

# Python - Quick Guide

## Python - Overview

Python is a high-level, multi-paradigm programming language. As Python is an interpreter-based language, it is easier to learn compared to some of the other mainstream languages. Python is a dynamically typed language with very intuitive data types.

Python is an open-source and cross-platform programming language. It is available for use under **Python Software Foundation License** (compatible to GNU General Public License) on all the major operating system platforms Linux, Windows and Mac OS.

The design philosophy of Python emphasizes on simplicity, readability and unambiguity. Python is known for its batteries included approach as Python software is distributed with a comprehensive standard library of functions and modules.

Python's design philosophy is documented in the **Zen of Python**. It consists of nineteen aphorisms such as –

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated

To obtain the complete Zen of Python document, type **import this** in the Python shell –

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
```

Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now *is* better than never.  
Although never *is* often better than \*right\* now.  
If the implementation *is* hard to explain, it's a bad idea.  
If the implementation *is* easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!

Python supports imperative, structured as well as object-oriented programming methodology. It provides features of functional programming as well.

## Python - History

**Guido Van Rossum**, a Dutch programmer, created Python programming language. In the late 80's, he had been working on the development of ABC language in a computer science research institute named **Centrum Wiskunde & Informatica** (CWI) in the Netherlands. In 1991, Van Rossum conceived and published Python as a successor of ABC language.

For many uninitiated people, the word Python is related to a species of snake. Rossum though attributes the choice of the name Python to a popular comedy series "**Monty Python's Flying Circus**" on BBC.

Being the principal architect of Python, the developer community conferred upon him the title of "**Benevolent Dictator for Life** (BDFL). However, in 2018, Rossum relinquished the title. Thereafter, the development and distribution of the reference implementation of Python is handled by a nonprofit organization **Python Software Foundation**.

Important stages in the history of Python –

## Python 0.9.0

Python's first published version is 0.9. It was released in February 1991. It consisted of support for core object-oriented programming principles.

## Python 1.0

In January 1994, version 1.0 was released, armed with functional programming tools, features like support for complex numbers etc.

Learn **Python** in-depth with real-world projects through our **Python certification course**. Enroll and become a certified expert to boost your career.

## Python 2.0

Next major version – Python 2.0 was launched in October 2000. Many new features such as list comprehension, garbage collection and Unicode support were included with it.

## Python 3.0

Python 3.0, a completely revamped version of Python was released in December 2008. The primary objective of this revamp was to remove a lot of discrepancies that had crept in Python 2.x versions. Python 3 was backported to Python 2.6. It also included a utility named as **python2to3** to facilitate automatic translation of Python 2 code to Python 3.

## EOL for Python 2.x

Even after the release of Python 3, Python Software Foundation continued to support the Python 2 branch with incremental micro versions till 2019. However, it decided to discontinue the support by the end of year 2020, at which time Python 2.7.17 was the last version in the branch.

## Current Version

Meanwhile, more and more features have been incorporated into Python's 3.x branch. As of date, Python **3.11.2** is the current stable version, released in February 2023.

## What's New in Python 3.11?

One of the most important features of Python's version 3.11 is the significant improvement in speed. According to Python's official documentation, this version is faster than the previous version (3.10) by up to 60%. It also states that the standard benchmark suite shows a 25% faster execution rate.

- Python 3.11 has a better exception messaging. Instead of generating a long traceback on the occurrence of an exception, we now get the exact expression causing the error.
- As per the recommendations of PEP 678, the **add\_note()** method is added to the `BaseException` class. You can call this method inside the `except` clause and pass a custom error message.
- It also adds the **croot()** function in the `maths` module. It returns the cube root of a given number.
- A new module **tomllib** is added in the standard library. TOML (Tom's Obvious Minimal Language) can be parsed with `tomllib` module function.

## Python - Features

In this chapter, let's highlight some of the important features of Python that make it widely popular.

### Python is Easy to Learn

This is one of the most important reasons for the popularity of Python. Python has a limited set of keywords. Its features such as simple syntax, usage of indentation to avoid

clutter of curly brackets and dynamic typing that doesn't necessitate prior declaration of variable help a beginner to learn Python quickly and easily.

## Python is Interpreter Based

Instructions in any programming languages must be translated into machine code for the processor to execute them. Programming languages are either compiler based or interpreter based.

In case of a compiler, a machine language version of the entire source program is generated. The conversion fails even if there is a single erroneous statement. Hence, the development process is tedious for the beginners. The C family languages (including C, C++, Java, C Sharp etc) are compiler based.

Python is an interpreter based language. The interpreter takes one instruction from the source code at a time, translates it into machine code and executes it. Instructions before the first occurrence of error are executed. With this feature, it is easier to debug the program and thus proves useful for the beginner level programmer to gain confidence gradually. Python therefore is a beginner-friendly language.

## Python is Interactive

Standard Python distribution comes with an interactive shell that works on the principle of REPL (Read – Evaluate – Print – Loop). The shell presents a Python prompt `>>>`. You can type any valid Python expression and press Enter. Python interpreter immediately returns the response and the prompt comes back to read the next expression.

```
>>> 2*3+1  
7
```

```
>>> print ("Hello World")
Hello World
```

The interactive mode is especially useful to get familiar with a library and test out its functionality. You can try out small code snippets in interactive mode before writing a program.

## Python is MultiParadigm

Python is a completely object-oriented language. Everything in a Python program is an object. However, Python conveniently encapsulates its object orientation to be used as an imperative or procedural language – such as C. Python also provides certain functionality that resembles functional programming. Moreover, certain third-party tools have been developed to support other programming paradigms such as aspect-oriented and logic programming.

## Python's Standard Library

Even though it has a very few keywords (only Thirty Five), Python software is distributed with a standard library made of large number of modules and packages. Thus Python has out of box support for programming needs such as serialization, data compression, internet data handling, and many more. Python is known for its batteries included approach.



## Python is Open Source and Cross Platform

Python's standard distribution can be downloaded from <https://www.python.org/downloads/> without any restrictions. You can download pre-

compiled binaries for various operating system platforms. In addition, the source code is also freely available, which is why it comes under open source category.

Python software (along with the documentation) is distributed under Python Software Foundation License. It is a BSD style permissive software license and compatible to GNU GPL (General Public License).

Python is a cross-platform language. Pre-compiled binaries are available for use on various operating system platforms such as Windows, Linux, Mac OS, Android OS. The reference implementation of Python is called CPython and is written in C. You can download the source code and compile it for your OS platform.

A Python program is first compiled to an intermediate platform independent byte code. The virtual machine inside the interpreter then executes the byte code. This behaviour makes Python a cross-platform language, and thus a Python program can be easily ported from one OS platform to other.

## Python for GUI Applications

Python's standard distribution has an excellent graphics library called Tkinter. It is a Python port for the vastly popular GUI toolkit called TCL/Tk. You can build attractive user-friendly GUI applications in Python. GUI toolkits are generally written in C/C++. Many of them have been ported to Python. Examples are PyQt, WxWidgets, PySimpleGUI etc.

## Python's Database Connectivity

Almost any type of database can be used as a backend with the Python application. DB-API is a set of specifications for database driver software to let Python communicate with a relational database. With many third party libraries, Python can also work with NoSQL databases such as MongoDB.

## Python is Extensible

The term extensibility implies the ability to add new features or modify existing features. As stated earlier, CPython (which is Python's reference implementation) is written in C. Hence one can easily write modules/libraries in C and incorporate them in the standard library. There are other implementations of Python such as Jython (written in Java) and IPython (written in C#). Hence, it is possible to write and merge new functionality in these implementations with Java and C# respectively.

## Python's Active Developer Community

As a result of Python's popularity and open-source nature, a large number of Python developers often interact with online forums and conferences. Python Software Foundation also has a significant member base, involved in the organization's mission to "promote, protect, and advance the Python programming language"

Python also enjoys a significant institutional support. Major IT companies Google, Microsoft, and Meta contribute immensely by preparing documentation and other resources.

## Python vs C++

Both Python and C++ are among the most popular programming languages. Both of them have their advantages and disadvantages. In this chapter, we shall take a look at their characteristic features.

### Compiled vs Interpreted

Like C, C++ is also a compiler-based language. A compiler translates the entire code in a machine language code specific to the operating system in use and processor architecture.

Python is interpreter-based language. The interpreter executes the source code line by line.

### Cross platform

When a C++ source code such as hello.cpp is compiled on Linux, it can be only run on any other computer with Linux operating system. If required to run on other OS, it needs to be compiled.

Python interpreter doesn't produce compiled code. Source code is converted to byte code every time it is run on any operating system without any changes or additional steps.

### Portability

Python code is easily portable from one OS to other. C++ code is not portable as it must be recompiled if the OS changes.

### Speed of Development

C++ program is compiled to the machine code. Hence, its execution is faster than interpreter based language.

Python interpreter doesn't generate the machine code. Conversion of intermediate byte code to machine language is done on each execution of program.

If a program is to be used frequently, C++ is more efficient than Python.

## Easy to Learn

Compared to C++, Python has a simpler syntax. Its code is more readable. Writing C++ code seems daunting in the beginning because of complicated syntax rule such as use of curly braces and semicolon for sentence termination.

Python doesn't use curly brackets for marking a block of statements. Instead, it uses indents. Statements of similar indent level mark a block. This makes a Python program more readable.

## Static vs Dynamic Typing

C++ is a statically typed language. The type of variables for storing data need to be declared in the beginning. Undeclared variables can't be used. Once a variable is declared to be of a certain type, value of only that type can be stored in it.

Python is a dynamically typed language. It doesn't require a variable to be declared before assigning it a value. Since, a variable may store any type of data, it is called dynamically typed.

## OOP Concepts

Both C++ and Python implement object oriented programming concepts. C++ is closer to the theory of OOP than Python. C++ supports the concept of data encapsulation as the visibility of the variables can be defined as public, private and protected.

Python doesn't have the provision of defining the visibility. Unlike C++, Python doesn't support method overloading. Because it is dynamically typed, all the methods are polymorphic in nature by default.

C++ is in fact an extension of C. One can say that additional keywords are added in C so that it supports OOP. Hence, we can write a C type procedure oriented program in C++.

Python is completely object oriented language. Python's data model is such that, even if you can adapt a procedure oriented approach, Python internally uses object-oriented methodology.

## Garbage Collection

C++ uses the concept of pointers. Unused memory in a C++ program is not cleared automatically. In C++, the process of garbage collection is manual. Hence, a C++ program is likely to face memory related exceptional behavior.

Python has a mechanism of automatic garbage collection. Hence, Python program is more robust and less prone to memory related issues.

## Application Areas

Because C++ program compiles directly to machine code, it is more suitable for systems programming, writing device drivers, embedded systems and operating system utilities.

Python program is suitable for application programming. Its main area of application today is data science, machine learning, API development etc.

The following table summarizes the comparison between C++ and Python –

Criteria	C++	Python
Execution	Compiler based	Interpreter based
Typing	Static typing	Dynamic typing
Portability	Not portable	Highly portable
Garbage collection	Manual	Automatic
Syntax	Tedious	Simple
Performance	Faster execution	Slower execution
Application areas	Embedded systems, device drivers	Machine learning, web applications

## Python - Hello World Program

Hello World program is a basic computer code written in a general purpose programming language, used as a test program. It doesn't ask for any input and displays a Hello World message on the output console. It is used to test if the software needed to compile and run the program has been installed correctly.

It is very easy to display the Hello World message using the Python interpreter. Launch the interpreter from a command terminal of your operating system and issue the print statement from the Python prompt as follows –

```
PS C:\Users\mlath> python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World")
Hello World
```

Similarly, Hello World message is printed in Linux.

```
mvl@GNVBL3:~$ python3
Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World")
Hello World
```

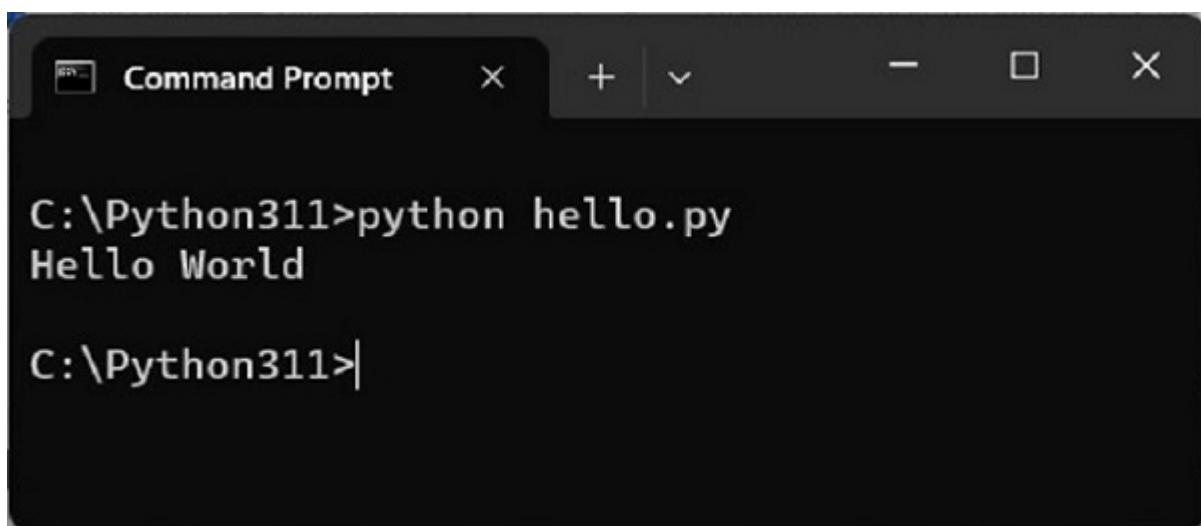
Python interpreter also works in scripted mode. Open any text editor, enter the following text and save as Hello.py

```
print ("Hello World")
```

For Windows OS, open the command prompt terminal (CMD) and run the program as shown below –

```
C:\Python311>python hello.py
Hello World
```

The terminal shows the Hello World message.



While working on Ubuntu Linux, you have to follow the same steps, save the code and run from Linux terminal. We use vi editor for saving the program.

A screenshot of a terminal window titled "mvl@GNVBGL3: ~". The terminal contains the Python code `print ("Hello World")`. Below the code, there is a diff tool interface with the following labels: "-- REPLACE --" on the left, "1,1" in the center, and "All" on the right. The background of the terminal is dark, and the code is displayed in white and red.

To run the program from Linux terminal

```
mvl@GNVBGL3:~$ python3 hello.py  
Hello World
```

In Linux, you can convert a Python program into a self executable script. The first statement in the code should be a shebang. It must contain the path to Python executable. In Linux, Python is installed in /usr/bin directory, and the name of the executable is python3. Hence, we add this statement to hello.py file

```
#!/usr/bin/python3  
print ("Hello World")
```

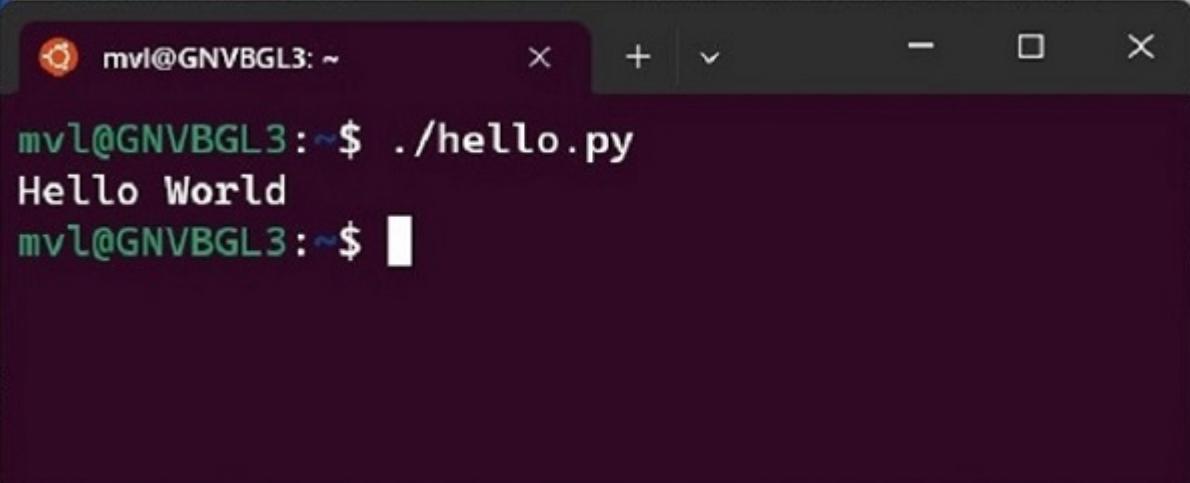
You also need to give the file executable permission by using the chmod +x command

```
mvl@GNVBGL3:~$ chmod +x hello.py
```

Then, you can run the program with following command line –

```
mvl@GNVBGL3:~$ ./hello.py
```

The **output** is shown below –



```
mvl@GNVBGL3:~$ ./hello.py
Hello World
mvl@GNVBGL3:~$
```

Thus, we can write and run Hello World program in Python using the interpreter mode and script mode.

## Python - Application Areas

Python is a general-purpose programming language. It is suitable for development of wide range of software applications. Over last few years Python is the preferred language of choice for developers in following application areas –

### Python for Data Science

Python's recent meteoric rise in the popularity charts is largely due its Data science libraries. Python has become an essential skill for data scientists. Today, real time web applications, mobile applications and other devices generate huge amount of data. Python's data science libraries help companies generate business insights from this data.

Libraries like NumPy, Pandas and Matplotlib are extensively used to apply mathematical algorithms to the data and generate visualizations. Commercial and community Python distributions like Anaconda and ActiveState bundle all the essential libraries required for data science.

### Python for Machine Learning

Python libraries such as Scikit-learn and TensorFlow help in building models for prediction of trends like customer satisfaction, projected values of stocks etc. based upon the past data. Machine learning applications include (but not restricted to) medical diagnosis, statistical arbitrage, basket analysis, sales prediction etc.

### Python for Web Development

Python's web frameworks facilitate rapid web application development. Django, Pyramid, Flask are very popular among the web developer community. etc. make it very easy to develop and deploy simple as well as complex web applications.

Latest versions of Python provide asynchronous programming support. Modern web frameworks leverage this feature to develop fast and high performance web apps and APIs.

## Python for Computer Vision and Image processing

OpenCV is a widely popular library for capturing and processing images. Image processing algorithms extract information from images, reconstruct image and video data. Computer Vision uses image processing for face detection and pattern recognition. OpenCV is a C++ library. Its Python port is extensively used because of its rapid development feature.

Some of the application areas of computer vision are robotics, industrial surveillance and automation, biometrics etc.

## Python for Embedded Systems and IoT

Micropython (<https://micropython.org/>), a lightweight version especially for microcontrollers like Arduino. Many automation products, robotics, IoT, and kiosk applications are built around Arduino and programmed with Micropython. Raspberry Pi is also very popular low cost single board computer used for these type of applications.

## Python for Job Scheduling and Automation

Python found one of its first applications in automating CRON (Command Run ON) jobs. Certain tasks like periodic data backups, can be written in Python scripts scheduled to be invoked automatically by operating system scheduler.

Many software products like Maya embed Python API for writing automation scripts (something similar to Excel macros).

## Try Python Online

If you are new to Python, it is a good idea to get yourself familiar with the language syntax and features by trying out one of the many online resources, before you proceed to install Python software on your computer.

You can launch Python interactive shell from the home page of Python's official website <https://www.python.org/>.

The screenshot shows the Python.org homepage. At the top, there's a navigation bar with links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the navigation is a search bar with a 'GO' button and a 'Socialize' link. The main content area features the Python logo and a section titled 'Quick & Easy to Learn'. It includes a code snippet demonstrating basic Python syntax:

```
# Simple output (with Unix-style line endings)
>>> print("Hello, I'm Python!")
Hello, I'm Python!
# Input, assignment
>>> name = input('What is your name?\n')
What is your name?
Python
>>> print(f'Hi, {name}.')
Hi, Python.
```

Below this, a callout box highlights the ease of learning Python with the text: "Experienced programmers in any other language can pick up Python very quickly, and beginners find the clean syntax and indentation structure easy to learn. [Whet your appetite](#) with our Python 3 overview." There are also numbered buttons (1-5) for navigating through the page.

At the bottom of the main content area, a banner states: "Python is a programming language that lets you work quickly and integrate systems more effectively. [» Learn More](#)".

In front of the Python prompt (>>>), any valid Python expression can be entered and evaluated.

The screenshot shows an online Python console interface. The top navigation bar includes links for About, Downloads, Documentation, Community, Success Stories, News, and Events. The main area of the console displays the following Python session:

```
Python 3.10.5 (main, Jul 22 2022, 17:09:35) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> b=3
>>> print (a+b)
5
>>> print ("Hello world")
Hello world
>>> 
```

At the bottom right of the console window, it says "Online console from PythonAnywhere".

The Tutorialspoint website also has a **Coding Ground** section –

(<https://www.tutorialspoint.com/codingground.htm>)

Here you can find online compilers for various languages including Python. Visit [https://www.tutorialspoint.com/execute\\_python\\_online.php](https://www.tutorialspoint.com/execute_python_online.php). You can experiment with the interactive mode and the scripted mode of Python interpreter.

The screenshot shows a web-based Python interpreter interface. On the left, a code editor displays the following Python script:

```

1 # Online Python-3 Compiler (Interpreter)
2 a = 2
3 b = 3
4 print(a + b)
5 print("Hello World")

```

On the right, a terminal window shows the execution of this code. The output is:

```

webmaster@e90e4284e606:/home$ python3
Python 3.10.6 (main, Nov 14 2022, 16:10
:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or
"license" for more information.

