.lexsort() function

This function is used to sort the array using the sequence of keys indirectly. This function performs similarly to the numpy.argsort() which returns the array of indices of sorted data.

Consider the following example.

Example

```
1. import numpy as np
2.
3. a = np.array(['a','b','c','d','e'])
4.
5. b = np.array([12, 90, 380, 12, 211])
6.
7. ind = np.lexsort((a,b))
8.
9. print("printing indices of sorted data")
10.
11. print(ind)
12.
13. print("using the indices to sort the array")
14.
15. for i in ind:
16.     print(a[i],b[i])
```

Output:

```
printing indices of sorted data
[0 3 1 4 2]
using the indices to sort the array
a 12
d 12
b 90
e 211
c 380
```

numpy.nonzero() function

This function is used to find the location of the non-zero elements from the array.

Consider the following example.

Example

```
1. import numpy as np
2.
3. b = np.array([12, 90, 380, 12, 211])
4.
5. print("printing original array",b)
6.
7. print("printing location of the non-zero elements")
8.
9. print(b.nonzero())
```

Output:

```
printing original array [ 12  90 380  12 211]
printing location of the non-zero elements
(array([0, 1, 2, 3, 4]),)
```

numpy.where() function

This function is used to return the indices of all the elements which satisfies a particular condition.

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. b = np.array([12, 90, 380, 12, 211])
- 4.
5. **print**(np.where(b>12))
- 6.
7. c = np.array([[20, 24],[21, 23]])
- 8.
9. **print**(np.where(c>20))

Output:

```
(array([1, 2, 4]),)
(array([0, 1, 1]), array([1, 0, 1]))
```

NumPy Copies and Views

The copy of an input array is physically stored at some other location and the content stored at that particular location is returned which is the copy of the input array whereas the different view of the same memory location is returned in the case of view.

In this section of the tutorial, we will consider the way by which, the different copies and views are generated from some memory location.

Array Assignment

The assignment of a numpy array to another array doesn't make the direct copy of the original array, instead, it makes another array with the same content and same id. It represents the reference to the original array. Changes made on this reference are also reflected in the original array.

The **id()** function returns the universal identifier of the array similar to the pointer in C.

Consider the following example.

Example

1. **import** numpy as np

```
2.  
3. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])  
4.  
5. print("Original Array:\n",a)  
6.  
7. print("\nID of array a:",id(a))  
8.  
9. b = a  
10.  
11. print("\nmaking copy of the array a")  
12.  
13. print("\nID of b:",id(b))  
14.  
15. b.shape = 4,3;  
16.  
17. print("\nChanges on b also reflect to a:")  
18. print(a)
```

Output:

```
Original Array:  
[[ 1  2  3  4]  
 [ 9  0  2  3]  
 [ 1  2  3 19]]  
  
ID of array a: 139663602288640  
  
making copy of the array a  
  
ID of b: 139663602288640  
  
Changes on b also reflect to a:  
[[ 1  2  3]  
 [ 4  9  0]  
 [ 2  3  1]  
 [ 2  3 19]]
```

ndarray.view() method

The ndarray.view() method returns the new array object which contains the same content as the original array does. Since it is a new array object, changes made on this object do not reflect the original array.

Consider the following example.

Example

```
1. import numpy as np  
2.  
3. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])  
4.
```

```

5. print("Original Array:\n",a)
6.
7. print("\nID of array a:",id(a))
8.
9. b = a.view()
10.
11. print("\nID of b:",id(b))
12.
13. print("\nprinting the view b")
14. print(b)
15.
16. b.shape = 4,3;
17.
18. print("\nChanges made to the view b do not reflect a")
19. print("\nOriginal array \n",a)
20. print("\nview\n",b)

```

Output:

```

Original Array:
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

ID of array a: 140280414447456

ID of b: 140280287000656

printing the view b
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

Changes made to the view b do not reflect a

Original array
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

view
[[ 1  2  3]
 [ 4  9  0]
 [ 2  3  1]
 [ 2  3 19]]

```

ndarray.copy() method

It returns the deep copy of the original array which doesn't share any memory with the original array. The modification made to the deep copy of the original array doesn't reflect the original array.