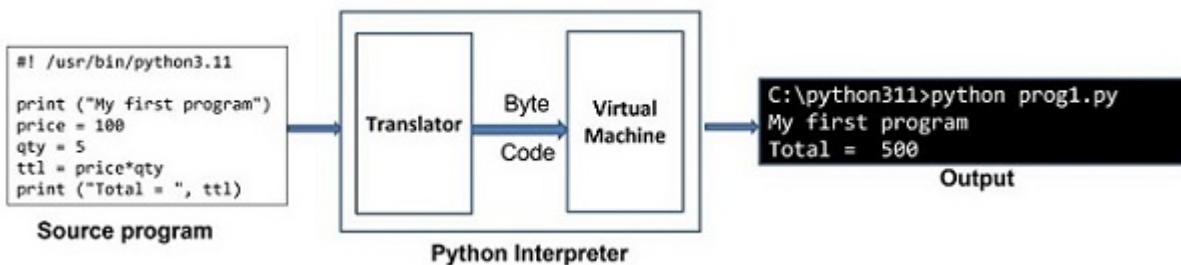
>>> a=2
>>> b=3
>>> print (a+b)
5
>>> print ("Hello Python")
Hello Python
>>>

```

## Python - Interpreter

Python is an interpreter-based language. In a Linux system, Python's executable is installed in **/usr/bin/** directory. For Windows, the executable (python.exe) is found in the installation folder (for example **C:\python311**). In this chapter, you will **how Python interpreter works**, its interactive and scripted mode.

Python code is executed by one statement at a time method. Python interpreter has two components. The translator checks the statement for syntax. If found correct, it generates an intermediate byte code. There is a Python virtual machine which then converts the byte code in native binary and executes it. The following diagram illustrates the mechanism:



Python interpreter has an interactive mode and a scripted mode.

### Interactive Mode

When launched from a command line terminal without any additional options, a Python prompt **>>>** appears and the Python interpreter works on the principle of **REPL (Read, Evaluate, Print, Loop)**. Each command entered in front of the Python prompt is read, translated and executed. A typical interactive session is as follows.

```
>>> price = 100
>>> qty = 5
>>> ttl = price*qty
>>> ttl
500
>>> print ("Total = ", ttl)
Total = 500
```

To close the interactive session, enter the end-of-line character (ctrl+D for Linux and ctrl+Z for Windows). You may also type **quit()** in front of the Python prompt and press Enter to return to the OS prompt.

The interactive shell available with standard Python distribution is not equipped with features like line editing, history search, auto-completion etc. You can use other advanced interactive interpreter software such as **IPython** and **bpython**.

## Scripting Mode

Instead of entering and obtaining the result of one instruction at a time – as in the interactive environment, it is possible to save a set of instructions in a text file, make sure that it has **.py** extension, and use the name as the command line parameter for Python command.

Save the following lines as **prog1.py**, with the use of any text editor such as vim on Linux or Notepad on Windows.

```
</> Open Compiler
print ("My first program")
price = 100
qty = 5
ttl = price*qty
print ("Total = ", ttl)
```

Launch Python with this name as command line argument.

```
C:\Users\Acer>python prog1.py
My first program
Total = 500
```

Note that even though Python executes the entire script, it is still executed in one-by-one fashion.

In case of any compiler-based language such as Java, the source code is not converted in byte code unless the entire code is error-free. In Python, on the other hand, statements are executed until first occurrence of error is encountered.

Let us introduce an error purposefully in the above code.

&lt;/&gt;

Open Compiler

```
print ("My first program")
price = 100
qty = 5
ttl = prive*qty #Error in this statement
print ("Total = ", ttl)
```

Note the misspelt variable **prive** instead of **price**. Try to execute the script again as before –

```
C:\Users\Acer>python prog1.py
My first program
Traceback (most recent call last):
  File "C:\Python311\prog1.py", line 4, in <module>
    ttl = prive*qty
           ^
NameError: name 'prive' is not defined. Did you mean: 'price'?
```

Note that the statements before the erroneous statement are executed and then the error message appears. Thus it is now clear that Python script is executed in interpreted manner.

In addition to executing the Python script as above, the script itself can be a selfexecutable in Linux, like a shell script. You have to add a **shebang** line on top of the script. The shebang indicates which executable is used to interpret Python statements in the script. Very first line of the script starts with **#!** And followed by the path to Python executable.

Modify the prog1.py script as follows –

```
#!/usr/bin/python3.11
print ("My first program")
price = 100
qty = 5
ttl = price*qty
print ("Total = ", ttl)
```

To mark the script as self-executable, use the **chmod** command

```
user@ubuntu20:~$ chmod +x prog1.py
```

You can now execute the script directly, without using it as a command-line argument.

```
user@ubuntu20:~$ ./hello.py
```

## IPython

IPython (stands for **Interactive Python**) is an enhanced and powerful interactive environment for Python with many functionalities compared to the standard Python shell. IPython was originally developed by Fernando Perez in 2001.

IPython has the following important features –

- IPython's object introspection ability to check properties of an object during runtime.
- Its syntax highlighting proves to be useful in identifying the language elements such as keywords, variables etc.
- The history of interactions is internally stored and can be reproduced.
- Tab completion of keywords, variables and function names is one of the most important features.
- IPython's Magic command system is useful for controlling Python environment and performing OS tasks.
- It is the main kernel for Jupyter notebook and other front-end tools of Project Jupyter.

Install IPython with PIP installer utility.

```
pip3 install ipython
```

Launch IPython from command-line

```
C:\Users\Acer>ipython
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type 'copyright', 'credits' or 'license' for more information
IPython 8.4.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]:
```

Instead of the regular >>> prompt as in standard interpreter, you will notice two major IPython prompts as explained below –

- In[1] appears before any input expression.
- Out[1] appears before the Output appears.

```
In [1]: price = 100
In [2]: quantity = 5
In [3]: ttl = price*quantity
In [4]: ttl
Out[4]: 500
In [5]:
```

Tab completion is one of the most useful enhancements provided by IPython. IPython pops up appropriate list of methods as you press tab key after dot in front of object.

In the following example, a string is defined. Press tab key after the "." symbol and as a response, the attributes of string class are shown. You can navigate to the required one.

```
In [5]: var = "Hello World"
In [6]: var.
    capitalize() encode      format      isalpha      isidentifier
    casefold   endswith    format_map  isascii      islower
    center     expandtabs  index       isdecimal    isnumeric
    count      find        isalnum     isdigit      isprintable
    
```

IPython provides information (introspection) of any object by putting '?' in front of it. It includes docstring, function definitions and constructor details of class. For example to explore the string object var defined above, in the input prompt enter var?.

```
In [5]: var = "Hello World"
In [6]: var?
Type: str
String form: Hello World
Length: 11
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str
Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
```

```
encoding defaults to sys.setdefaultencoding().
errors defaults to 'strict'.
```

IPython's magic functions are extremely powerful. Line magics let you run DOS commands inside IPython. Let us run the **dir** command from within IPython console

```
In [8]: !dir *.exe
Volume in drive F has no label.
Volume Serial Number is E20D-C4B9

Directory of F:\Python311

07-02-2023 16:55      103,192 python.exe
07-02-2023 16:55      101,656 pythonw.exe
              2 File(s)   204,848 bytes
                 0 Dir(s)  105,260,306,432 bytes free
```

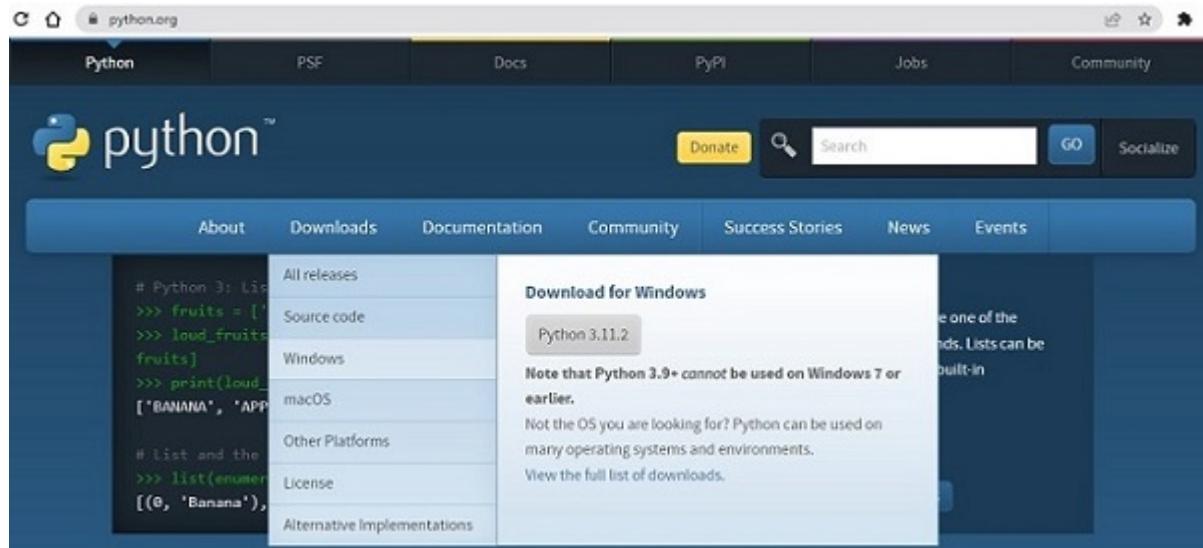
Jupyter notebook is a web-based interface to programming environments of Python, Julia, R and many others. For Python, it uses IPython as its main kernel.

## Python - Environment Setup

First step in the journey of learning Python is to install it on your machine. Today most computer machines, especially having Linux OS, have Python pre-installed. However, it may not be the latest version.

In this section, we shall learn to install the latest version of Python, **Python 3.11.2**, on Linux, Windows and Mac OS.

Latest version of Python for all the operating system environments can be downloaded from PSF's official website.



## Install Python on Ubuntu Linux

To check whether Python is already installed, open the Linux terminal and enter the following command –

```
user@ubuntu20:~$ python3 --version
```

In Ubuntu Linux, the easiest way to install Python is to use **apt – Advanced Packaging Tool**. It is always recommended to update the list of packages in all the configured repositories.

```
user@ubuntu20:~$ sudo apt update
```

Even after the update, the latest version of Python may not be available for install, depending upon the version of Ubuntu you are using. To overcome this, add the **deadsnakes** repository.

```
user@ubuntu20:~$ sudo apt-get install software-properties-common  
user@ubuntu20:~$ sudo add-apt-repository ppa:deadsnakes/ppa
```

Update the package list again.

```
user@ubuntu20:~$ sudo apt update
```

To install the latest Python 3.11 version, enter the following command in the terminal –

```
user@ubuntu20:~$ sudo apt-get install python3.11
```

Check whether it has been properly installed.

```
user@ubuntu20:~$ python3.11  
Python 3.11.2 (main, Feb 8 2023, 14:49:24) [GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print ("Hello World")  
Hello World  
>>>
```

## Install Python on Windows

It may be noted that Python's version 3.10 onwards cannot be installed on Windows 7 or earlier operating systems.

The recommended way to install Python is to use the official installer. Link to the latest stable version is given on the home page itself. It is also found at

[https://www.python.org/downloads/windows/.](https://www.python.org/downloads/windows/)

You can find embeddable packages and installers for 32 as well as 64-bit architecture.

- Python 3.11.2 - Feb. 8, 2023

**Note that Python 3.11.2 cannot be used on Windows 7 or earlier.**

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- Download Windows installer (64-bit)
- Download Windows installer (ARM64)

Let us download 64-bit Windows installer –

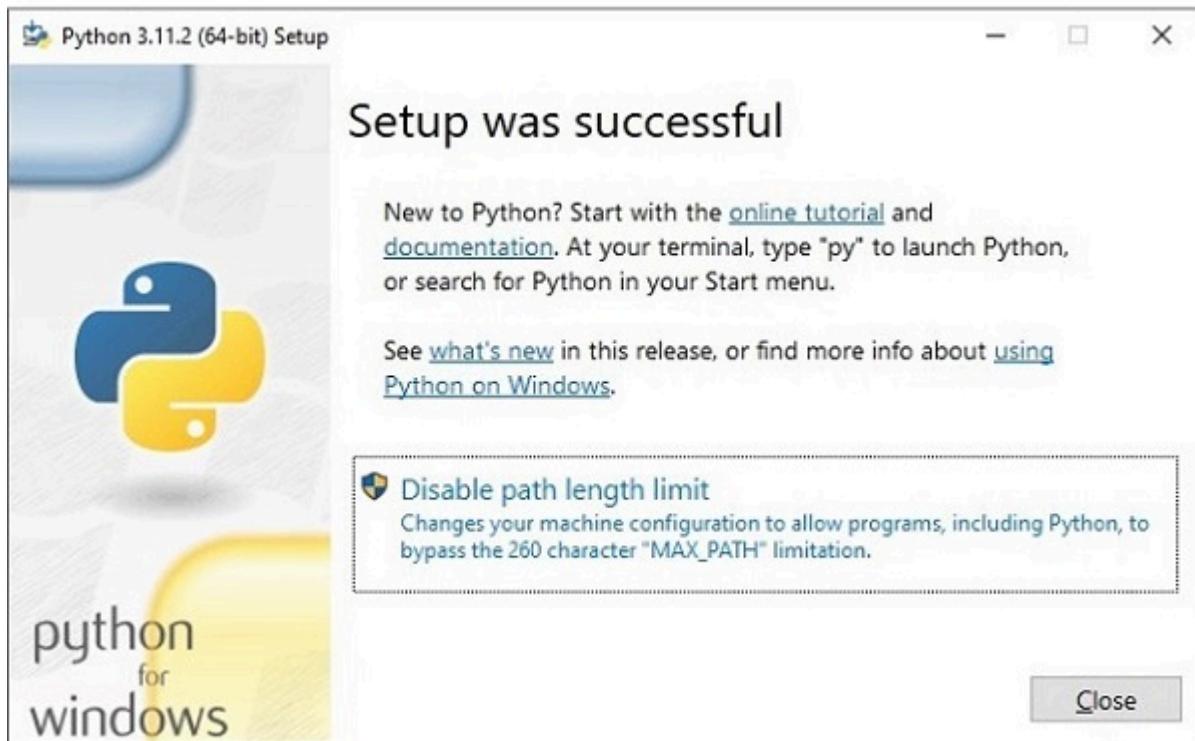
(<https://www.python.org/ftp/python/3.11.2/python-3.11.2-amd64.exe>)

Double click on the file where it has been downloaded to start the installation.



Although you can straight away proceed by clicking the Install Now button, it is advised to choose the installation folder with a relatively shorter path, and tick the second check box to update the PATH variable.

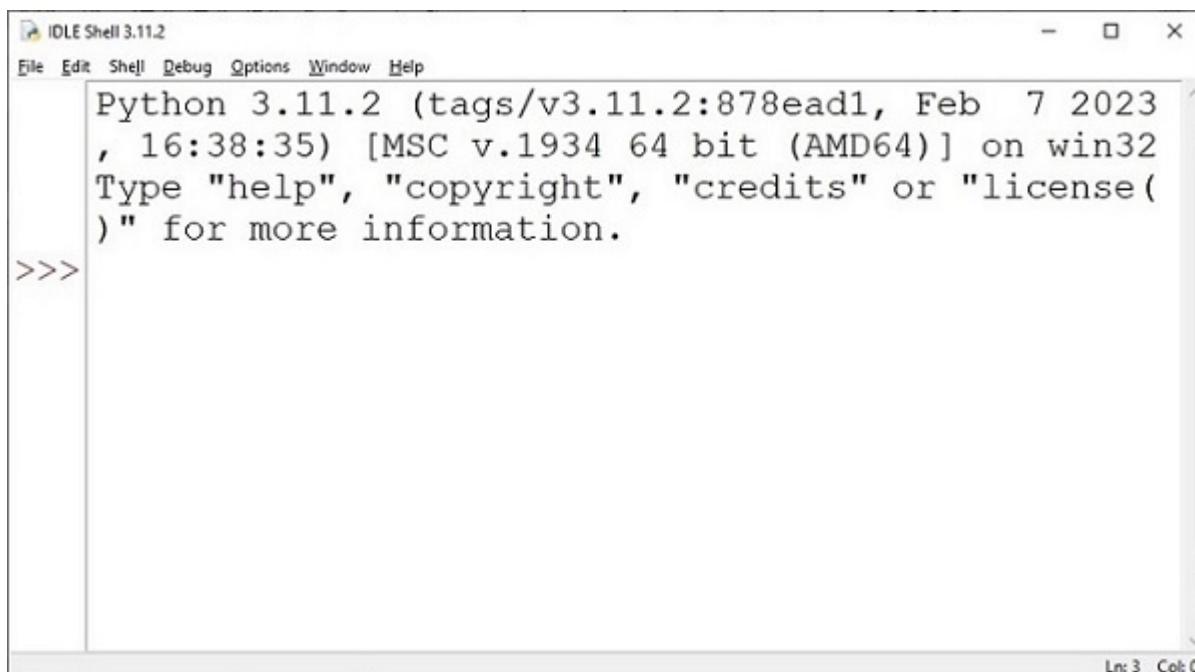
Accept defaults for rest of the steps in this installation wizard to complete the installation.



Open the Window Command Prompt terminal and run Python to check the success of installation.

```
C:\Users\Acer>python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python's standard library has an executable module called **IDLE** – short for **Integrated Development and Learning Environment**. Find it from Window start menu and launch.



IDLE contains Python shell (interactive interpreter) and a customizable multi-window text editor with features such as syntax highlighting, smart indent, auto completion etc. It is cross-platform so works the same on Windows, MacOS and Linux. It also has a debugger with provision to set breakpoints, stepping, and viewing of global and local namespaces.

## Install Python on MacOS

Earlier versions of MacOS used to have Python 2.7 pre-installed in it. However, now that the version no longer supported, it has been discontinued. Hence, you need to install Python on your own.

On a Mac computer, Python can be installed by two methods –

- Using the official installer
- Manual installation with homebrew

You can find macOS 64-bit universal2 installer on the downloads page of the official website –

<https://www.python.org/ftp/python/3.11.2/python-3.11.2-macos11.pkg>

The installation process is more or less similar to that on Windows. Usually, accepting the default options during the wizard steps should do the work.



The frequently required utilities such as PIP and IDLE are also installed by this installation wizard.

Alternately, you can opt for the installation from command line. You need to install **Homebrew**, Mac's package manager, if it is not already available. You can follow the instructions for installation at <https://docs.brew.sh/Installation>.

After that, open the terminal and enter the following commands –

```
brew update && brew upgrade  
brew install python3
```

Latest version of Python will now be installed.

## Install Python from Source Code

If you are an experienced developer, with good knowledge of C++ and Git tool, you can follow the instructions in this section to build Python executable along with the modules in the standard library.

You must have the C compiler for the OS that you are on. In Ubuntu and MacOS, **gcc** compiler is available. For Windows, you should install Visual Studio 2017 or later.

## Steps to Build Python on Linux/Mac

Download the source code of Python's latest version either from Python's official website or its GitHub repository.

Download the source tarball :

<https://www.python.org/ftp/python/3.11.2/Python3.11.2.tgz>

Extract the files with the command –

```
tar -xvzf /home/python/Python-3.11.2.tgz
```

Alternately, clone the main branch of Python's GitHub repository. (You should have git installed)

```
git clone -b main https://github.com/python/cpython
```

A configure script comes in the source code. Running this script will create the makefile.

```
./configure --enable-optimizations
```

Followed by this, use the make tool to build the files and then make install to put the final files in /usr/bin/ directory.

```
make  
make install
```

Python has been successfully built from the source code.

If you use Windows, make sure you have **Visual Studio 2017** and **Git for Windows** installed. Clone the Python source code repository by the same command as above.

Open the windows command prompt in the folder where the source code is placed. Run the following batch file

```
PCbuild\get_externals.bat
```

This downloads the source code dependencies (OpenSSL, Tk etc.)

Open Visual Studio and **PCbuild/sbuild.sln** solution, and build (press F10) the debug folder shows **python\_d.exe** which is the debug version of Python executable.

To build from command prompt, use the following command –

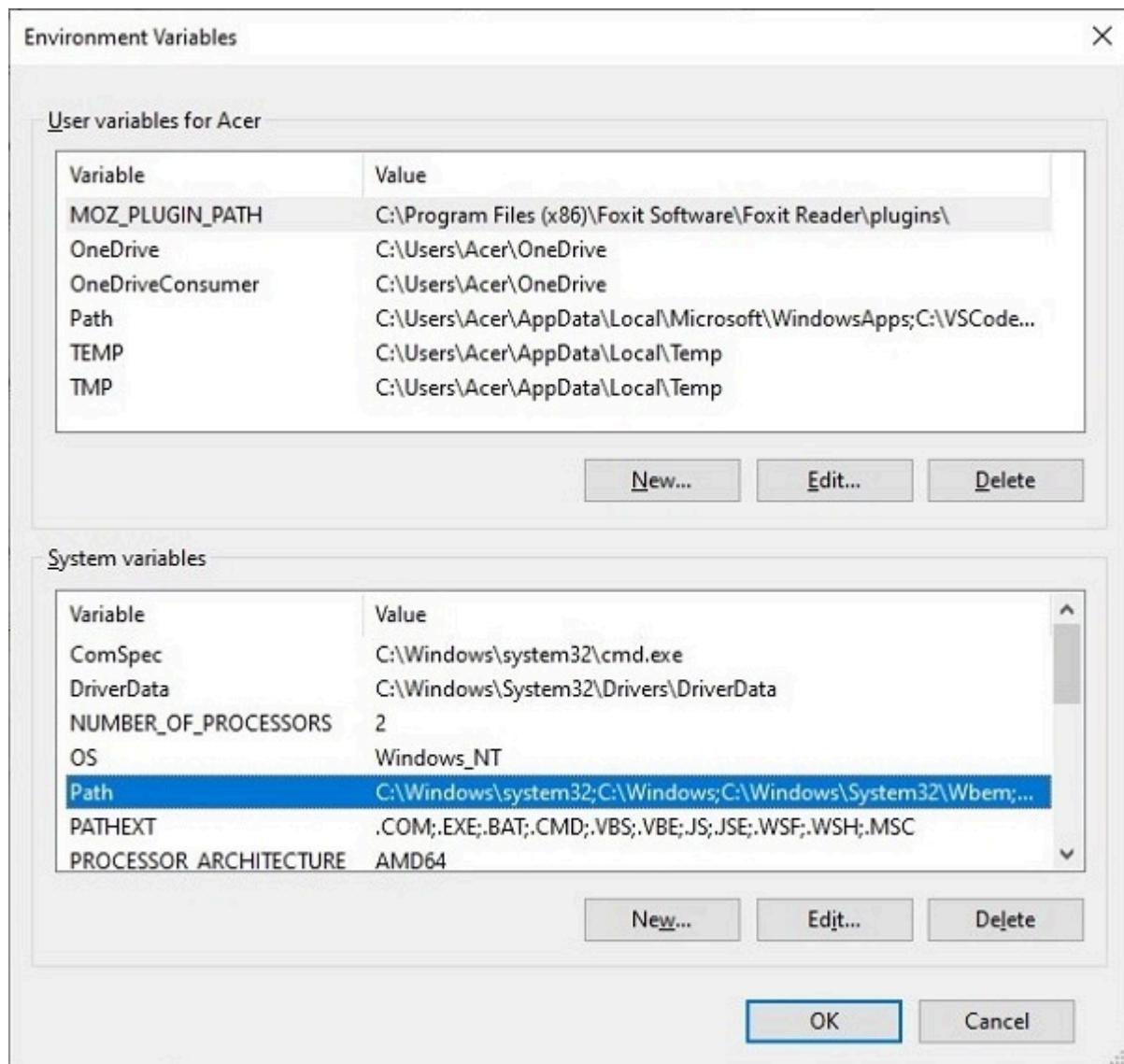
```
PCbuild\build.bat -e -d -p x64
```

Thus, in this chapter, you learned how to install Python from the pre-built binaries as well as from the source code.

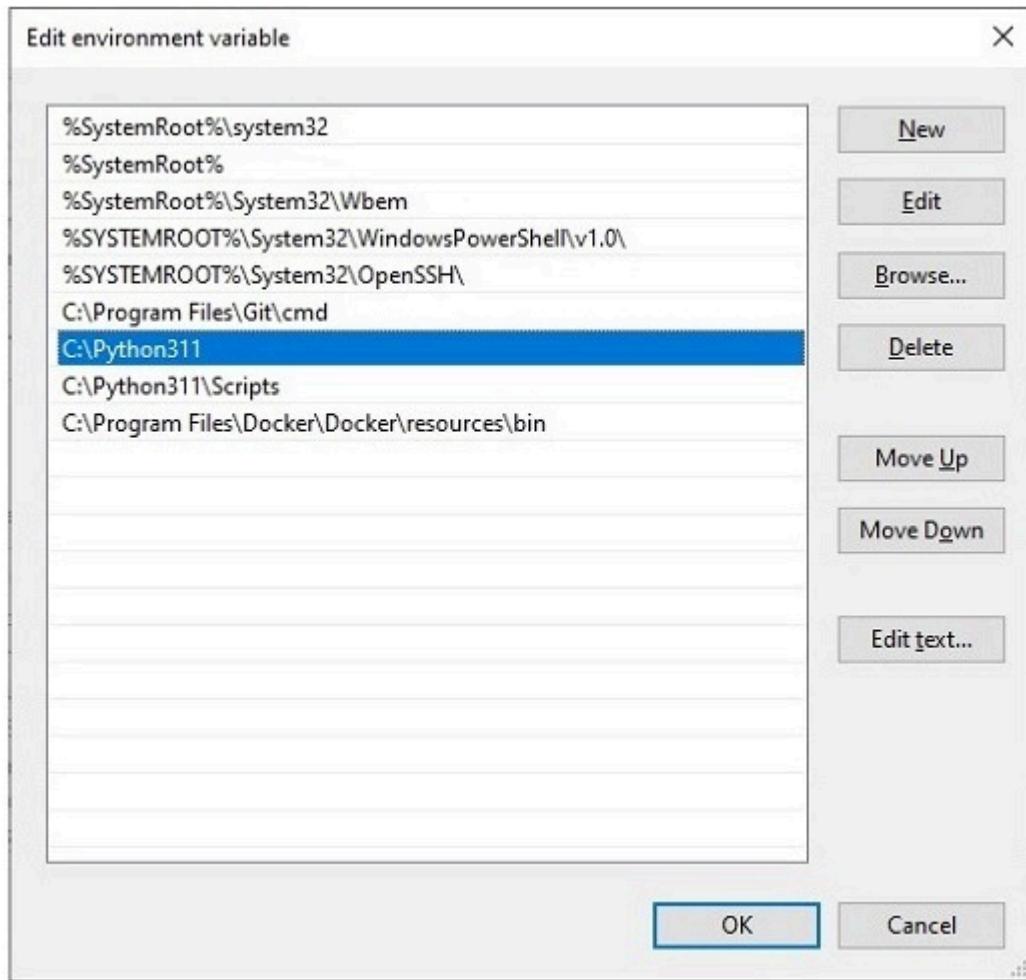
## Setting Up the PATH

When the Python software is installed, it should be accessible from anywhere in the file system. For this purpose, the **PATH** environment variable needs to be updated. A system PATH is a string consisting of folder names separated by **semicolon (;**). Whenever an executable program is invoked from the command line, the operating system searches for it in the folders listed in the PATH variable. We need to append Python's installation folder to the PATH string.

In case of Windows operating system, if you have enabled "add python.exe to system path" option on the first screen of the installation wizard, the path will be automatically updated. To do it manually, open Environment Variables section from Advanced System Settings.



Edit the Path variable, and add a new entry. Enter the name of the installation folder in which Python has been installed, and press OK.



To add the Python directory to the path for a particular session in Linux –

**In the bash shell (Linux)** – type **export PATH="\$PATH:/usr/bin/python3.11"** and press Enter.

## Python Command Line Options

We know that interactive Python interpreter can be invoked from the terminal simply by calling Python executable. Note that no additional parameters or options are needed to start the interactive session.

```
user@ubuntu20:~$ python3.11
Python 3.11.2 (main, Feb 8 2023, 14:49:24) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World")
Hello World
>>>
```

Python interpreter also responds to the following command line options –

**-c <command>**

Interpreters execute one or more statements in a string, separated by newlines (;) symbol.

```
user@ubuntu20:~$ python3 -c "a=2;b=3;print(a+b)"  
5
```

## -m <module-name>

Interpreter executes the contents of named module as the `__main__` module. Since the argument is a module name, you must not give a file extension (.py).

Consider the following example. Here, the `timeit` module in standard library has a command line interface. The `-s` option sets up the arguments for the module.

```
C:\Users\Acer>python -m timeit -s "text = 'sample string'; char = 'g'  
'char in text'"  
5000000 loops, best of 5: 49.4 nsec per loop
```

## <script>

Interpreter executes the Python code contained in script with .py extension, which must be a filesystem path (absolute or relative).

Assuming that a text file with the name `hello.py` contains `print ("Hello World")` statement is present in the current directory. The following command line usage of script option.

```
C:\Users\Acer>python hello.py  
Hello World
```

## ? Or -h or –help

This command line option prints a short description of all command line options and corresponding environment variables and exit.

## -V or --version

This command line option prints the Python version number

```
C:\Users\Acer>python -V  
Python 3.11.2  
C:\Users\Acer>python --version  
Python 3.11.2
```

## Python Environment Variables

The operating system uses path environment variable to search for any executable (not only Python executable). Python specific environment variables allow you to configure the behaviour of Python. For example, which folder locations to check to import a module. Normally Python interpreter searches for the module in the current folder. You can set one or more alternate folder locations.

Python environment variables may be set temporarily for the current session or may be persistently added in the System Properties as in case of path variable.

### PYTHONPATH

As mentioned above, if you want the interpreter should search for a module in other folders in addition to the current, one or more such folder locations are stored as PYTHONPATH variable.

First, save **hello.py** script in a folder different from Python's installation folder, let us say **c:\modulepath\hello.py**

To make the module available to the interpreter globally, set PYTHONPATH

```
C:\Users\Acer>set PYTHONPATH= c:\modulepath  
C:\Users\Acer>echo %PYTHONPATH%  
c:\modulepath
```

Now you can import the module even from any directory other than **c:\modulepath** directory.

```
>>> import hello  
Hello World  
>>>
```

### PYTHONHOME

Set this variable to change the location of the standard Python libraries. By default, the libraries are searched in **/usr/local/pythonversion** in case of Linux and **installdirectory\lib** in Windows. For example, **c:\python311\lib**.

### PYTHONSTARTUP

Usually, this variable is set to a Python script, which you intend to get automatically executed every time Python interpreter starts.

Let us create a simple script as follows and save it as **startup.py** in the Python installation folder –

```
print ("Example of Start up file")
print ("Hello World")
```

Now set the PYTHONSTARTUP variable and assign name of this file to it. After that start the Python interpreter. It shows the output of this script before you get the prompt.

```
F:\311_2>set PYTHONSTARTUP=startup.py
F:\311_2>echo %PYTHONSTARTUP%
startup.py
F:\311_2>python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Example of Start up file
Hello World
>>>
```

## PYTHONCASEOK

This environment is available for use only on Windows and MacOSX, not on Linux. It causes Python to ignore the cases in import statement.

## PYTHONVERBOSE

If this variable is set to a non-empty string it is equivalent to specifying python -v command. It results in printing a message, showing the place (filename or built-in module) each time a module is initialized. If set to an integer – say 2, it is equivalent to specifying -v two times. (python --v).

## PYTHONDONTWRITEBYTECODE

Normally, the imported modules are compiled to .pyc file. If this variable is set to a not null string, the .pyc files on the import of source modules are not created.

## PYTHONWARNINGS

Python's warning messages are redirected to the standard error stream, **sys.stderr**. This environment variable is equivalent to the python -W option. The following are allowed values of this variable –

- PYTHONWARNINGS=default # Warn once per call location

- PYTHONWARNINGS=error # Convert to exceptions
- PYTHONWARNINGS=always # Warn every time
- PYTHONWARNINGS=module # Warn once per calling module
- PYTHONWARNINGS=once # Warn once per Python process
- PYTHONWARNINGS=ignore # Never warn

## Python - Virtual Environment

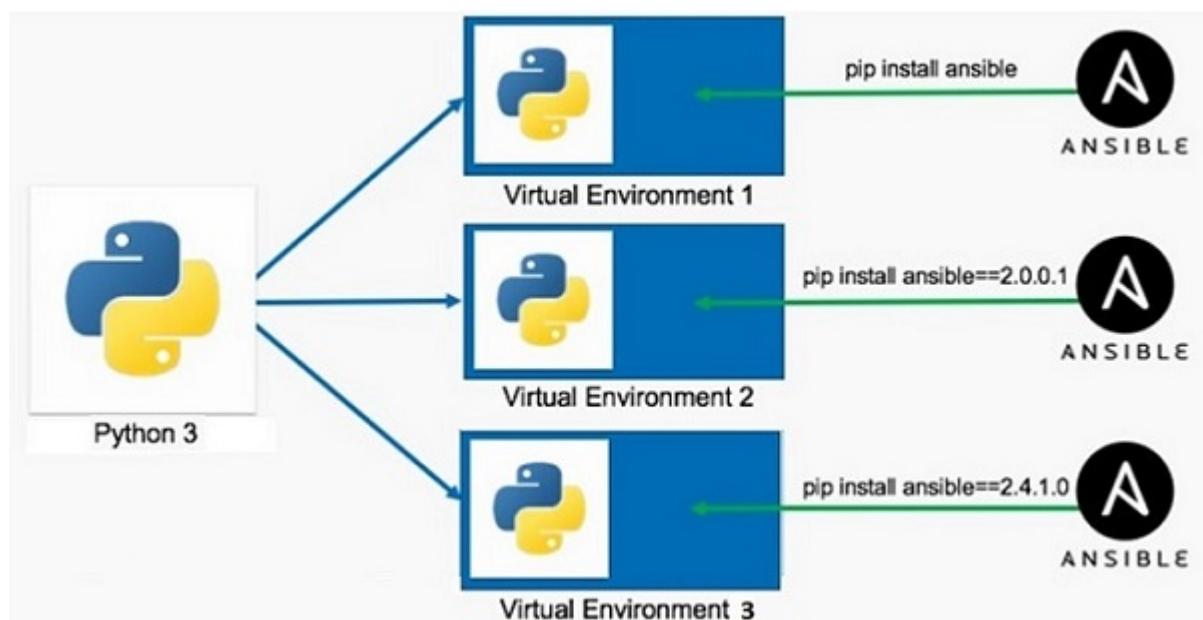
In this chapter, you will get to know what a virtual environment in Python is, how to create and use a virtual environment for building a Python application.

When you install Python software on your computer, it is available for use from anywhere in the filesystem. This is a system-wide installation.

While developing an application in Python, one or more libraries may be required to be installed using the pip utility (e.g., **pip3 install somelib**). Moreover, an application (let us say App1) may require a particular version of the library – say **somelib 1.0**. At the same time another Python application (for example App2) may require newer version of same library say **somelib 2.0**. Hence by installing a new version, the functionality of App1 may be compromised because of conflict between two different versions of same library.

This conflict can be avoided by providing two isolated environments of Python in the samemachine. These are called virtual environment. A virtual environment is a separatedirectory structure containing isolated installation having a local copy of Python interpreter, standard library and other modules.

The following figure shows the purpose of advantage of using virtual environment. Using the global Python installation, more than one virtual environments are created, each having different version of the same library, so that conflict is avoided.



This functionality is supported by **venv** module in standard Python distribution. Use following commands to create a new virtual environment.

```
C:\Users\Acer>md\pythonapp
C:\Users\Acer>cd\pythonapp
C:\pythonapp>python -m venv myenv
```

Here, **myenv** is the folder in which a new Python virtual environment will be created showing following directory structure –

```
Directory of C:\pythonapp\myenv
22-02-2023 09:53 <DIR> .
22-02-2023 09:53 <DIR> ..
22-02-2023 09:53 <DIR> Include
22-02-2023 09:53 <DIR> Lib
22-02-2023 09:53 77 pyvenv.cfg
22-02-2023 09:53 <DIR> Scripts
```

The utilities for activating and deactivating the virtual environment as well as the local copy of Python interpreter will be placed in the scripts folder.

```
Directory of C:\pythonapp\myenv\scripts
22-02-2023 09:53 <DIR> .
22-02-2023 09:53 <DIR> ..
22-02-2023 09:53 2,063 activate
22-02-2023 09:53 992 activate.bat
22-02-2023 09:53 19,611 Activate.ps1
22-02-2023 09:53 393 deactivate.bat
22-02-2023 09:53 106,349 pip.exe
22-02-2023 09:53 106,349 pip3.10.exe
22-02-2023 09:53 106,349 pip3.exe
22-02-2023 09:53 242,408 python.exe
22-02-2023 09:53 232,688 pythonw.exe
```

To enable this new virtual environment, execute **activate.bat** in Scripts folder.

```
C:\pythonapp>myenv\scripts\activate
(myenv) C:\pythonapp>
```

Note the name of the virtual environment in the parentheses. The Scripts folder contains a local copy of Python interpreter. You can start a Python session in this virtual environment.

To confirm whether this Python session is in virtual environment check the **sys.path**.

```
(myvenv) C:\pythonapp>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec 6 2021, 19:10:37) [MSC v.1929
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
[ '', 'C:\\Python310\\python310.zip', 'C:\\Python310\\DLLs',
'C:\\Python310\\lib', 'C:\\Python310', 'C:\\pythonapp\\myvenv',
'C:\\pythonapp\\myvenv\\lib\\site-packages' ]
>>>
```

The scripts folder of this virtual environment also contains pip utilities. If you install a package from PyPI, that package will be active only in current virtual environment. To deactivate this environment, run **deactivate.bat**.

## Python - Basic Syntax

In Python, the term syntax refers to the rules of forming a statement or expression. Python language is known for its clean and simple syntax. It also has a limited set of keywords and simpler punctuation rules as compared to other languages. In this chapter, let us understand about basic syntax of Python.

A Python program comprises of predefined keywords and identifiers representing functions, classes, modules etc. Python has clearly defined rules for forming identifiers, writing statements and comments in Python source code.

## Python Keywords

A predefined set of keywords is the most important aspect of any programming language. These keywords are reserved words. They have a predefined meaning, they must be used only for its predefined purpose and as per the predefined rules of syntax. Programming logic is encoded with these keywords.

As of Python 3.11 version, there are 35 (Thirty Five) keywords in Python. To obtain the list of Python keywords, enter the following help command in Python shell.

```
>>> help("keywords")
```

Here is a list of the Python keywords. Enter any keyword to get more help.

1. False	10. class	19. from	28. or
----------	-----------	----------	--------

2. None	11. continue	20. global	29. pass
3. True	12. def	21. if	30. raise
4. and	13. del	22. import	31. return
5. as	14. elif	23. in	32. try
6. assert	15. else	24. is	33. while
7. async	16. except	25. lambda	34. with
8. await	17. finally	26. nonlocal	35. yield
9. break	18. for	27. not	

All the keywords are alphabetic and all (except False, None and True) are in lowercase.  
The list of keywords is also given by kwlist property defined in keyword module

```
>>> import keyword
>>> keyword.kwlist
['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']
```

How to verify if any word is a keyword or not? Most Python IDEs provide coloured syntax highlighting feature, where keywords are represented in a specific colour.

Shown below is a Python code in VS Code. Keywords (if and else), identifiers and constants appear in distinct colour scheme.

The screenshot shows the Visual Studio Code interface. The left sidebar has icons for Explorer, Open Editors, TestFolder, and others. The 'OPEN EDITORS' section shows 'hello.py' is open. The 'TESTFOLDER' section shows '.vscode' and 'db.sqlite3'. The main editor area contains the following Python code:

```

1 amount = 1000
2
3 if amount>500:
4     rate = 10
5 else:
6     rate = 5
7
8 interest = amount*rate/100
9 print ("Interest = ", interest)

```

The keyword module also has a `iskeyword()` function. It returns True for a valid keyword, False otherwise.

```

>>> import keyword
>>> keyword.iskeyword("else")
True
>>> keyword.iskeyword("Hello")
False

```

Python keywords can be broadly classified in following categories –

<b>Value Keywords</b>	True, False, None
<b>Operator Keywords</b>	and, or, not, in, is
<b>Conditional Flow Keywords</b>	if, elif, else
<b>Keywords for loop control</b>	for, while, break, continue
<b>Structure Keywords</b>	def, class, with, pass, lambda
<b>Keywords for returning</b>	return, yield
<b>Import Keywords</b>	import, from, as
<b>Keywords about ExceptionHandling</b>	try, except, raise, finally, assert
<b>Keywords for Asynchronous Programming</b>	async, await

## Variable Scope Keywords

del, global, nonlocal

We shall learn about the usage of each of these keywords as we go along in this tutorial.

## Python Identifiers

Various elements in a Python program, other than keywords, are called identifiers. An identifier is a user-given name to variables, functions, classes, modules, packages etc. in the source code. Python has laid down certain rules to form an identifier. These rules are –

- An identifier should start with either an alphabet (lower or upper case) or underscore (\_). More than one alpha-numeric characters or underscores may follow.
- Use of any keyword as n identifier is not allowed, as keywords have a predefined meaning.
- Conventionally, name of class begins with uppercase alphabet. Other elements like variable or function start with lowercase alphabet.
- As per another Python convention, single underscore in the beginning of a variable name is used to indicate a private variable.
- Use two underscores in beginning of identifier indicates that the variable is strongly private.
- Two leading and trailing underscores are used in language itself for special purpose. For example, \_\_add\_\_, \_\_init\_\_

According to the above rules, here are some valid identifiers –

- Student
- score
- aTotal
- sum\_age
- \_\_count
- TotalValue
- price1
- cost\_of\_item
- \_\_init\_\_

Some invalid formations of identifiers are also given below –

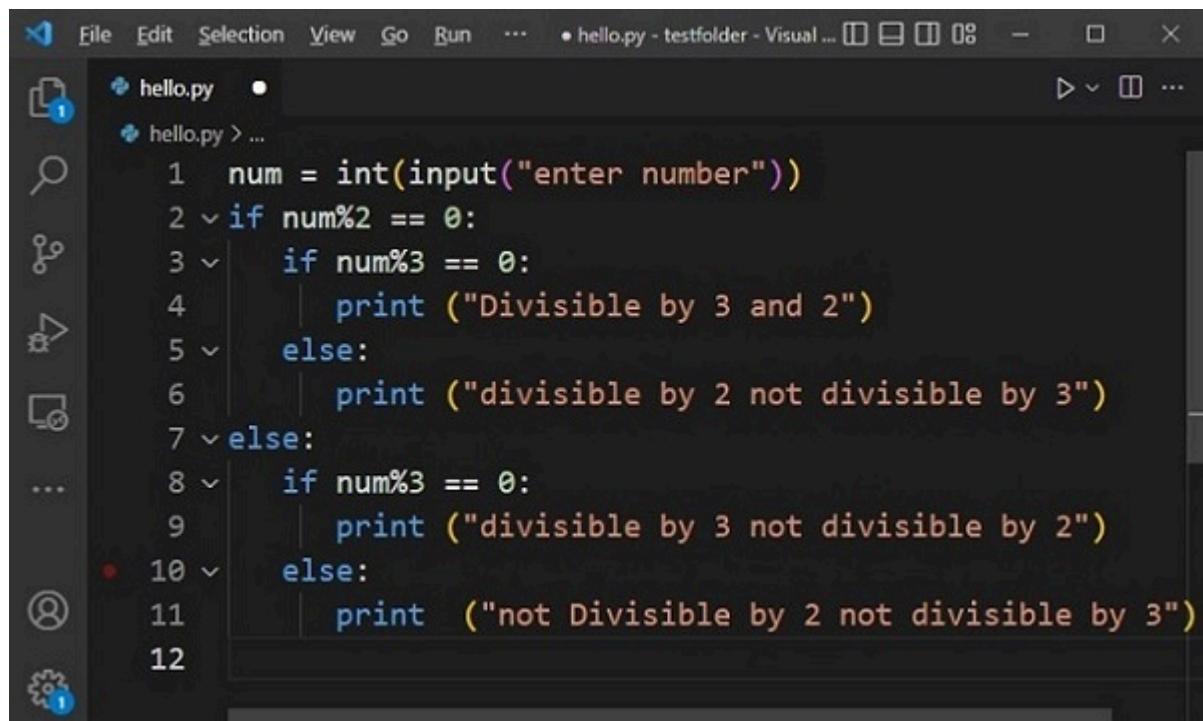
- 1001
- Name of student
- price-1
- ft.in

It may be noted that identifiers are case sensitive. As a result, Name and name are two different identifiers.