Consider the following example.

Example

```
1. import numpy as np
2.
3. a = np.array([[1,2,3,4],[9,0,2,3],[1,2,3,19]])
4.
5. print("Original Array:\n",a)
6.
7. print("\nID of array a:",id(a))
8.
9. b = a.copy()
10.
11. print("\nID of b:",id(b))
12.
13. print("\nprinting the deep copy b")
14. print(b)
15.
16. b.shape = 4,3;
17.
18. print("\nChanges made to the copy b do not reflect a")
19. print("\nOriginal array \n",a)
20. print("\nCopy\n",b)
```

Output:

```
Original Array:
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

ID of array a: 139895697586176

ID of b: 139895570139296

printing the deep copy b
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

Changes made to the copy b do not reflect a

Original array
[[ 1  2  3  4]
 [ 9  0  2  3]
 [ 1  2  3 19]]

Copy
[[ 1  2  3]
 [ 4  9  0]
 [ 2  3  1]
 [ 2  3 19]]
```

NumPy Matrix Library

NumPy contains a matrix library, i.e. `numpy.matlib` which is used to configure matrices instead of `ndarray` objects.

`numpy.matlib.empty()` function

This function is used to return a new matrix with the uninitialized entries. The syntax to use this function is given below.

1. `numpy.matlib.empty(shape, dtype, order)`

It accepts the following parameter.

1. `shape`: It is the tuple defining the shape of the matrix.
2. `dtype`: It is the data type of the matrix.
3. `order`: It is the insertion order of the matrix, i.e. C or F.

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `import numpy.matlib`
- 4.
5. `print(numpy.matlib.empty((3,3)))`

Output:

```
[ [6.90262230e-310 6.90262230e-310 6.90262304e-310]
  [6.90262304e-310 6.90261674e-310 6.90261552e-310]
  [6.90261326e-310 6.90262311e-310 3.95252517e-322] ]
```

`numpy.matlib.zeros()` function

This function is used to create the matrix where the entries are initialized to zero.

Consider the following example.

Example

1. `import numpy as np`
- 2.
3. `import numpy.matlib`
- 4.
5. `print(numpy.matlib.zeros((4,3)))`

Output:

```
[[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]
```

numpy.matlib.ones() function

This function returns a matrix with all the elements initialized to 1.

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. **import** numpy.matlib
- 4.
5. **print**(numpy.matlib.ones((2,2)))

Output:

```
[[1. 1.]  
 [1. 1.]]
```

numpy.matlib.eye() function

This function returns a matrix with the diagonal elements initialized to 1 and zero elsewhere. The syntax to use this function is given below.

1. numpy.matlib.eye(n, m, k, dtype)

It accepts the following parameters.

1. n: It represents the number of rows in the resulting matrix.
2. m: It represents the number of columns, defaults to n.
3. k: It is the index of diagonal.
4. dtype: It is the data type of the output

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. **import** numpy.matlib
- 4.
5. **print**(numpy.matlib.eye(n = 3, M = 3, k = 0, dtype = int))

Output:

```
[[1 0 0]
```

```
[0 1 0]
[0 0 1]
```

numpy.matlib.identity() function

This function is used to return an identity matrix of the given size. An identity matrix is the one with diagonal elements initialized to 1 and all other elements to zero.

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. **import** numpy.matlib
- 4.
5. **print**(numpy.matlib.identity(5, dtype = int))

Output:

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

numpy.matlib.rand() function

This function is used to generate a matrix where all the entries are initialized with random values.

Consider the following example.

Example

1. **import** numpy as np
- 2.
3. **import** numpy.matlib
- 4.
5. **print**(numpy.matlib.rand(3,3))

Output:

```
[[0.86201511 0.86980769 0.06704884]
 [0.80531086 0.53814098 0.84394673]
 [0.85653048 0.8146121 0.35744405]]
```