## Python Indents

Use of indents in code is one of the important features of Python's syntax. Often in a program, you might require grouping more than one statements together as a block. For example, in case of more than one statements if a condition is true/false. Different programming languages have different methods to mark the scope and extent of group of statements in constructs like class, function, conditional and loop. C, C++, Java etc. make use of curly brackets to mark the block. Python uses uniform indentation to mark block of statements, thereby it increases the readability of the code.

To mark the beginning of a block, type the ":" symbol and press Enter. Any Python-aware editor (like IDLE, or VS Code) goes to the next line leaving additional whitespace (called indent). Subsequent statements in the block follow same level of indent. To signal end of the block, the whitespace is dedented by pressing the backspace key. The following example illustrates the use of indents in Python:



```
File Edit Selection View Go Run ... • hello.py - testfolder - Visual ... □ □ □ □ □ - □ ×
hello.py
hello.py > ...
1 num = int(input("enter number"))
2 if num%2 == 0:
3     if num%3 == 0:
4         print ("Divisible by 3 and 2")
5     else:
6         print ("divisible by 2 not divisible by 3")
7 else:
8     if num%3 == 0:
9         print ("divisible by 3 not divisible by 2")
10    else:
11        print ("not Divisible by 2 not divisible by 3")
12
```

At this juncture, you may not understand how the code works. But don't worry. Just see how indent level increases after colon symbol.

## Python Statements

A statement in Python is any instruction that the Python interpreter can execute. A statement comprises of one or more keywords, operators, identifiers, a : symbol to mark beginning of block, or backslash \ as continuation character.

The statement may be a simple assignment statement such as amount = 1000 or it may be a compound statement with multiple statements grouped together in uniformly indented block, as in conditional or looping constructs.

You can enter a statement in front of the Python prompt of the interactive shell, or in the editor window. Usually, text terminated by Enter key (called newline character) is recognized as a statement by Python interpreter. Hence, each line in the editor is a statement, unless it starts with the comment character (#).

&lt;/&gt;

Open Compiler

```
print ("My first program")
price = 100
qty = 5
ttl = price*qty
print ("Total = ", ttl)
```

Each line in the above code is a statement. Occasionally, a Python statement may spill over multiple lines. To do so, use backslash (\) as continuation character. A long string can be conveniently broken in multiple lines as shown below –

&lt;/&gt;

Open Compiler

```
name = "Ravi"
string = "Hello {} \
Welcome to Python Tutorial \
from TutorialsPoint".format(name)
print (string)
```

The string (with an embedded string variable name) spreads over multiple lines for better readability. The output will be –

Hello Ravi Welcome to Python Tutorial from TutorialsPoint

The continuation character also helps in writing a long arithmetic expression in a more readable manner.

For example, the equation  $\frac{(a+b) \times (c-d)}{(a-b) \times (c+d)}$  is coded in Python as follows –

```
a=10
b=5
c=5
d=10
expr = (a+b)*(c-d)/ \
        (a-b)*(c+d)
print(expr)
```

The use of back-slash symbol (\) is not necessary if items in a list, tuple or dictionary object spill over multiple lines.

```
Subjects = ["English", "French", "Sanskrit",
           "Physics", "Maths",
           "Computer Sci", "History"]
```

Python also allows use of semicolon to put more than one statements in a single line in the editor. Look at the following examples –

```
a=10; b=5; c=5; d=10
if a>10: b=20; c=50
```

## Python - Variables

In this chapter, you will learn what are variables in Python and how to use them.

Data items belonging to different data types are stored in computer's memory. Computer's memory locations are having a number or address, internally represented in binary form. Data is also stored in binary form as the computer works on the principle of binary representation. In the following diagram, a string **May** and a number **18** is shown as stored in memory locations.

# Memory

100	101	102	103	104
200	201	202	203	204
300	301	302	303	304
400	401	402	403	404
500	501	502	503	504

May  
18

If you know the assembly language, you will convert these data items and the memory address, and give a machine language instruction. However, it is not easy for everybody. Language translator such as Python interpreter performs this type of conversion. It stores the object in a randomly chosen memory location. Python's built-in **`id()`** function returns the address where the object is stored.

```
>>> "May"
>>> id("May")
2167264641264
>>> 18
18
>>> id(18)
140714055169352
```

Once the data is stored in the memory, it should be accessed repeatedly for performing a certain process. Obviously, fetching the data from its ID is cumbersome. High level languages like Python make it possible to give a suitable alias or a label to refer to the memory location.

In the above example, let us label the location of May as month, and location in which 18 is stored as age. Python uses the assignment operator (=) to bind an object with the label.

```
>>> month="May"  
>>> age=18
```

The data object (May) and its name (month) have the same id(). The id() of 18 and age are also same.

```
>>> id(month)  
2167264641264  
>>> id(age)  
140714055169352
```

The label is an identifier. It is usually called as a variable. A Python variable is a symbolic name that is a reference or pointer to an object.

## Naming Convention

Name of the variable is user specified, and is formed by following the rules of forming an identifier.

- Name of Python variable should start with either an alphabet (lower or upper case) or underscore (\_). More than one alpha-numeric characters or underscores may follow.
- Use of any keyword as Python variable is not allowed, as keywords have a predefined meaning.
- Name of a variable in Python is case sensitive. As a result, age and Age cannot be used interchangeably.
- You should choose the name of variable that is mnemonic, such that it indicates the purpose. It should not be very short, but not vary lengthy either.

If the name of variable contains multiple words, we should use these naming patterns –

- **Camel case** – First letter is a lowercase, but first letter of each subsequent word is in uppercase. For example: kmPerHour, pricePerLitre
- **Pascal case** – First letter of each word is in uppercase. For example: KmPerHour, PricePerLitre
- **Snake case** – Use single underscore (\_) character to separate words. For example: km\_per\_hour, price\_per\_litre

Once you use a variable to identify a data object, it can be used repeatedly without its id() value. Here, we have a variables height and width of a rectangle. We can compute the

area and perimeter with these variables.

```
>>> width=10
>>> height=20
>>> area=width*height
>>> area
200
>>> perimeter=2*(width+height)
>>> perimeter
60
```

Use of variables is especially advantageous when writing scripts or programs. Following script also uses the above variables.

</>

Open Compiler

```
#!/usr/bin/python3.11
width = 10
height = 20
area = width*height
perimeter = 2*(width+height)
print ("Area = ", area)
print ("Perimeter = ", perimeter)
```

Save the above script with .py extension and execute from command-line. The result would be –

```
Area = 200
Perimeter = 60
```

## Assignment Statement

In languages such as C/C++ and Java, one needs to declare the variable and its type before assigning it any value. Such prior declaration of variable is not required in Python.

Python uses = symbol as the assignment operator. Name of the variable identifier appears on the left of = symbol. The expression on its right is evaluated and the value is assigned to the variable. Following are the examples of assignment statements

```
>>> counter = 10
>>> counter = 10 # integer assignment
```

```
>>> price = 25.50 # float assignment
>>> city = "Hyderabad" # String assignment
>>> subjects = ["Physics", "Maths", "English"] # List assignment
>>> mark_list = {"Rohit":50, "Kiran":60, "Lata":70} # dictionary assignment
```

Python's built-in **print()** function displays the value of one or more variables.

```
>>> print(counter, price, city)
10 25.5 Hyderabad
>>> print(subjects)
['Physics', 'Maths', 'English']
>>> print(mark_list)
{'Rohit': 50, 'Kiran': 60, 'Lata': 70}
```

Value of any expression on the right of = symbol is assigned to the variable on left.

```
>>> x = 5
>>> y = 10
>>> z = x+y
```

However, the expression on the left and variable on the right of = operator is not allowed.

```
>>> x = 5
>>> y = 10
>>> x+y=z
File "<stdin>", line 1
    x+y=z
    ^
    ^
```

SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?

Though  $z=x+y$  and  $x+y=z$  are equivalent in Mathematics, it is not so here. It's because = is an equation symbol, while in Python it is an assignment operator.

## Multiple Assignments

In Python, you can initialize more than one variables in a single statement. In the following case, three variables have same value.

```
>>> a=10
>>> b=10
```

```
>>> c=10
```

Instead of separate assignments, you can do it in a single assignment statement as follows –

```
>>> a=b=c=10
>>> print (a,b,c)
10 10 10
```

In the following case, we have three variables with different values.

```
>>> a=10
>>> b=20
>>> c=30
```

These separate assignment statements can be combined in one. You need to give comma separated variable names on left, and comma separated values on the right of = operator.

```
>>> a,b,c = 10,20,30
>>> print (a,b,c)
10 20 30
```

The concept of variable works differently in Python than in C/C++.

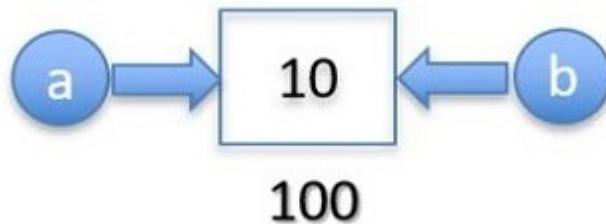
In C/C++, a variable is a named memory location. If a=10 and also b=10, both are two different memory locations. Let us assume their memory address is 100 and 200 respectively.



If a different value is assigned to "a" – say 50, 10 in the address 100 is overwritten.



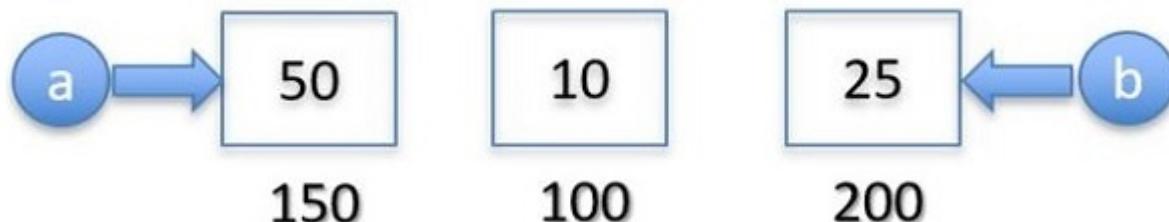
A Python variable refers to the object and not the memory location. An object is stored in memory only once. Multiple variables are really the multiple labels to the same object.



The statement `a=50` creates a new **int** object 50 in the memory at some other location, leaving the object 10 referred by "b".



Further, if you assign some other value to b, the object 10 remains unreferred.



Python's garbage collector mechanism releases the memory occupied by any unreferred object.

Python's identity operator **is** returns True if both the operands have same `id()` value.

```

>>> a=b=10
>>> a is b
True
>>> id(a), id(b)
(140731955278920, 140731955278920)
  
```

## Python - Data Types

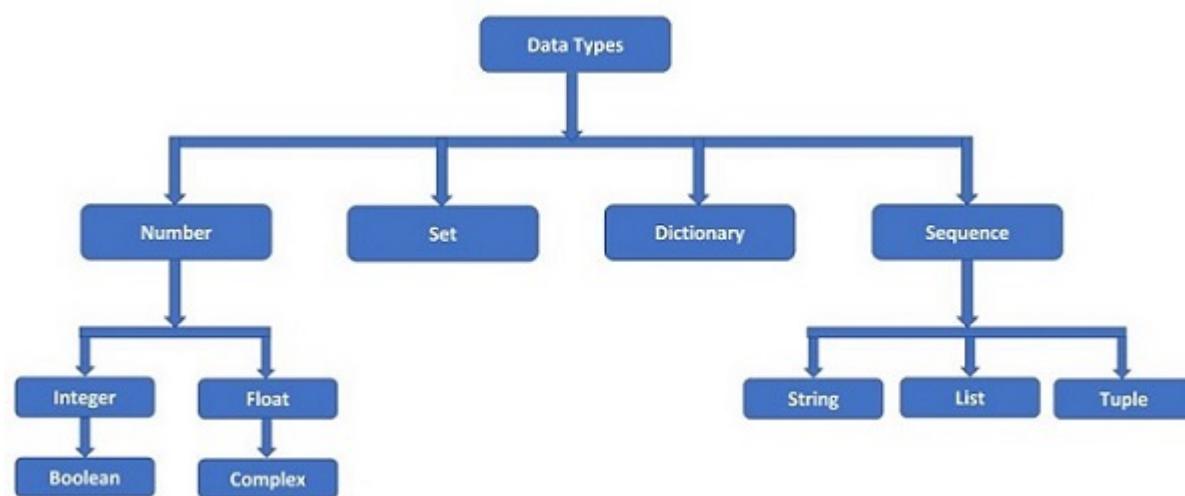
Computer is a data processing device. Computer stores the data in its memory and processes it as per the given program. Data is a representation of facts about a certain object.

Some examples of data –

- **Data of students** – name, gender, class, marks, age, fee etc.
- **Data of books in library** – title, author, publisher, price, pages, year of publication etc.
- **Data of employees in an office** – name, designation, salary, department, branch, etc.

Data type represents a kind of value and determines what operations can be done on it. Numeric, non-numeric and Boolean (true/false) data are the most obvious data types. However, each programming language has its own classification largely reflecting its programming philosophy.

Python identifies the data by different data types as per the following diagram –



Python's data model defines four main data types. They are Number, Sequence, Set and Dictionary (also called Mapping)

## Number Type

Any data item having a numeric value is a number. There are Four standard number data types in Python. They are integer, floating point, Boolean and Complex. Each of them have built-in classes in Python library, called **int**, **float**, **bool** and **complex** respectively.

In Python, a number is an object of its corresponding class. For example, an integer number 123 is an object of **int** class. Similarly, 9.99 is a floating point number, which is an object of **float** class.

Python's standard library has a built-in function **type()**, which returns the class of the given object. Here, it is used to check the type of an integer and floating point number.

```
>>> type(123)
<class 'int'>
>>> type(9.99)
<class 'float'>
```

The fractional component of a float number can also be represented in **scientific** format. A number -0.000123 is equivalent to its scientific notation 1.23E-4 (or 1.23e-4).

A complex number is made up of two parts – **real** and **imaginary**. They are separated by '+' or '-' signs. The imaginary part is suffixed by 'j' which is the imaginary number. The square root of -1 ( $\sqrt{-1}$ ), is defined as imaginary number. Complex number in Python is represented as  $x+yj$ , where x is the real part, and y is the imaginary part. So,  $5+6j$  is a complex number.

```
>>> type(5+6j)
<class 'complex'>
```

A Boolean number has only two possible values, as represented by the keywords, **True** and **False**. They correspond to integer 1 and 0 respectively.

```
>>> type (True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

With Python's arithmetic operators you can perform operations such as addition, subtraction etc.

## Sequence Types

Sequence is a collection data type. It is an ordered collection of items. Items in the sequence have a positional index starting with 0. It is conceptually similar to an array in C or C++. There are three sequence types defined in Python. String, List and Tuple.

## Strings in Python

A string is a sequence of one or more Unicode characters, enclosed in single, double or triple quotation marks (also called inverted commas). As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.

```
>>> 'Welcome To TutorialsPoint'
'Welcome To TutorialsPoint'
```

```
>>> "Welcome To TutorialsPoint"
'Welcome To TutorialsPoint'
>>> '''Welcome To TutorialsPoint'''
'Welcome To TutorialsPoint'
```

A string in Python is an object of **str** class. It can be verified with **type()** function.

```
>>> type("Welcome To TutorialsPoint")
<class 'str'>
```

You want to embed some text in double quotes as a part of string, the string itself should be put in single quotes. To embed a single quoted text, string should be written in double quotes.

```
>>> 'Welcome to "Python Tutorial" from TutorialsPoint'
'Welcome to "Python Tutorial" from TutorialsPoint'
>>> "Welcome to 'Python Tutorial' from TutorialsPoint"
"Welcome to 'Python Tutorial' from TutorialsPoint"
```

Since a string is a sequence, each character in it is having a positional index starting from 0. To form a string with triple quotes, you may use triple single quotes, or triple double quotes – both versions are similar.

```
>>> '''Welcome To TutorialsPoint'''
'Welcome To TutorialsPoint'
>>> """Welcome To TutorialsPoint"""
'Welcome To TutorialsPoint'
```

Triple quoted string is useful to form a multi-line string.

```
>>> '''
... Welcome To
... Python Tutorial
... from TutorialsPoint
...
...
'\nWelcome To\nPython Tutorial \nfrom TutorialsPoint\n'
```

A string is a non-numeric data type. Obviously, we cannot perform arithmetic operations on it. However, operations such as **slicing** and **concatenation** can be done. Python's str class defines a number of useful methods for string processing. We shall learn these methods in the subsequent chapter on Strings.

## List in Python

In Python, List is an ordered collection of any type of data items. Data items are separated by comma (,) symbol and enclosed in square brackets ([]). A list is also a sequence, hence.

each item in the list has an index referring to its position in the collection. The index starts from 0.

The list in Python appears to be similar to array in C or C++. However, there is an important difference between the two. In C/C++, array is a homogenous collection of data of similar types. Items in the Python list may be of different types.

```
>>> [2023, "Python", 3.11, 5+6j, 1.23E-4]
```

A list in Python is an object of **list** class. We can check it with `type()` function.

```
>>> type([2023, "Python", 3.11, 5+6j, 1.23E-4])
<class 'list'>
```

As mentioned, an item in the list may be of any data type. It means that a list object can also be an item in another list. In that case, it becomes a nested list.

```
>>> [[ 'One', 'Two', 'Three'], [1,2,3], [1.0, 2.0, 3.0]]
```

A list item may be a tuple, dictionary, set or object of user defined class also.

List being a sequence, it supports slicing and concatenation operations as in case of string. With the methods/functions available in Python's built-in list class, we can add, delete or update items, and sort or rearrange the items in the desired order. We shall study these aspects in a subsequent chapter.

## Tuples in Python

In Python, a Tuple is an ordered collection of any type of data items. Data items are separated by comma (,) symbol and enclosed in parentheses or round brackets (). A tuple is also a sequence, hence each item in the tuple has an index referring to its position in the collection. The index starts from 0.

```
>>> (2023, "Python", 3.11, 5+6j, 1.23E-4)
```

In Python, a tuple is an object of **tuple** class. We can check it with the `type()` function.

```
>>> type((2023, "Python", 3.11, 5+6j, 1.23E-4))
<class 'tuple'>
```

As in case of a list, an item in the tuple may also be a list, a tuple itself or an object of any other Python class.

```
>>> ([ 'One', 'Two', 'Three'], 1,2.0,3, (1.0, 2.0, 3.0))
```

To form a tuple, use of parentheses is optional. Data items separated by comma without any enclosing symbols are treated as a tuple by default.

```
>>> 2023, "Python", 3.11, 5+6j, 1.23E-4
(2023, 'Python', 3.11, (5+6j), 0.000123)
```

The two sequence types list and tuple appear to be similar except the use of delimiters, list uses square brackets ([]) while tuple uses parentheses. However, there is one major

difference between list and tuple. List is mutable object, whereas tuple is **immutable**. An object is immutable means once it is stored in the memory, it cannot be changed.

Let us try to understand the mutability concept. We have a list and tuple object with same data items.

```
>>> l1=[1,2,3]
>>> t1=(1,2,3)
```

Both are sequences, hence each item in both has an index. Item at index number 1 in both is 2.

```
>>> l1[1]
2
>>> t1[1]
2
```

Let us try to change the value of item index number 1 from 2 to 20 in list as well as tuple.

```
>>> l1[1]
2
>>> t1[1]
2
>>> l1[1]=20
>>> l1
[1, 20, 3]
```

```
>>> t1[1]=20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The error message '**tuple' object does not support item assignment**' tells you that a tuple object cannot be modified once it is formed. This is called an immutable object.

Immutability of tuple also means that Python's tuple class doesn't have the functionality to add, delete or sort items in a tuple. However, since it is a sequence, we can perform slicing and concatenation.

## Dictionary Type

Python's dictionary is example of **mapping** type. A mapping object 'maps' value of one object with another. In a language dictionary we have pairs of word and corresponding meaning. Two parts of pair are key (word) and value (meaning). Similarly, Python dictionary is also a collection of **key:value** pairs. The pairs are separated by comma and put inside curly brackets {}. To establish mapping between key and value, the semicolon':' symbol is put between the two.

```
>>> {1:'one', 2:'two', 3:'three'}
```

Each key in a dictionary must be unique, and should be a number, string or tuple. The value object may be of any type, and may be mapped with more than one keys (they need not be unique)

In Python, dictionary is an object of the built-in **dict** class. We can check it with the `type()` function.

```
>>> type({1:'one', 2:'two', 3:'three'})
<class 'dict'>
```

Python's dictionary is not a sequence. It is a collection of items but each item (key:value pair) is not identified by positional index as in string, list or tuple. Hence, slicing operation cannot be done on a dictionary. Dictionary is a mutable object, so it is possible to perform add, modify or delete actions with corresponding functionality defined in dict class. These operations will be explained in a subsequent chapter.

## Set Type

Set is a Python implementation of set as defined in Mathematics. A set in Python is a collection, but is not an indexed or ordered collection as string, list or tuple. An object

cannot appear more than once in a set, whereas in List and Tuple, same object can appear more than once.

Comma separated items in a set are put inside curly brackets or braces. Items in the set collection may be of different data types.

```
>>> {2023, "Python", 3.11, 5+6j, 1.23E-4}
{(5+6j), 3.11, 0.000123, 'Python', 2023}
```

Note that items in the set collection may not follow the same order in which they are entered. The position of items is optimized by Python to perform operations over set as defined in mathematics.

Python's Set is an object of built-in **set** class, as can be checked with the `type()` function.

```
>>> type({2023, "Python", 3.11, 5+6j, 1.23E-4})
<class 'set'>
```

A set can store only immutable objects such as number (int, float, complex or bool), string or tuple. If you try to put a list or a dictionary in the set collection, Python raises a **TypeError**.

```
>>> {[ 'One', 'Two', 'Three'], 1,2,3, (1.0, 2.0, 3.0)}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

**Hashing** is a mechanism in computer science which enables quicker searching of objects in computer's memory. **Only immutable objects are hashable.**

Even if a set doesn't allow mutable items, the set itself is mutable. Hence, add/delete/update operations are permitted on a set object, using the methods in built-in set class. Python also has a set of operators to perform set manipulation. The methods and operators are explained in a latter chapter

## Python - Type Casting

In manufacturing, casting is the process of pouring a liquefied or molten metal into a mold, and letting it cool to obtain the desired shape. In programming, casting refers to converting an object of one type into another. Here, we shall learn about type casting in Python.

In Python there are different data types, such as numbers, sequences, mappings etc. There may be a situation where, you have the available data of one type but you want to

use it in another form. For example, the user has input a string but you want to use it as a number. Python's type casting mechanism let you do that.

## Implicit Casting in Python

Casting is of two types – **implicit** and **explicit**.

When any language compiler/interpreter automatically converts object of one type into other, it is called implicit casting. Python is a strongly typed language. It doesn't allow automatic type conversion between unrelated data types. For example, a string cannot be converted to any number type. However, an integer can be cast into a float. Other languages such as JavaScript is a weakly typed language, where an integer is coerced into a string for concatenation.

Note that memory requirement of each type is different. For example, an integer object in Python occupies 4 bytes of memory, while a float object needs 8 bytes because of its fractional part. Hence, Python interpreter doesn't automatically convert a float to int, because it will result in loss of data. On the other hand, int can be easily converted into float by setting its fractional part to 0.

Implicit int to float casting takes place when any arithmetic operation one int and float operands is done.

We have an integer and one float variable

```
>>> a=10 # int object  
>>> b=10.5 # float object
```

To perform their addition, 10 – the integer object is upgraded to 10.0. It is a float, but equivalent to its earlier numeric value. Now we can perform addition of two floats.

```
>>> c=a+b  
>>> print (c)  
20.5
```

In implicit type casting, the object with lesser byte size is upgraded to match the byte size of other object in the operation. For example, a Boolean object is first upgraded to int and then to float, before the addition with a floating point object. In the following example, we try to add a Boolean object in a float.

```
>>> a=True  
>>> b=10.5  
>>> c=a+b
```

```
>>> print (c)
11.5
```

Note that True is equal to 1, and False is equal to 0.

Although automatic or implicit casting is limited to **int** to **float** conversion, you can use Python's built-in functions to perform the explicit conversions such as string to integer.

## int() Function

Python's built-in int() function converts an integer literal to an integer object, a float to integer, and a string to integer if the string itself has a valid integer literal representation.

Using int() with an int object as argument is equivalent to declaring an **int** object directly.

```
>>> a = int(10)
>>> a
10
```

is same as –

```
>>> a = 10
>>> a
10
>>> type(a)
<class 'int'>
```

If the argument to int() function is a float object or floating point expression, it returns an int object. For example –

```
>>> a = int(10.5) #converts a float object to int
>>> a
10
>>> a = int(2*3.14) #expression results float, is converted to int
>>> a
6
>>> type(a)
<class 'int'>
```

The int() function also returns integer 1 if a Boolean object is given as argument.

```
>>> a=int(True)
>>> a
```

```
1
>>> type(a)
<class 'int'>
```

## String to Integer

The int() function returns an integer from a string object, only if it contains a valid integer representation.

```
>>> a = int("100")
>>> a
100
>>> type(a)
<class 'int'>
>>> a = ("10"+"01")
>>> a = int("10"+"01")
>>> a
1001
>>> type(a)
<class 'int'>
```

However, if the string contains a non-integer representation, Python raises ValueError.

```
>>> a = int("10.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '10.5'
>>> a = int("Hello World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello World'
```

The int() function also returns integer from binary, octal and hexa-decimal string. For this, the function needs a base parameter which must be 2, 8 or 16 respectively. The string should have a valid binary/octal/Hexa-decimal representation.

## Binary String to Integer

The string should be made up of 1 and 0 only, and the base should be 2.

```
>>> a = int("110011", 2)
>>> a
51
```

The Decimal equivalent of binary number 110011 is 51.

## Octal String to Integer

The string should only contain 0 to 7 digits, and the base should be 8.

```
>>> a = int("20", 8)
>>> a
16
```

The Decimal equivalent of octal 20 is 16.

## Hexa-Decimal String to Integer

The string should contain only the Hexadecimal symbols i.e., 0-9 and A, B, C, D, E or F. Base should be 16.

```
>>> a = int("2A9", 16)
>>> a
681
```

Decimal equivalent of Hexadecimal 2A9 is 681.

You can easily verify these conversions with calculator app in Windows, Ubuntu or Smartphones.

## float() Function

float() is a built-in function in Python. It returns a float object if the argument is a float literal, integer or a string with valid floating point representation.

Using float() with an float object as argument is equivalent to declaring a float object directly

```
>>> a = float(9.99)
>>> a
9.99
>>> type(a)
<class 'float'>
```

is same as –

```
>>> a = 9.99
>>> a
```

```
9.99
```

```
>>> type(a)
<class 'float'>
```

If the argument to float() function is an integer, the returned value is a floating point with fractional part set to 0.

```
>>> a = float(100)
>>> a
100.0
>>> type(a)
<class 'float'>
```

The float() function returns float object from a string, if the string contains a valid floating point number, otherwise ValueError is raised.

```
>>> a = float("9.99")
>>> a
9.99
>>> type(a)
<class 'float'>
>>> a = float("1,234.50")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1,234.50'
```

The reason of ValueError here is the presence of comma in the string.

For the purpose of string to float conversion, the scientific notation of floating point is also considered valid.

```
>>> a = float("1.00E4")
>>> a
10000.0
>>> type(a)
<class 'float'>
>>> a = float("1.00E-4")
>>> a
0.0001
>>> type(a)
<class 'float'>
```

## str() Function

We saw how a Python obtains integer or float number from corresponding string representation. The str() function works the opposite. It surrounds an integer or a float object with quotes ('') to return a str object. The str() function returns the string.

representation of any Python object. In this section, we shall see different examples of str() function in Python.

The str() function has three parameters. First required parameter (or argument) is the object whose string representation we want. Other two operators, encoding and errors, are optional.

We shall execute str() function in Python console to easily verify that the returned object is a string, with the enclosing quotation marks ('').

Integer to string

```
>>> a = str(10)
>>> a
'10'
>>> type(a)
<class 'str'>
```

## Float to String

str() function converts floating point objects with both the notations of floating point, standard notation with a decimal point separating integer and fractional part, and the scientific notation to string object.

```
>>> a=str(11.10)
>>> a
'11.1'
>>> type(a)
<class 'str'>
>>> a = str(2/5)
>>> a
'0.4'
>>> type(a)
<class 'str'>
```

In the second case, a division expression is given as argument to str() function. Note that the expression is evaluated first and then result is converted to string.

Floating points in scientific notations using E or e and with positive or negative power are converted to string with str() function.

```
>>> a=str(10E4)
>>> a
'100000.0'
>>> type(a)
<class 'str'>
>>> a=str(1.23e-4)
>>> a
'0.000123'
>>> type(a)
<class 'str'>
```

When Boolean constant is entered as argument, it is surrounded by () so that True becomes 'True'. List and Tuple objects can also be given argument to str() function. The resultant string is the list/tuple surrounded by () .

```
>>> a=str('True')
>>> a
'True'
>>> a=str([1,2,3])
>>> a
'[1, 2, 3]'
>>> a=str((1,2,3))
>>> a
'(1, 2, 3)'
>>> a=str({1:100, 2:200, 3:300})
>>> a
'{1: 100, 2: 200, 3: 300}'
```

## Conversion of Sequence Types

List, Tuple and String are Python's sequence types. They are ordered or indexed collection of items.

A string and tuple can be converted into a list object by using the **list()** function. Similarly, the **tuple()** function converts a string or list to a tuple.

We shall an object each of these three sequence types and study their inter-conversion.

```
>>> a=[1,2,3,4,5]
>>> b=(1,2,3,4,5)
```

```

>>> c="Hello"
### list() separates each character in the string and builds the list
>>> obj=list(c)
>>> obj
['H', 'e', 'l', 'l', 'o']

### The parentheses of tuple are replaced by square brackets
>>> obj=list(b)
>>> obj
[1, 2, 3, 4, 5]

### tuple() separates each character from string and builds a tuple of
characters
>>> obj=tuple(c)
>>> obj
('H', 'e', 'l', 'l', 'o')

### square brackets of list are replaced by parentheses.
>>> obj=tuple(a)
>>> obj
(1, 2, 3, 4, 5)

### str() function puts the list and tuple inside the quote symbols.
>>> obj=str(a)
>>> obj
'[1, 2, 3, 4, 5]'

>>> obj=str(b)
>>> obj
'(1, 2, 3, 4, 5)'

```

Thus Python's explicit type casting feature allows conversion of one data type to other with the help of its built-in functions.

## Python - Unicode System

Software applications often require to display messages output in a variety in different languages such as in English, French, Japanese, Hebrew, or Hindi. Python's string type uses the Unicode Standard for representing characters. It makes the program possible to work with all these different possible characters.

A character is the smallest possible component of a text. 'A', 'B', 'C', etc., are all different characters. So are 'È' and 'Í'.

According to The Unicode standard, characters are represented by code points. A code point value is an integer in the range 0 to 0x10FFFF.

A sequence of code points is represented in memory as a set of code units, mapped to 8-bit bytes. The rules for translating a Unicode string into a sequence of bytes are called a

character encoding.

Three types of encodings are present, UTF-8, UTF-16 and UTF-32. UTF stands for **Unicode Transformation Format**.

Python 3.0 onwards has built-in support for Unicode. The **str** type contains Unicode characters, hence any string created using single, double or the triple-quoted string syntax is stored as Unicode. The default encoding for Python source code is UTF-8.

Hence, string may contain literal representation of a Unicode character (3/4) or its Unicode value (\u00BE).

&lt;/&gt;

Open Compiler

```
var = "3/4"  
print (var)  
var = "\u00BE"  
print (var)
```

This above code will produce the following **output** –

```
'3/4'  
3/4
```

In the following example, a string '10' is stored using the Unicode values of 1 and 0 which are \u0031 and \u0030 respectively.

&lt;/&gt;

Open Compiler

```
var = "\u0031\u0030"  
print (var)
```

It will produce the following **output** –

```
10
```

Strings display the text in a human-readable format, and bytes store the characters as binary data. Encoding converts data from a character string to a series of bytes. Decoding translates the bytes back to human-readable characters and symbols. It is important not to confuse these two methods. encode is a string method, while decode is a method of the Python byte object.

In the following example, we have a string variable that consists of ASCII characters. ASCII is a subset of Unicode character set. The encode() method is used to convert it into a bytes object.

&lt;/&gt;

Open Compiler

```
string = "Hello"
tobytes = string.encode('utf-8')
print (tobytes)
string = tobytes.decode('utf-8')
print (string)
```

The decode() method converts byte object back to the str object. The encodeing method used is utf-8.

b'Hello'

Hello

In the following example, the Rupee symbol (₹) is stored in the variable using its Unicode value. We convert the string to bytes and back to str.

&lt;/&gt;

Open Compiler

```
string = "\u20B9"
print (string)
tobytes = string.encode('utf-8')
print (tobytes)
string = tobytes.decode('utf-8')
print (string)
```

When you execute the above code, it will produce the following **output** –

₹

b'\xe2\x82\xb9'

₹

## Python - Literals

In computer science, a literal is a notation for representing a fixed value in source code. For example, in the assignment statement.

```
x = 10
```

Here 10 is a literal as numeric value representing 10 is directly stored in memory. However,

```
y = x*2
```

Here, even if the expression evaluates to 20, it is not literally included in source code. You can also declare an int object with built-in int() function –

```
x = int(10)
```

However, this is also an indirect way of instantiation and not with literal.

You can create use literal representation for creating object of any built-in data type.

## Integer Literal

Any representation involving only the digit symbols (0 to 9) creates an object of **int** type. The object so declared may be referred by a variable using an assignment operator.

Take a look at the following **example** –

```
x = 10  
y = -25  
z = 0
```

Python allows an integer to be represented as an octal number or a hexadecimal number. A numeric representation with only eight digit symbols (0 to 7) but prefixed by 0o or 0O is an octal number.

```
x = 0034
```

Similarly, a series of hexadecimal symbols (0 to 9 and a to f), prefixed by 0x or 0X represents an integer in Hexadecimal form.

```
x = 0X1C
```

However, it may be noted that, even if you use octal or hexadecimal literal notation, Python internally treats it as of **int** type.

</>

Open Compiler

```
# Using Octal notation
x = 0034
print ("0034 in octal is", x, type(x))
# Using Hexadecimal notation
x = 0X1c
print ("0X1c in Hexadecimal is", x, type(x))
```

When you run this code, it will produce the following **output** –

```
0O34 in octal is 28 <class 'int'>
0X1c in Hexadecimal is 28 <class 'int'>
```

## Float Literal

A floating point number consists of an integral part and a fractional part. Conventionally, a decimal point symbol (.) separates these two parts in a literal representation of a float. For example,

```
x = 25.55
y = 0.05
z = -12.2345
```

For a floating point number which is too large or too small, where number of digits before or after decimal point is more, a scientific notation is used for a compact literal representation. The symbol E or e followed by positive or negative integer, follows after the integer part.

For example, a number 1.23E05 is equivalent to 123000.00. Similarly, 1.23e-2 is equivalent to 0.0123

</>

Open Compiler

```
# Using normal floating point notation
x = 1.23
print ("1.23 in normal float literal is", x, type(x))
# Using Scientific notation
x = 1.23E5
print ("1.23E5 in scientific notation is", x, type(x))
x = 1.23E-2
print ("1.23E-2 in scientific notation is", x, type(x))
```

Here, you will get the following **output** –

```
1.23 in normal float literal is 1.23 <class 'float'>
1.23E5 in scientific notation is 123000.0 <class 'float'>
1.23E-2 in scientific notation is 0.0123 <class 'float'>
```

## Complex Literal

A complex number comprises of a real and imaginary component. The imaginary component is any number (integer or floating point) multiplied by square root of "-1" ( $\sqrt{-1}$ ). In literal representation ( $\sqrt{-1}$ ) is representation by "j" or "J". Hence, a literal representation of a complex number takes a form  $x+yj$ .

&lt;/&gt;

Open Compiler

```
#Using literal notation of complex number
x = 2+3j
print ("2+3j complex literal is", x, type(x))
y = 2.5+4.6j
print ("2.5+4.6j complex literal is", x, type(x))
```

This code will produce the following **output** –

```
2+3j complex literal is (2+3j) <class 'complex'>
2.5+4.6j complex literal is (2+3j) <class 'complex'>
```

## String Literal

A string object is one of the sequence data types in Python. It is an immutable sequence of Unicode code points. Code point is a number corresponding to a character according to Unicode standard. Strings are objects of Python's built-in class 'str'.

String literals are written by enclosing a sequence of characters in single quotes ('hello'), double quotes ("hello") or triple quotes (''"hello"" or """hello""").

&lt;/&gt;

Open Compiler

```
var1='hello'
print (''hello' in single quotes is:', var1, type(var1))
```

```

var2="hello"
print ('"hello" in double quotes is:', var1, type(var1))
var3='''hello'''
print ("'''hello''' in triple quotes is:", var1, type(var1))
var4="""hello"""
print ('"""hello"" in triple quotes is:', var1, type(var1))

```

Here, you will get the following **output** –

```

'hello' in single quotes is: hello <class 'str'>
"hello" in double quotes is: hello <class 'str'>
'''hello''' in triple quotes is: hello <class 'str'>
"""hello"" in triple quotes is: hello <class 'str'>

```

If it is required to embed double quotes as a part of string, the string itself should be put in single quotes. On the other hand, if single quoted text is to be embedded, string should be written in double quotes.

</>

[Open Compiler](#)

```

var1='Welcome to "Python Tutorial" from TutorialsPoint'
print (var1)
var2="Welcome to 'Python Tutorial' from TutorialsPoint"
print (var2)

```

It will produce the following **output** –

```

Welcome to "Python Tutorial" from TutorialsPoint
Welcome to 'Python Tutorial' from TutorialsPoint

```

## List Literal

List object in Python is a collection of objects of other data type. List is an ordered collection of items not necessarily of same type. Individual object in the collection is accessed by index starting with zero.

Literal representation of a list object is done with one or more items which are separated by comma and enclosed in square brackets [ ].

</>

[Open Compiler](#)

```
L1=[1,"Ravi",75.50, True]
print (L1, type(L1))
```

It will produce the following **output** –

```
[1, 'Ravi', 75.5, True] <class 'list'>
```

## Tuple Literal

Tuple object in Python is a collection of objects of other data type. Tuple is an ordered collection of items not necessarily of same type. Individual object in the collection is accessed by index starting with zero.

Literal representation of a tuple object is done with one or more items which are separated by comma and enclosed in parentheses () .

```
</>
```

Open Compiler

```
T1=(1,"Ravi",75.50, True)
print (T1, type(T1))
```

It will produce the following **output** –

```
[1, 'Ravi', 75.5, True] <class tuple>
```

Default delimiter for Python sequence is parentheses, which means a comma separated sequence without parentheses also amounts to declaration of a tuple.

```
</>
```

Open Compiler

```
T1=1,"Ravi",75.50, True
print (T1, type(T1))
```

Here too, you will get the same **output** –

```
[1, 'Ravi', 75.5, True] <class tuple>
```

## Dictionary Literal

Like list or tuple, dictionary is also a collection data type. However, it is not a sequence. It is an unordered collection of items, each of which is a key-value pair. Value is bound to key by the ":" symbol. One or more key:value pairs separated by comma are put inside curly brackets to form a dictionary object.

```
capitals={"USA":"New York", "France":"Paris", "Japan":"Tokyo",
"India":"New Delhi"}
numbers={1:"one", 2:"Two", 3:"three",4:"four"}
points={"p1":(10,10), "p2":(20,20)}
```

Key should be an immutable object. Number, string or tuple can be used as key. Key cannot appear more than once in one collection. If a key appears more than once, only the last one will be retained. Values can be of any data type. One value can be assigned to more than one keys. For example,

```
staff={"Krishna":"Officer", "Rajesh":"Manager", "Ragini":"officer", "Anil":"Clerk"
```

## Python - Operators

In Python as well as any programming language, Operators are symbols (sometimes keywords) that are predefined to perform a certain most commonly required operations on one or more operands.

### Types of Operators

Python language supports the following types of operators –

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

## Python - Arithmetic Operators

In Python, numbers are the most frequently used data type. Python uses the same symbols for basic arithmetic operations Everybody is familiar with, i.e., "+" for addition, "-" for subtraction, "\*" for multiplication (most programming languages use "\*" instead of the "x" as used in maths/algebra), and "/" for division (again for the "÷" used in Mathematics).

In addition, Python defines few more arithmetic operators. They are "%" (Modulus), "\*\*" (Exponent) and "//" (Floor division).

Arithmetic operators are binary operators in the sense they operate on two operands. Python fully supports mixed arithmetic. That is, the two operands can be of two different number types. In such a situation, Python widens the narrower of the operands. An integer object is narrower than float object, and float is narrower than complex object. Hence, the result of arithmetic operation of int and a float is a float. Result of float and a complex is a complex number, similarly, operation on an integer and a complex object results in a complex object.

Let us study these operators with examples.

## Python – Addition Operator (+)

This operator pronounced as plus, is a basic arithmetic operator. It adds the two numeric operands on the either side and returns the addition result.

In the following example, the two integer variables are the operands for the "+" operator.

```
</> Open Compiler
a=10
b=20
print ("Addition of two integers")
print ("a =",a,"b =",b,"addition =",a+b)
```

It will produce the following **output** –

```
Addition of two integers
a = 10 b = 20 addition = 30
```

Addition of integer and float results in a float.

```
</> Open Compiler
a = 10.5
b = 20.5
addition = a + b
```

```
a=10  
b=20.5  
print ("Addition of integer and float")  
print ("a =",a,"b =",b,"addition =",a+b)
```

It will produce the following **output** –

```
Addition of integer and float  
a = 10 b = 20.5 addition = 30.5
```

The result of adding float to complex is a complex number.

```
a=10+5j  
b=20.5  
print ("Addition of complex and float")  
print ("a=",a,"b=",b,"addition=",a+b)
```

It will produce the following **output** –

```
Addition of complex and float  
a= (10+5j) b= 20.5 addition= (30.5+5j)
```

## Python – Subtraction Operator (-)

This operator, known as minus, subtracts the second operand from the first. The resultant number is negative if the second operand is larger.

First example shows subtraction of two integers.

```
</>  
  
a=10  
b=20  
print ("Subtraction of two integers:")  
print ("a =",a,"b =",b,"a-b =",a-b)  
print ("a =",a,"b =",b,"b-a =",b-a)
```

Open Compiler

Result –

Subtraction of two integers

a = 10 b = 20 a-b = -10

a = 10 b = 20 b-a = 10

Subtraction of an integer and a float follows the same principle.

</>

Open Compiler

```
a=10  
b=20.5  
print ("subtraction of integer and float")  
print ("a=",a,"b=",b,"a-b=",a-b)  
print ("a=",a,"b=",b,"b-a=",b-a)
```

It will produce the following **output** –

subtraction of integer and float

a= 10 b= 20.5 a-b= -10.5

a= 10 b= 20.5 b-a= 10.5

In the subtraction involving a complex and a float, real component is involved in the operation.

</>

Open Compiler

```
a=10+5j  
b=20.5  
print ("subtraction of complex and float")  
print ("a=",a,"b=",b,"a-b=",a-b)  
print ("a=",a,"b=",b,"b-a=",b-a)
```

It will produce the following **output** –

subtraction of complex and float

a= (10+5j) b= 20.5 a-b= (-10.5+5j)

a= (10+5j) b= 20.5 b-a= (10.5-5j)

## Python – Multiplication Operator (\*)

The \* (asterisk) symbol is defined as a multiplication operator in Python (as in many languages). It returns the product of the two operands on its either side. If any of the operands negative, the result is also negative. If both are negative, the result is positive. Changing the order of operands doesn't change the result

&lt;/&gt;

[Open Compiler](#)

```
a=10
b=20
print ("Multiplication of two integers")
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following **output** –

Multiplication of two integers

a = 10 b = 20 a\*b = 200

In multiplication, a float operand may have a standard decimal point notation, or a scientific notation.

&lt;/&gt;

[Open Compiler](#)

```
a=10
b=20.5
print ("Multiplication of integer and float")
print ("a=",a,"b=",b,"a*b=",a*b)
a=-5.55
b=6.75E-3
print ("Multiplication of float and float")
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following **output** –

Multiplication of integer and float

a = 10 b = 20.5 a-b = -10.5

Multiplication of float and float

a = -5.55 b = 0.00675 a\*b = -0.03746249999999996

For the multiplication operation involving one complex operand, the other operand multiplies both the real part and imaginary part.

&lt;/&gt;

[Open Compiler](#)

```
a=10+5j  
b=20.5  
print ("Multiplication of complex and float")  
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following **output** –

```
Multiplication of complex and float  
a = (10+5j) b = 20.5 a*b = (205+102.5j)
```

## Python – Division Operator (/)

The "/" symbol is usually called as forward slash. The result of division operator is numerator (left operand) divided by denominator (right operand). The resultant number is negative if any of the operands is negative. Since infinity cannot be stored in the memory, Python raises `ZeroDivisionError` if the denominator is 0.

The result of division operator in Python is always a float, even if both operands are integers.

&lt;/&gt;

[Open Compiler](#)

```
a=10  
b=20  
print ("Division of two integers")  
print ("a=",a,"b=",b,"a/b=",a/b)  
print ("a=",a,"b=",b,"b/a=",b/a)
```

It will produce the following **output** –

```
Division of two integers  
a= 10 b= 20 a/b= 0.5  
a= 10 b= 20 b/a= 2.0
```

In Division, a float operand may have a standard decimal point notation, or a scientific notation.

&lt;/&gt;

[Open Compiler](#)

```
a=10
b=-20.5
print ("Division of integer and float")
print ("a=",a,"b=",b,"a/b=",a/b)
a=-2.50
b=1.25E2
print ("Division of float and float")
print ("a=",a,"b=",b,"a/b=",a/b)
```

It will produce the following **output** –

```
Division of integer and float
a= 10 b= -20.5 a/b= -0.4878048780487805
Division of float and float
a= -2.5 b= 125.0 a/b= -0.02
```

When one of the operands is a complex number, division between the other operand and both parts of complex number (real and imaginary) object takes place.

</>
Open Compiler

```
a=7.5+7.5j
b=2.5
print ("Division of complex and float")
print ("a =",a,"b =",b,"a/b =",a/b)
print ("a =",a,"b =",b,"b/a =",b/a)
```

It will produce the following **output** –

```
Division of complex and float
a = (7.5+7.5j) b = 2.5 a/b = (3+3j)
a = (7.5+7.5j) b = 2.5 b/a = (0.1666666666666666-0.1666666666666666j)
```

If the numerator is 0, the result of division is always 0 except when denominator is 0, in which case, Python raises ZeroDivisionError with Division by Zero error message.

</>
Open Compiler

```
a=0
b=2.5
print ("a=",a,"b=",b,"a/b=",a/b)
print ("a=",a,"b=",b,"b/a=",b/a)
```

It will produce the following **output** –

```
a= 0 b= 2.5 a/b= 0.0
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 20, in <module>
    print ("a=",a,"b=",b,"b/a=",b/a)
                  ~^~
ZeroDivisionError: float division by zero
```

## Python – Modulus Operator (%)

Python defines the "%" symbol, which is known as Percent symbol, as Modulus (or modulo) operator. It returns the remainder after the denominator divides the numerator. It can also be called Remainder operator. The result of the modulus operator is the number that remains after the integer quotient. To give an example, when 10 is divided by 3, the quotient is 3 and remainder is 1. Hence, 10%3 (normally pronounced as 10 mod 3) results in 1.

If both the operands are integer, the modulus value is an integer. If numerator is completely divisible, remainder is 0. If numerator is smaller than denominator, modulus is equal to the numerator. If denominator is 0, Python raises ZeroDivisionError.

&lt;/&gt;

Open Compiler

```
a=10
b=2
print ("a=",a, "b=",b, "a%b=", a%b)
a=10
b=4
print ("a=",a, "b=",b, "a%b=", a%b)
print ("a=",a, "b=",b, "b%a=", b%a)
a=0
b=10
print ("a=",a, "b=",b, "a%b=", a%b)
print ("a=", a, "b=", b, "b%a=", b%a)
```

It will produce the following **output** –

```
a= 10 b= 2 a%b= 0
a= 10 b= 4 a%b= 2
a= 10 b= 4 b%a= 4
a= 0 b= 10 a%b= 0
```

Traceback (most recent call last):

```
File "C:\Users\mlath\examples\example.py", line 13, in <module>
    print ("a=", a, "b=", b, "b%a=",b%a)
                  ~^~
```

`ZeroDivisionError: integer modulo by zero`

If any of the operands is a float, the mod value is always float.

</>

Open Compiler

```
a=10
b=2.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=10
b=1.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=7.7
b=2.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=12.4
b=3
print ("a=",a, "b=",b, "a%b=", a%b)
```

It will produce the following **output** –

```
a= 10 b= 2.5 a%b= 0.0
a= 10 b= 1.5 a%b= 1.0
a= 7.7 b= 2.5 a%b= 0.20000000000000018
a= 12.4 b= 3 a%b= 0.40000000000000036
```

Python doesn't accept complex numbers to be used as operand in modulus operation. It throws `TypeError: unsupported operand type(s) for %`.

## Python – Exponent Operator (\*\*)

Python uses `**` (double asterisk) as the exponent operator (sometimes called raised to operator). So, for `a**b`, you say a raised to b, or even bth power of a.

If in the exponentiation expression, both operands are integer, result is also an integer. In case either one is a float, the result is float. Similarly, if either one operand is complex number, exponent operator returns a complex number.

If the base is 0, the result is 0, and if the index is 0 then the result is always 1.

&lt;/&gt;

Open Compiler

```

a=10
b=2
print ("a=",a, "b=",b, "a**b=", a**b)
a=10
b=1.5
print ("a=",a, "b=",b, "a**b=", a**b)
a=7.7
b=2
print ("a=",a, "b=",b, "a**b=", a**b)
a=1+2j
b=4
print ("a=",a, "b=",b, "a**b=", a**b)
a=12.4
b=0
print ("a=",a, "b=",b, "a**b=", a**b)
print ("a=",a, "b=",b, "b**a=", b**a)

```

It will produce the following **output** –

```

a= 10 b= 2 a**b= 100
a= 10 b= 1.5 a**b= 31.622776601683793
a= 7.7 b= 2 a**b= 59.290000000000006
a= (1+2j) b= 4 a**b= (-7-24j)
a= 12.4 b= 0 a**b= 1.0
a= 12.4 b= 0 b**a= 0.0

```

## Python – Floor Division Operator (//)

Floor division is also called as integer division. Python uses `//` (double forward slash) symbol for the purpose. Unlike the modulus or modulo which returns the remainder, the floor division gives the quotient of the division of operands involved.

If both operands are positive, floor operator returns a number with fractional part removed from it. For example, the floor division of 9.8 by 2 returns 4 (pure division is 4.9, strip the fractional part, result is 4).

But if one of the operands is negative, the result is rounded away from zero (towards negative infinity). Floor division of -9.8 by 2 returns 5 (pure division is -4.9, rounded away from 0).

&lt;/&gt;

Open Compiler

```
a=9  
b=2  
print ("a=",a, "b=",b, "a//b=", a//b)  
a=9  
b=-2  
print ("a=",a, "b=",b, "a//b=", a//b)  
a=10  
b=1.5  
print ("a=",a, "b=",b, "a//b=", a//b)  
a=-10  
b=1.5  
print ("a=",a, "b=",b, "a//b=", a//b)
```

It will produce the following **output** –

```
a= 9 b= 2 a//b= 4  
a= 9 b= -2 a//b= -5  
a= 10 b= 1.5 a//b= 6.0  
a= -10 b= 1.5 a//b= -7.0
```

## Python – Complex Number Arithmetic

Arithmetic operators behave slightly differently when the both operands are complex number objects.

Addition and subtraction of complex numbers is a simple addition/subtraction of respective real and imaginary components.

&lt;/&gt;

Open Compiler

```
a=2.5+3.4j
b=-3+1.0j
print ("Addition of complex numbers - a=",a, "b=",b, "a+b=", a+b)
print ("Subtraction of complex numbers - a=",a, "b=",b, "a-b=", a-b)
```

It will produce the following **output** –

```
Addition of complex numbers - a= (2.5+3.4j) b= (-3+1j) a+b= (-0.5+4.4j)
Subtraction of complex numbers - a= (2.5+3.4j) b= (-3+1j) a-b=
(5.5+2.4j)
```

Multiplication of complex numbers is similar to multiplication of two binomials in algebra. If " $a+bj$ " and " $x+yj$ " are two complex numbers, then their multiplication is given by this formula –

$$(a+bj)*(x+yj) = ax+ayj+xbj+byj^2 = (ax-by)+(ay+xb)j$$

**For example,**

```
a=6+4j
b=3+2j
c=a*b
c=(18-8)+(12+12)j
c=10+24j
```

The following program confirms the result –

```
a=6+4j
b=3+2j
print ("Multiplication of complex numbers - a=",a, "b=",b, "a*b=", a*b)
```

To understand the how the division of two complex numbers takes place, we should use the conjugate of a complex number. Python's complex object has a `conjugate()` method that returns a complex number with the sign of imaginary part reversed.

```
>>> a=5+6j
>>> a.conjugate()
(5-6j)
```

To divide two complex numbers, divide and multiply the numerator as well as the denominator with the conjugate of denominator.

```
a=6+4j
b=3+2j
c=a/b
c=(6+4j)/(3+2j)
c=(6+4j)*(3-2j)/3+2j)*(3-2j)
c=(18-12j+12j+8)/(9-6j+6j+4)
c=26/13
c=2+0j
```

To verify, run the following code –

&lt;/&gt;

Open Compiler

```
a=6+4j
b=3+2j
print ("Division of complex numbers - a=",a, "b=",b, "a/b=", a/b)
```

Complex class in Python doesn't support the modulus operator (%) and floor division operator (//).

## Python - Assignment Operators

The = (equal to) symbol is defined as assignment operator in Python. The value of Python expression on its right is assigned to a single variable on its left. The = symbol as in programming in general (and Python in particular) should not be confused with its usage in Mathematics, where it states that the expressions on the either side of the symbol are equal.

In addition to the simple assignment operator, Python provides few more assignment operators for advanced use. They are called cumulative or augmented assignment operators. In this chapter, we shall learn to use augmented assignment operators defined in Python.

Consider following Python statements –

```
a=10
b=5
a=a+b
print (a)
```

At the first instance, at least for somebody new to programming but who knows maths, the statement "a=a+b" looks strange. How could a be equal to "a+b"? However, it needs

to be reemphasized that the `=` symbol is an assignment operator here and not used to show the equality of LHS and RHS.

Because it is an assignment, the expression on right evaluates to 15, the value is assigned to `a`.

In the statement `"a+=b"`, the two operators `+` and `=` can be combined in a `"+="` operator. It is called as add and assign operator. In a single statement, it performs addition of two operands `"a"` and `"b"`, and result is assigned to operand on left, i.e., `"a"`.

The `+=` operator is an augmented operator. It is also called cumulative addition operator, as it adds `"b"` in `"a"` and assigns the result back to a variable.

Python has the augmented assignment operators for all arithmetic and comparison operators.

## Python - Augmented Addition Operator (`+=`)

This operator combines addition and assignment in one statement. Since Python supports mixed arithmetic, the two operands may be of different types. However, the type of left operand changes to the operand of on right, if it is wider.

Following examples will help in understanding how the `"+="` operator works –

```
</> Open Compiler

a=10
b=5
print ("Augmented addition of int and int")
a+=b #equivalent to a=a+b
print ("a=",a, "type(a):", type(a))
a=10
b=5.5
print ("Augmented addition of int and float")
a+=b #equivalent to a=a+b
print ("a=",a, "type(a):", type(a))
a=10.50
b=5+6j
print ("Augmented addition of float and complex")
a+=b #equivalent to a=a+b
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented addition of int and int  
a= 15 type(a): <class 'int'>  
Augmented addition of int and float  
a= 15.5 type(a): <class 'float'>  
Augmented addition of float and complex  
a= (15.5+6j) type(a): <class 'complex'>
```

## Python – Augmented Subtraction Operator (-=)

Use -= symbol to perform subtract and assign operations in a single statement. The "a-=b" statement performs "a=a-b" assignment. Operands may be of any number type. Python performs implicit type casting on the object which is narrower in size.

&lt;/&gt;

Open Compiler

```
a=10  
b=5  
print ("Augmented subtraction of int and int")  
a-=b #equivalent to a=a-b  
print ("a=",a, "type(a):", type(a))  
a=10  
b=5.5  
print ("Augmented subtraction of int and float")  
a-=b #equivalent to a=a-b  
print ("a=",a, "type(a):", type(a))  
a=10.50  
b=5+6j  
print ("Augmented subtraction of float and complex")  
a-=b #equivalent to a=a-b  
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented subtraction of int and int  
a= 5 type(a): <class 'int'>  
Augmented subtraction of int and float  
a= 4.5 type(a): <class 'float'>  
Augmented subtraction of float and complex  
a= (5.5-6j) type(a): <class 'complex'>
```

## Python – Augmented Multiplication Operator (\*=)

The "`*=`" operator works on similar principle. "`a*=b`" performs multiply and assign operations, and is equivalent to "`a=a*b`". In case of augmented multiplication of two complex numbers, the rule of multiplication as discussed in the previous chapter is applicable.

&lt;/&gt;

Open Compiler

```

a=10
b=5
print ("Augmented multiplication of int and int")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))
a=10
b=5.5
print ("Augmented multiplication of int and float")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))
a=6+4j
b=3+2j
print ("Augmented multiplication of complex and complex")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))

```

It will produce the following **output** –

```

Augmented multiplication of int and int
a= 50 type(a): <class 'int'>
Augmented multiplication of int and float
a= 55.0 type(a): <class 'float'>
Augmented multiplication of complex and complex
a= (10+24j) type(a): <class 'complex'>

```

## Python – Augmented Division Operator (/=)

The combination symbol "`/=`" acts as divide and assignment operator, hence "`a/=b`" is equivalent to "`a=a/b`". The division operation of int or float operands is float. Division of two complex numbers returns a complex number. Given below are examples of augmented division operator.

&lt;/&gt;

[Open Compiler](#)

```
a=10
b=5
print ("Augmented division of int and int")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))
a=10
b=5.5
print ("Augmented division of int and float")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))
a=6+4j
b=3+2j
print ("Augmented division of complex and complex")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented division of int and int
a= 2.0 type(a): <class 'float'>
Augmented division of int and float
a= 1.81818181818181 type(a): <class 'float'>
Augmented division of complex and complex
a= (2+0j) type(a): <class 'complex'>
```

## Python – Augmented Modulus Operator (%=)

To perform modulus and assignment operation in a single statement, use the %= operator. Like the mod operator, its augmented version also is not supported for complex number.

&lt;/&gt;

[Open Compiler](#)

```
a=10
b=5
print ("Augmented modulus operator with int and int")
a%=b #equivalent to a=a%b
print ("a=",a, "type(a):", type(a))
a=10
```

```
b=5.5
print ("Augmented modulus operator with int and float")
a%=b #equivalent to a=a%b
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented modulus operator with int and int
a= 0 type(a): <class 'int'>
Augmented modulus operator with int and float
a= 4.5 type(a): <class 'float'>
```

## Python – Augmented Exponent Operator (\*\*=)

The "==" operator results in computation of "a" raised to "b", and assigning the value back to "a". Given below are some examples –

&lt;/&gt;

Open Compiler

```
a=10
b=5
print ("Augmented exponent operator with int and int")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))
a=10
b=5.5
print ("Augmented exponent operator with int and float")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))
a=6+4j
b=3+2j
print ("Augmented exponent operator with complex and complex")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented exponent operator with int and int
a= 100000 type(a): <class 'int'>
Augmented exponent operator with int and float
```

```
a= 316227.7660168379 type(a): <class 'float'>
Augmented exponent operator with complex and complex
a= (97.52306038414744-62.22529992036203j) type(a): <class 'complex'>
```

## Python – Augmented Floor division Operator (//=)

For performing floor division and assignment in a single statement, use the " //= " operator. "a//=b" is equivalent to "a=a//b". This operator cannot be used with complex numbers.

</>

Open Compiler

```
a=10
b=5
print ("Augmented floor division operator with int and int")
a//=b #equivalent to a=a//b
print ("a=",a, "type(a):", type(a))
a=10
b=5.5
print ("Augmented floor division operator with int and float")
a//=b #equivalent to a=a//b
print ("a=",a, "type(a):", type(a))
```

It will produce the following **output** –

```
Augmented floor division operator with int and int
a= 2 type(a): <class 'int'>
Augmented floor division operator with int and float
a= 1.0 type(a): <class 'float'>
```

## Python - Comparison Operators

Comparison operators in Python are very important in Python's conditional statements (**if**, **else** and **elif**) and looping statements (while and for loops). Like arithmetic operators, the comparison operators "-" also called relational operators, ("**<**" stands for less than, and "**>**" stands for greater than) are well known.

Python uses two more operators, combining "=" symbol with these two. The "**<=**" symbol is for less than or equal to. The "**>=**" symbol is for greater than or equal to.

Python has two more comparison operators in the form of "**==**" and "**!=**". They are for is equal to and is not equal to operators. Hence, there are six comparison operators in

## Python.

<	Less than	a < b
>	Greater than	a > b
<=	Less than or equal to	a <= b
>=	Greater than or equal to	a >= b
==	Is equal to	a == b
!=	Is not equal to	a != b

Comparison operators are binary in nature, requiring two operands. An expression involving a comparison operator is called a Boolean expression, and always returns either True or False.

&lt;/&gt;

Open Compiler

```
a=5
b=7
print (a>b)
print (a<b)
```

It will produce the following **output** –

False

True

Both the operands may be Python literals, variables or expressions. Since Python supports mixed arithmetic, you can have any number type operands.

The following code demonstrates the use of Python's **comparison operators with integer numbers** –

&lt;/&gt;

Open Compiler

```
print ("Both operands are integer")
a=5
b=7
print ("a=",a, "b=",b, "a>b is", a>b)
print ("a=",a, "b=",b, "a<b is", a<b)
```

```
print ("a=",a, "b=",b, "a==b is",a==b)
print ("a=",a, "b=",b, "a!=b is",a!=b)
```

It will produce the following **output** –

Both operands are integer

a= 5 b= 7 a>b is False

a= 5 b= 7 a<b is True

a= 5 b= 7 a==b is False

a= 5 b= 7 a!=b is True

## Comparison of Float Number

In the following example, an integer and a float operand are compared.

</>

Open Compiler

```
print ("comparison of int and float")
a=10
b=10.0
print ("a=",a, "b=",b, "a>b is", a>b)
print ("a=",a, "b=",b, "a<b is", a<b)
print ("a=",a, "b=",b, "a==b is", a==b)
print ("a=",a, "b=",b, "a!=b is", a!=b)
```

It will produce the following **output** –

comparison of int and float

a= 10 b= 10.0 a>b is False

a= 10 b= 10.0 a<b is False

a= 10 b= 10.0 a==b is True

a= 10 b= 10.0 a!=b is False

## Comparison of Complex umbers

Although complex object is a number data type in Python, its behavior is different from others. Python doesn't support < and > operators, however it does support equality (==) and inequality (!=) operators.

</>

Open Compiler

```
print ("comparison of complex numbers")
a=10+1j
b=10.-1j
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following **output** –

```
comparison of complex numbers
a= (10+1j) b= (10-1j) a==b is False
a= (10+1j) b= (10-1j) a!=b is True
```

You get a `TypeError` with less than or greater than operators.

&lt;/&gt;

Open Compiler

```
print ("comparison of complex numbers")
a=10+1j
b=10.-1j
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
```

It will produce the following **output** –

```
comparison of complex numbers
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 5, in <module>
    print ("a=",a, "b=",b,"a<b is",a<b)
               ^
TypeError: '<' not supported between instances of 'complex' and
'complex'
```

## Comparison of Booleans

Boolean objects in Python are really integers: `True` is 1 and `False` is 0. In fact, Python treats any non-zero number as `True`. In Python, comparison of Boolean objects is possible. "`False < True`" is `True`!

&lt;/&gt;

Open Compiler

```

print ("comparison of Booleans")
a=True
b=False
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)

```

It will produce the following **output** –

```

comparison of Booleans
a= True b= False a<b is False
a= True b= False a>b is True
a= True b= False a==b is False
a= True b= False a!=b is True

```

## Comparison of Sequence Types

In Python, comparison of only similar sequence objects can be performed. A string object is comparable with another string only. A list cannot be compared with a tuple, even if both have same items.

&lt;/&gt;

Open Compiler

```

print ("comparison of different sequence types")
a=(1,2,3)
b=[1,2,3]
print ("a=",a, "b=",b,"a<b is",a<b)

```

It will produce the following **output** –

```

comparison of different sequence types
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 5, in <module>
    print ("a=",a, "b=",b,"a<b is",a<b)
                           ^
TypeError: '<' not supported between instances of 'tuple' and 'list'

```

Sequence objects are compared by lexicographical ordering mechanism. The comparison starts from item at 0th index. If they are equal, comparison moves to next index till the

items at certain index happen to be not equal, or one of the sequences is exhausted. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one.

Which of the operands is greater depends on the difference in values of items at the index where they are unequal. For example, 'BAT'>'BAR' is True, as T comes after R in Unicode order.

If all items of two sequences compare equal, the sequences are considered equal.

&lt;/&gt;

Open Compiler

```
print ("comparison of strings")
a='BAT'
b='BALL'
print ("a=",a, "b=",b, "a<b is",a<b)
print ("a=",a, "b=",b, "a>b is",a>b)
print ("a=",a, "b=",b, "a==b is",a==b)
print ("a=",a, "b=",b, "a!=b is",a!=b)
```

It will produce the following **output** –

```
comparison of strings
a= BAT b= BALL a<b is False
a= BAT b= BALL a>b is True
a= BAT b= BALL a==b is False
a= BAT b= BALL a!=b is True
```

In the following example, two tuple objects are compared –

&lt;/&gt;

Open Compiler

```
print ("comparison of tuples")
a=(1,2,4)
b=(1,2,3)
print ("a=",a, "b=",b, "a<b is",a<b)
print ("a=",a, "b=",b, "a>b is",a>b)
print ("a=",a, "b=",b, "a==b is",a==b)
print ("a=",a, "b=",b, "a!=b is",a!=b)
```

It will produce the following **output** –

```
a= (1, 2, 4) b= (1, 2, 3) a<b is False
a= (1, 2, 4) b= (1, 2, 3) a>b is True
a= (1, 2, 4) b= (1, 2, 3) a==b is False
a= (1, 2, 4) b= (1, 2, 3) a!=b is True
```

## Comparison of Dictionary Objects

The use of "<" and ">" operators for Python's dictionary is not defined. In case of these operands, `TypeError: '<' not supported between instances of 'dict' and 'dict'` is reported.

Equality comparison checks if the length of both the dict items is same. Length of dictionary is the number of key-value pairs in it.

Python dictionaries are simply compared by length. The dictionary with fewer elements is considered less than a dictionary with more elements.

&lt;/&gt;

Open Compiler

```
print ("comparison of dictionary objects")
a={1:1,2:2}
b={2:2, 1:1, 3:3}
print ("a=",a, "b=",b, "a==b is",a==b)
print ("a=",a, "b=",b, "a!=b is",a!=b)
```

It will produce the following **output** –

```
comparison of dictionary objects
a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a==b is False
a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a!=b is True
```

## Python - Logical Operators

With Logical operators in Python, we can form compound Boolean expressions. Each operand for these logical operators is itself a Boolean expression. For example,

```
age>16 and marks>80
percentage<50 or attendance<75
```

Along with the keyword `False`, Python interprets `None`, numeric zero of all types, and empty sequences (strings, tuples, lists), empty dictionaries, and empty sets as `False`. All other values are treated as `True`.

There are three logical operators in Python. They are "and", "or" and "not". They must be in lowercase.

## The "and" Operator

For the compound Boolean expression to be True, both the operands must be True. If any or both operands evaluate to False, the expression returns False. The following table shows the scenarios.

<b>a</b>	<b>b</b>	<b>a and b</b>
F	F	F
F	T	F
T	F	F
T	T	T

## The "or" Operator

In contrast, the or operator returns True if any of the operands is True. For the compound Boolean expression to be False, both the operands have to be False, as the following table shows –

<b>a</b>	<b>b</b>	<b>a or b</b>
F	F	F
F	T	T
T	F	F
T	T	T

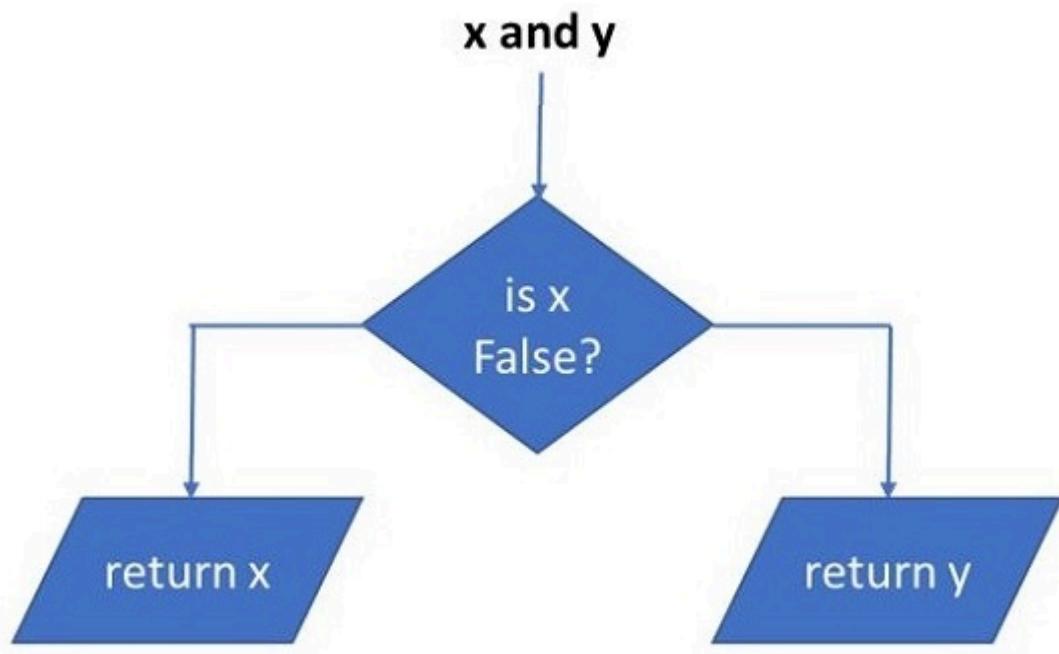
## The "not" Operator

This is a unary operator. The state of Boolean operand that follows, is reversed. As a result, not True becomes False and not False becomes True.

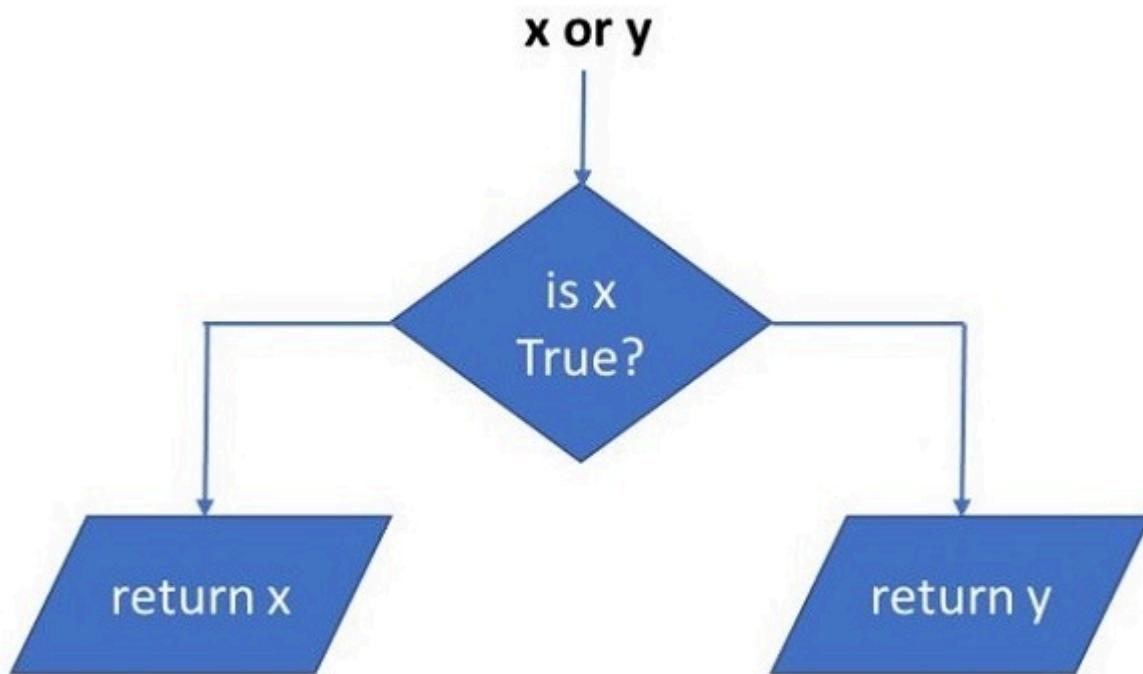
<b>a</b>	<b>not (a)</b>
F	T
T	F

## How the Python interpreter evaluates the logical operators?

The expression "x and y" first evaluates "x". If "x" is false, its value is returned; otherwise, "y" is evaluated and the resulting value is returned.



The expression "x or y" first evaluates "x"; if "x" is true, its value is returned; otherwise, "y" is evaluated and the resulting value is returned.



Some use cases of logical operators are given below –

&lt;/&gt;

Open Compiler

```
x = 10
y = 20
print("x > 0 and x < 10:", x > 0 and x < 10)
print("x > 0 and y > 10:", x > 0 and y > 10)
print("x > 10 or y > 10:", x > 10 or y > 10)
print("x%2 == 0 and y%2 == 0:", x%2 == 0 and y%2 == 0)
print ("not (x+y>15):", not (x+y)>15)
```

It will produce the following **output** –

```
x > 0 and x < 10: False
x > 0 and y > 10: True
x > 10 or y > 10: True
x%2 == 0 and y%2 == 0: True
not (x+y>15): False
```

We can use non-boolean operands with logical operators. Here, we need to note that any non-zero numbers, and non-empty sequences evaluate to True. Hence, the same truth tables of logical operators apply.

In the following example, numeric operands are used for logical operators. The variables "x", "y" evaluate to True, "z" is False

```
</> Open Compiler
x = 10
y = 20
z = 0
print("x and y:", x and y)
print("x or y:", x or y)
print("z or x:", z or x)
print("y or z:", y or z)
```

It will produce the following **output** –

```
x and y: 20
x or y: 10
z or x: 10
y or z: 20
```

The string variable is treated as True and an empty tuple as False in the following example –

```
</> Open Compiler  
a="Hello"  
b=tuple()  
print("a and b:",a and b)  
print("b or a:",b or a)
```

It will produce the following **output** –

```
a and b: ()  
b or a: Hello
```

Finally, two list objects below are non-empty. Hence x and y returns the latter, and x or y returns the former.

```
</> Open Compiler  
x=[1,2,3]  
y=[10,20,30]  
print("x and y:",x and y)  
print("x or y:",x or y)
```

It will produce the following **output** –

```
x and y: [10, 20, 30]  
x or y: [1, 2, 3]
```

## Python - Bitwise Operators

Python's bitwise operators are normally used with integer type objects. However, instead of treating the object as a whole, it is treated as a string of bits. Different operations are done on each bit in the string.

Python has six bitwise operators - &, |, ^, ~, << and >>. All these operators (except ~) are binary in nature, in the sense they operate on two operands. Each operand is a binary digit (bit) 1 or 0.

## Python – Bitwise AND Operator (&)

Bitwise AND operator is somewhat similar to logical and operator. It returns True only if both the bit operands are 1 (i.e. True). All the combinations are –

```
0 & 0 is 0
1 & 0 is 0
0 & 1 is 0
1 & 1 is 1
```

When you use integers as the operands, both are converted in equivalent binary, the & operation is done on corresponding bit from each number, starting from the least significant bit and going towards most significant bit.

Let us take two integers 60 and 13, and assign them to variables a and b respectively.

```
</> Open Compiler
a=60
b=13
print ("a:",a, "b:",b, "a&b:",a&b)
```

It will produce the following **output** –

```
a: 60 b: 13 a&b: 12
```

To understand how Python performs the operation, obtain the binary equivalent of each variable.

```
print ("a:", bin(a))
print ("b:", bin(b))
```

It will produce the following **output** –

```
a: 0b111100
b: 0b1101
```

For the sake of convenience, use the standard 8-bit format for each number, so that "a" is 00111100 and "b" is 00001101. Let us manually perform and operation on each corresponding bits of these two numbers.

```

0011 1100
  &
0000 1101
-----
0000 1100

```

Convert the resultant binary back to integer. You'll get 12, which was the result obtained earlier.

```

>>> int('00001100', 2)
12

```

## Python – Bitwise OR Operator (|)

The "|" symbol (called **pipe**) is the bitwise OR operator. If any bit operand is 1, the result is 1 otherwise it is 0.

0		0	is	0
0		1	is	1
1		0	is	1
1		1	is	1

Take the same values of a=60, b=13. The "|" operation results in 61. Obtain their binary equivalents.

&lt;/&gt;

Open Compiler

```

a=60
b=13
print ("a:",a, "b:",b, "a|b:",a|b)
print ("a:", bin(a))
print ("b:", bin(b))

```

It will produce the following **output** –

```

a: 60 b: 13 a|b: 61
a: 0b111100
b: 0b1101

```

To perform the "|" operation manually, use the 8-bit format.

```

0011 1100
|
0000 1101
-----
0011 1101

```

Convert the binary number back to integer to tally the result –

```

>>> int('00111101', 2)
61

```

## Python – Binary XOR Operator (^)

The term XOR stands for exclusive OR. It means that the result of OR operation on two bits will be 1 if only one of the bits is 1.

```

0 ^ 0 is 0
0 ^ 1 is 1
1 ^ 0 is 1
1 ^ 1 is 0

```

Let us perform XOR operation on a=60 and b=13.

```

</>
Open Compiler

a=60
b=13
print ("a:",a, "b:",b, "a^b:",a^b)

```

It will produce the following **output** –

```
a: 60 b: 13 a^b: 49
```

We now perform the bitwise XOR manually.

```

0011 1100
^
0000 1101
-----
0011 0001

```

The int() function shows 00110001 to be 49.

```
>>> int('00110001',2)
49
```

## Python – Binary NOT Operator (~)

This operator is the binary equivalent of logical NOT operator. It flips each bit so that 1 is replaced by 0, and 0 by 1, and returns the complement of the original number. Python uses 2's complement method. For positive integers, it is obtained simply by reversing the bits. For negative number,  $-x$ , it is written using the bit pattern for  $(x-1)$  with all of the bits complemented (switched from 1 to 0 or 0 to 1). Hence: (for 8 bit representation)

```
-1 is complement(1 - 1) = complement(0) = "11111111"
-10 is complement(10 - 1) = complement(9) = complement("00001001") = "11110110".
```

For  $a=60$ , its complement is –

```
</>
Open Compiler

a=60
print ("a:",a, "~a:", ~a)
```

It will produce the following **output** –

```
a: 60 ~a: -61
```

## Python – Left Shift Operator (<<)

Left shift operator shifts most significant bits to right by the number on the right side of the "<<" symbol. Hence, " $x << 2$ " causes two bits of the binary representation of to right. Let us perform left shift on 60.

```
</>
Open Compiler

a=60
print ("a:",a, "a<<2:", a<<2)
```

It will produce the following **output** –

```
a: 60 a<<2: 240
```

How does this take place? Let us use the binary equivalent of 60, and perform the left shift by 2.

```
0011 1100  
<<  
2  
-----  
1111 0000
```

Convert the binary to integer. It is 240.

```
>>> int('11110000',2)  
240
```

## Python – Right Shift Operator (>>)

Right shift operator shifts least significant bits to left by the number on the right side of the ">>" symbol. Hence, "x >> 2" causes two bits of the binary representation of to left. Let us perform right shift on 60.

```
</>  
  
a=60  
print ("a:",a, "a>>2:", a>>2)
```

It will produce the following **output** –

```
a: 60 a>>2: 15
```

Manual right shift operation on 60 is shown below –

```
0011 1100  
>>  
2  
-----  
0000 1111
```

Use int() function to convert the above binary number to integer. It is 15.

```
>>> int('00001111',2)
15
```

## Python - Membership Operators

The membership operators in Python help us determine whether an item is present in a given container type object, or in other words, whether an item is a member of the given container type object.

Python has two membership operators: "in" and "not in". Both return a Boolean result. The result of "in" operator is opposite to that of "not in" operator.

You can use in operator to check whether a substring is present in a bigger string, any item is present in a list or tuple, or a sub-list or sub-tuple is included in a list or tuple.

In the following example, different substrings are checked whether they belong to the string var="TutorialsPoint". Python differentiates characters on the basis of their Unicode value. Hence "To" is not the same as "to". Also note that if the "in" operator returns True, the "not in" operator evaluates to False.

```
</> Open Compiler
```

```
var = "TutorialsPoint"
a = "P"
b = "tor"
c = "in"
d = "To"
print (a, "in", var, ":", a in var)
print (b, "not in", var, ":", b not in var)
print (c, "in", var, ":", c in var)
print (d, "not in", var, ":", d not in var)
```

It will produce the following **output** –

```
P in TutorialsPoint : True
tor not in TutorialsPoint : False
in in TutorialsPoint : True
To not in TutorialsPoint : True
```

You can use the "in/not in" operator to check the membership of an item in the list or tuple.

[Open Compiler](#)

```
</>

var = [10,20,30,40]
a = 20
b = 10
c = a-b
d = a/2
print (a, "in", var, ":", a in var)
print (b, "not in", var, ":", b not in var)
print (c, "in", var, ":", c in var)
print (d, "not in", var, ":", d not in var)
```

It will produce the following **output** –

```
20 in [10, 20, 30, 40] : True
10 not in [10, 20, 30, 40] : False
10 in [10, 20, 30, 40] : True
10.0 not in [10, 20, 30, 40] : False
```

In the last case, "d" is a float but still it compares to True with 10 (an **int**) in the list. Even if a number expressed in other formats like binary, octal or hexadecimal are given the membership operators tell if it is inside the sequence.

```
>>> 0x14 in [10, 20, 30, 40]
True
```

However, if you try to check if two successive numbers are present in a list or tuple, the in operator returns False. If the list/tuple contains the successive numbers as a sequence itself, then it returns True.

```
</>

var = (10,20,30,40)
a = 10
b = 20
print ((a,b), "in", var, ":", (a,b) in var)
var = ((10,20),30,40)
a = 10
b = 20
print ((a,b), "in", var, ":", (a,b) in var)
```

It will produce the following **output** –

```
(10, 20) in (10, 20, 30, 40) : False  
(10, 20) in ((10, 20), 30, 40) : True
```

Python's membership operators also work well with the set objects.

</>

Open Compiler

```
var = {10,20,30,40}  
a = 10  
b = 20  
print (b, "in", var, ":", b in var)  
var = {(10,20),30,40}  
a = 10  
b = 20  
print ((a,b), "in", var, ":", (a,b) in var)
```

It will produce the following **output** –

```
20 in {40, 10, 20, 30} : True  
(10, 20) in {40, 30, (10, 20)} : True
```

Use of in as well as not in operators with dictionary object is allowed. However, Python checks the membership only with the collection of keys and not values.

</>

Open Compiler

```
var = {1:10, 2:20, 3:30}  
a = 2  
b = 20  
print (a, "in", var, ":", a in var)  
print (b, "in", var, ":", b in var)
```

It will produce the following **output** –

```
2 in {1: 10, 2: 20, 3: 30} : True  
20 in {1: 10, 2: 20, 3: 30} : False
```

# Python - Identity Operators

Python has two identity operators `is` and `is not`. Both return opposite Boolean values. The "`in`" operator evaluates to True if both the operand objects share the same memory location. The memory location of the object can be obtained by the "`id()`" function. If the `id()` of both variables is same, the "`in`" operator returns True (as a consequence, `is not` returns False).

&lt;/&gt;

Open Compiler

```
a="TutorialsPoint"  
b=a  
print ("id(a), id(b):", id(a), id(b))  
print ("a is b:", a is b)  
print ("b is not a:", b is not a)
```

It will produce the following **output** –

```
id(a), id(b): 2739311598832 2739311598832  
a is b: True  
b is not a: False
```

The list and tuple objects differently, which might look strange in the first instance. In the following example, two lists "a" and "b" contain same items. But their `id()` differs.

&lt;/&gt;

Open Compiler

```
a=[1,2,3]  
b=[1,2,3]  
print ("id(a), id(b):", id(a), id(b))  
print ("a is b:", a is b)  
print ("b is not a:", b is not a)
```

It will produce the following **output** –

```
id(a), id(b): 1552612704640 1552567805568  
a is b: False  
b is not a: True
```

The list or tuple contains the memory locations of individual items only and not the items itself. Hence "a" contains the addresses of 10,20 and 30 integer objects in a certain location which may be different from that of "b".

```
print (id(a[0]), id(a[1]), id(a[2]))
print (id(b[0]), id(b[1]), id(b[2]))
```

It will produce the following **output** –

```
140734682034984 140734682035016 140734682035048
140734682034984 140734682035016 140734682035048
```

Because of two different locations of "a" and "b", the "**is**" operator returns False even if the two lists contain same numbers.

## Python - Comments

A comment in a computer program is a piece of text that is meant to be an explanatory or descriptive annotation in the source code and is not to be considered by compiler/interpreter while generating machine language code. Ample use of comments in source program makes it easier for everybody concerned to understand more about syntax, usage and logic of the algorithm etc.

In a Python script, the symbol # marks the beginning of comment line. It is effective till the end of line in the editor. If # is the first character of line, then entire line is a comment. If used in the middle of a line, text before it is a valid Python expression, while text following it is treated as comment.

```
# this is a comment
print ("Hello World")
print ("How are you?") #also a comment but after a statement.
```

In Python, there is no provision to write multi-line comment, or a block comment. (As in C/C++, where multiple lines inside /\* .. \*/ are treated as multi-line comment).

Each line should have the "#" symbol at the start to be marked as comment. Many Python IDEs have shortcuts to mark a block of statements as comment.

A triple quoted multi-line string is also treated as comment if it is not a docstring of a function or class.

```
...
First line in the comment
Second line in the comment
```

```
Third line in the comment  
...  
print ("Hello World")
```

## Python - User Input

Every computer application should have a provision to accept data from the user when it is running. This makes the application interactive. Depending on how it is developed, an application may accept the user input in the form of text entered in the console (**sys.stdin**), a graphical layout, or a web-based interface. In this chapter, we shall learn how Python accepts the user input from the console, and displays the output also on the console.

Python interpreter works in interactive and scripted mode. While the interactive mode is good for quick evaluations, it is less productive. For repeated execution of same set of instructions, scripted mode should be used.

Let us write a simple Python script to start with.

&lt;/&gt;

Open Compiler

```
#! /usr/bin/python3.11  
name = "Kiran"  
city = "Hyderabad"  
print ("Hello My name is", name)  
print ("I am from", city)
```

Save the above code as hello.py and run it from the command-line. Here's the output

```
C:\python311> python var1.py  
Hello My name is Kiran  
I am from Hyderabad
```

The program simply prints the values of the two variables in it. If you run the program repeatedly, the same output will be displayed every time. To use the program for another name and city, you can edit the code, change name to say "Ravi" and city to "Chennai". Every time you need to assign different value, you will have to edit the program, save and run, which is not the ideal way.

### input() Function

Obviously, you need some mechanism to assign different value to the variable in the runtime – while the program is running. Python's **input()** function does the same job.

Python's standard library has the `input()` function.

```
>>> var = input()
```

When the interpreter encounters it, it waits for the user to enter data from the standard input stream (keyboard) till the Enter key is pressed. The sequence of characters may be stored in a string variable for further use.

On reading the Enter key, the program proceeds to the next statement. Let us change our program to store the user input in name and city variables.

```
#! /usr/bin/python3.11
name = input()
city = input()
print ("Hello My name is", name)
print ("I am from ", city)
```

When you run, you will find the cursor waiting for user's input. Enter values for name and city. Using the entered data, the output will be displayed.

```
Ravi
Chennai
Hello My name is Ravi
I am from Chennai
```

Now, the variables are not assigned any specific value in the program. Every time you run, different values can be input. So, your program has become truly interactive.

Inside the `input()` function, you may give a **prompt** text, which will appear before the cursor when you run the code.

```
#! /usr/bin/python3.11
name = input("Enter your name : ")
city = input("enter your city : ")
print ("Hello My name is", name)
print ("I am from ", city)
```

When you run the program displays the prompt message, basically helping the user what to enter.

```
Enter your name: Praveen Rao  
enter your city: Bengaluru  
Hello My name is Praveen Rao  
I am from Bengaluru
```

## Numeric Input

Let us write a Python code that accepts width and height of a rectangle from the user and computes the area.

```
#!/usr/bin/python3.11  
width = input("Enter width : ")  
height = input("Enter height : ")  
area = width*height  
print ("Area of rectangle = ", area)
```

Run the program, and enter width and height.

```
Enter width: 20  
Enter height: 30  
Traceback (most recent call last):  
  File "C:\Python311\var1.py", line 5, in <module>  
    area = width*height  
TypeError: can't multiply sequence by non-int of type 'str'
```

Why do you get a **TypeError** here? The reason is, Python always read the user input as a string. Hence, `width="20"` and `height="30"` are the strings and obviously you cannot perform multiplication of two strings.

To overcome this problem, we shall use **int()**, another built-in function from Python's standard library. It converts a string object to an integer.

To accept an integer input from the user, read the input in a string, and type cast it to integer with `int()` function –

```
w= input("Enter width : ")  
width=int(w)  
h= input("Enter height : ")  
height=int(h)
```

You can combine the input and type cast statements in one –

```
#!/usr/bin/python3.11
width = int(input("Enter width : "))
height = int(input("Enter height : "))
area = width*height
print ("Area of rectangle = ", area)
```

Now you can input any integer value to the two variables in the program –

```
Enter width: 20
Enter height: 30
Area of rectangle = 600
```

Python's **float()** function converts a string into a float object. The following program accepts the user input and parses it to a float variable – rate, and computes the interest on an amount which is also input by the user.

```
#!/usr/bin/python3.11
amount = float(input("Enter Amount : "))
rate = float(input("Enter rate of interest : "))
interest = amount*rate/100
print ("Amount: ", amount, "Interest: ", interest)
```

The program ask user to enter amount and rate; and displays the result as follows –

```
Enter Amount: 12500
Enter rate of interest: 6.5
Amount: 12500.0 Interest: 812.5
```

## print() Function

Python's print() function is a built-in function. It is the most frequently used function, that displays value of Python expression given in parenthesis, on Python's console, or standard output (**sys.stdout**).

```
print ("Hello World ")
```

Any number of Python expressions can be there inside the parenthesis. They must be separated by comma symbol. Each item in the list may be any Python object, or a valid Python expression.

```
#!/usr/bin/python3.11
a = "Hello World"
```

```
b = 100
c = 25.50
d = 5+6j
print ("Message: a")
print (b, c, b-c)
print(pow(100, 0.5), pow(c,2))
```

The first call to print() displays a string literal and a string variable. The second prints value of two variables and their subtraction. The **pow()** function computes the square root of a number and square value of a variable.

Message Hello World

100 25.5 74.5  
10.0 650.25

If there are multiple comma separated objects in the print() function's parenthesis, the values are separated by a white space " ". To use any other character as a separator, define a **sep** parameter for the print() function. This parameter should follow the list of expressions to be printed.

In the following output of print() function, the variables are separated by comma.

```
</> Open Compiler

#!/usr/bin/python3.11
city="Hyderabad"
state="Telangana"
country="India"
print(city, state, country, sep=',')
```

The effect of `sep=','` can be seen in the result –

Hyderabad,Telangana,India

The print() function issues a newline character ('\n') at the end, by default. As a result, the output of the next print() statement appears in the next line of the console.

```
</> Open Compiler

city="Hyderabad"
state="Telangana"
```

```
print("City:", city)
print("State:", state)
```

Two lines are displayed as the output –

```
City: Hyderabad
State: Telangana
```

To make these two lines appear in the same line, define **end** parameter in the first print() function and set it to a whitespace string " ".

</>

Open Compiler

```
city="Hyderabad"
state="Telangana"
country="India"
print("City:", city, end=" ")
print("State:", state)
```

Output of both the print() functions appear in continuation.

```
City: Hyderabad State: Telangana
```

## Python - Numbers

Most of the times you work with numbers in whatever you do. Obviously, any computer application deals with numbers. Hence, programming languages, Python included, have built-in support to store and process numeric data.

In this chapter, we shall learn about properties of Python number types in detail.

Three number types, integers (**int**), floating point numbers (**float**) and **complex** numbers, are built into the Python interpreter. Python also has a built-in Boolean data type called **bool**. It can be treated as a sub-type of int type, since its two possible values True and False represent the integers 1 and 0 respectively.

### Python – Integers

In Python, any number without the provision to store a fractional part is an integer. (Note that if the fractional part in a number is 0, it doesn't mean that it is an integer. For example a number 10.0 is not an integer, it is a float with 0 fractional part whose numeric

value is 10.) An integer can be zero, positive or a negative whole number. For example, 1234, 0, -55.

There are three ways to form an integer object. With literal representation, any expression evaluating to an integer, and using **int()** function.

Literal is a notation used to represent a constant directly in the source code. For example –

```
>>> a =10
```

However, look at the following assignment of the integer variable c.

```
a=10  
b=20  
c=a+b  
print ("a:", a, "type:", type(a))
```

It will produce the following **output** –

```
a: 10 type: <class 'int'>
```

Here, c is indeed an integer variable, but the expression a+b is evaluated first, and its value is indirectly assigned to c.

The third method of forming an integer object is with the return value of int() function. It converts a floating point number or a string in an integer.

```
>>> a=int(10.5)  
>>> b=int("100")
```

You can represent an integer as a binary, octal or Hexa-decimal number. However, internally the object is stored as an integer.

## Binary Numbers

A number consisting of only the binary digits (1 and 0) and prefixed with **0b** is a binary number. If you assign a binary number to a variable, it still is an int variable.

To represent an integer in binary form, store it directly as a literal, or use int() function, in which the base is set to 2

```
</>
```

Open Compiler

```
a=0b101  
print ("a:",a, "type:",type(a))  
b=int("0b101011",2)  
print ("b:",b, "type:",type(b))
```

It will produce the following **output** –

```
a: 5 type: <class 'int'>  
b: 43 type: <class 'int'>
```

There is also a **bin()** function in Python. It returns a binary string equivalent of an integer.

```
</>  
  
a=43  
b=bin(a)  
print ("Integer:",a, "Binary equivalent:",b)
```

[Open Compiler](#)

It will produce the following **output** –

```
Integer: 43 Binary equivalent: 0b101011
```

## Octal Numbers

An octal number is made up of digits 0 to 7 only. In order to specify that the integer uses octal notation, it needs to be prefixed by **0o** (lowercase O) or **0O** (uppercase O). A literal representation of octal number is as follows –

```
a=00107  
print (a, type(a))
```

It will produce the following **output** –

```
71 <class 'int'>
```

Note that the object is internally stored as integer. Decimal equivalent of octal number 107 is 71.

Since octal number system has 8 symbols (0 to 7), its base is 7. Hence, while using int() function to convert an octal string to integer, you need to set the base argument to 8.

&lt;/&gt;

[Open Compiler](#)

```
a=int('20',8)
print (a, type(a))
```

It will produce the following **output** –

```
16 <class 'int'>
```

Decimal equivalent of octal 30 is 16.

In the following code, two int objects are obtained from octal notations and their addition is performed.

&lt;/&gt;

[Open Compiler](#)

```
a=0056
print ("a:",a, "type:",type(a))
b=int("0031",8)
print ("b:",b, "type:",type(b))
c=a+b
print ("addition:", c)
```

It will produce the following **output** –

```
a: 46 type: <class 'int'>
b: 25 type: <class 'int'>
addition: 71
```

To obtain the octal string for an integer, use **oct()** function.

```
a=oct(71)
print (a, type(a))
```

## Hexa-decimal Numbers

As the name suggests, there are 16 symbols in the Hexadecimal number system. They are 0-9 and A to F. The first 10 digits are same as decimal digits. The alphabets A, B, C, D, E and F are equivalents of 11, 12, 13, 14, 15, and 16 respectively. Upper or lower cases may be used for these letter symbols.

For the literal representation of an integer in Hexadecimal notation, prefix it by **0x** or **0X**.

&lt;/&gt;

[Open Compiler](#)

```
a=0XA2
print (a, type(a))
```

It will produce the following **output** –

```
162 <class 'int'>
```

To convert a Hexadecimal string to integer, set the base to 16 in the **int()** function.

```
a=int('0X1e', 16)
print (a, type(a))
```

Try out the following code snippet. It takes a Hexadecimal string, and returns the integer.

&lt;/&gt;

[Open Compiler](#)

```
num_string = "A1"
number = int(num_string, 16)
print ("Hexadecimal:", num_string, "Integer:", number)
```

It will produce the following **output** –

```
Hexadecimal: A1 Integer: 161
```

However, if the string contains any symbol apart from the Hexadecimal symbol chart (for example X001), it raises following error –

Traceback (most recent call last):

```
File "C:\Python311\var1.py", line 4, in <module>
    number = int(num_string, 16)
ValueError: invalid literal for int() with base 16: 'X001'
```

Python's standard library has **hex()** function, with which you can obtain a hexadecimal equivalent of an integer.

&lt;/&gt;

[Open Compiler](#)

```
a=hex(161)  
print (a, type(a))
```

It will produce the following **output** –

```
0xa1 <class 'str'>
```

Though an integer can be represented as binary or octal or hexadecimal, internally it is still integer. So, when performing arithmetic operation, the representation doesn't matter.

```
</>
```

Open Compiler

```
a=10 #decimal  
b=0b10 #binary  
c=0010 #octal  
d=0XA #Hexadecimal  
e=a+b+c+d  
print ("addition:", e)
```

It will produce the following **output** –

```
addition: 30
```

## Python – Floating Point Numbers

A floating point number has an integer part and a fractional part, separated by a decimal point symbol (.). By default, the number is positive, prefix a dash (-) symbol for a negative number.

A floating point number is an object of Python's float class. To store a float object, you may use a literal notation, use the value of an arithmetic expression, or use the return value of float() function.

Using literal is the most direct way. Just assign a number with fractional part to a variable. Each of the following statements declares a float object.

```
>>> a=9.99  
>>> b=0.999  
>>> c=-9.99  
>>> d=-0.999
```

In Python, there is no restriction on how many digits after the decimal point can a floating point number have. However, to shorten the representation, the **E** or **e** symbol is used. E stands for Ten raised to. For example, E4 is 10 raised to 4 (or 4<sup>th</sup> power of 10), e-3 is 10 raised to -3.

In scientific notation, number has a coefficient and exponent part. The coefficient should be a float greater than or equal to 1 but less than 10. Hence, 1.23E+3, 9.9E-5, and 1E10 are the examples of floats with scientific notation.

```
>>> a=1E10  
>>> a  
10000000000.0  
>>> b=9.90E-5  
>>> b  
9.9e-05  
>>> 1.23E3  
1230.0
```

The second approach of forming a float object is indirect, using the result of an expression. Here, the quotient of two floats is assigned to a variable, which refers to a float object.

```
</>  
  
a=10.33  
b=2.66  
c=a/b  
print ("c:", c, "type", type(c))
```

Open Compiler

It will produce the following **output** –

```
c: 3.8834586466165413 type <class 'float'>
```

Python's `float()` function returns a float object, parsing a number or a string if it has the appropriate contents. If no arguments are given in the parenthesis, it returns 0.0, and for an **int** argument, fractional part with 0 is added.

```
>>> a=float()  
>>> a  
0.0  
>>> a=float(10)
```

```
>>> a  
10.0
```

Even if the integer is expressed in binary, octal or hexadecimal, the float() function returns a float with fractional part as 0.

```
</>  
  
a=float(0b10)  
b=float(0010)  
c=float(0xA)  
print (a,b,c, sep=",")
```

[Open Compiler](#)

It will produce the following **output** –

```
2.0,8.0,10.0
```

The float() function retrieves a floating point number out of a string that encloses a float, either in standard decimal point format, or having scientific notation.

```
</>  
  
a=float("-123.54")  
b=float("1.23E04")  
print ("a=",a,"b=",b)
```

[Open Compiler](#)

It will produce the following **output** –

```
a= -123.54 b= 12300.0
```

In mathematics, infinity is an abstract concept. Physically, infinitely large number can never be stored in any amount of memory. For most of the computer hardware configurations, however, a very large number with 400th power of 10 is represented by Inf. If you use "Infinity" as argument for float() function, it returns Inf.

```
</>  
  
a=1.00E400  
print (a, type(a))
```

[Open Compiler](#)

```
a=float("Infinity")
print (a, type(a))
```

It will produce the following **output** –

```
inf <class 'float'>
inf <class 'float'>
```

One more such entity is Nan (stands for Not a Number). It represents any value that is undefined or not representable.

```
>>> a=float('Nan')
>>> a
Nan
```

## Python – Complex Numbers

In this section, we shall know in detail about Complex data type in Python. Complex numbers find their applications in mathematical equations and laws in electromagnetism, electronics, optics, and quantum theory. Fourier transforms use complex numbers. They are Used in calculations with wavefunctions, designing filters, signal integrity in digital electronics, radio astronomy, etc.

A complex number consists of a real part and an imaginary part, separated by either "+" or "-". The real part can be any floating point (or itself a complex number) number. The imaginary part is also a float/complex, but multiplied by an imaginary number.

In mathematics, an imaginary number "i" is defined as the square root of -1 ( $\sqrt{-1}$ ). Therefore, a complex number is represented as " $x+yi$ ", where  $x$  is the real part, and " $y$ " is the coefficient of imaginary part.

Quite often, the symbol "j" is used instead of "I" for the imaginary number, to avoid confusion with its usage as current in theory of electricity. Python also uses "j" as the imaginary number. Hence, " $x+yj$ " is the representation of complex number in Python.

Like int or float data type, a complex object can be formed with literal representation or using complex() function. All the following statements form a complex object.

```
>>> a=5+6j
>>> a
(5+6j)
>>> type(a)
<class 'complex'>
```

```
>>> a=2.25-1.2j
>>> a
(2.25-1.2j)
>>> type(a)
<class 'complex'>
>>> a=1.01E-2+2.2e3j
>>> a
(0.0101+2200j)
>>> type(a)
<class 'complex'>
```

Note that the real part as well as the coefficient of imaginary part have to be floats, and they may be expressed in standard decimal point notation or scientific notation.

Python's **complex()** function helps in forming an object of complex type. The function receives arguments for real and imaginary part, and returns the complex number.

There are two versions of complex() function, with two arguments and with one argument. Use of complex() with two arguments is straightforward. It uses first argument as real part and second as coefficient of imaginary part.

&lt;/&gt;

Open Compiler

```
a=complex(5.3,6)
b=complex(1.01E-2, 2.2E3)
print ("a:", a, "type:", type(a))
print ("b:", b, "type:", type(b))
```

It will produce the following **output** –

```
a: (5.3+6j) type: <class 'complex'>
b: (0.0101+2200j) type: <class 'complex'>
```

In the above example, we have used x and y as float parameters. They can even be of complex data type.

&lt;/&gt;

Open Compiler

```
a=complex(1+2j, 2-3j)
print (a, type(a))
```

It will produce the following **output** –

```
(4+4j) <class 'complex'>
```

Surprised by the above example? Put "x" as  $1+2j$  and "y" as  $2-3j$ . Try to perform manual computation of " $x+yj$ " and you'll come to know.

```
complex(1+2j, 2-3j)
=(1+2j)+(2-3j)*j
=1+2j +2j+3
=4+4j
```

If you use only one numeric argument for `complex()` function, it treats it as the value of real part; and imaginary part is set to 0.

```
</>
```

[Open Compiler](#)

```
a=complex(5.3)
print ("a:", a, "type:", type(a))
```

It will produce the following **output** –

```
a: (5.3+0j) type: <class 'complex'>
```

The `complex()` function can also parse a string into a complex number if its only argument is a string having complex number representation.

In the following snippet, user is asked to input a complex number. It is used as argument. Since Python reads the input as a string, the function extracts the complex object from it.

```
</>
```

[Open Compiler](#)

```
a= "5.5+2.3j"
b=complex(a)
print ("Complex number:", b)
```

It will produce the following **output** –

```
Complex number: (5.5+2.3j)
```

Python's built-in `complex` class has two attributes `real` and `imag` – they return the real and coefficient of imaginary part from the object.

&lt;/&gt;

[Open Compiler](#)

```
a=5+6j
print ("Real part:", a.real, "Coefficient of Imaginary part:", a.imag)
```

It will produce the following **output** –

Real part: 5.0 Coefficient of Imaginary part: 6.0

The complex class also defines a `conjugate()` method. It returns another complex number with the sign of imaginary component reversed. For example, conjugate of  $x+yj$  is  $x-yj$ .

```
>>> a=5-2.2j
>>> a.conjugate()
(5+2.2j)
```

## Python - Booleans

In Python, **bool** is a sub-type of `int` type. A `bool` object has two possible values, and it is initialized with Python keywords, `True` and `False`.

```
>>> a=True
>>> b=False
>>> type(a), type(b)
(<class 'bool'>, <class 'bool'>)
```

A `bool` object is accepted as argument to type conversion functions. With `True` as argument, the `int()` function returns 1, `float()` returns 1.0; whereas for `False`, they return 0 and 0.0 respectively. We have a one argument version of `complex()` function.

If the argument is a complex object, it is taken as real part, setting the imaginary coefficient to 0.

&lt;/&gt;

[Open Compiler](#)

```
a=int(True)
print ("bool to int:", a)
a=float(False)
print ("bool to float:", a)
```

```
a=complex(True)
print ("bool to complex:", a)
```

On running this code, you will get the following **output** –

```
bool to int: 1
bool to float: 0.0
bool to complex: (1+0j)
```

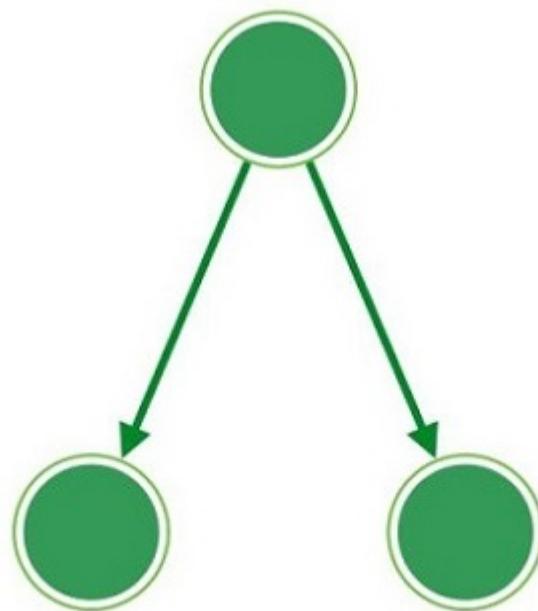
## Python - Control Flow

By default, the instructions in a computer program are executed in a sequential manner, from top to bottom, or from start to end. However, such sequentially executing programs can perform only simplistic tasks. We would like the program to have a decision-making ability, so that it performs different steps depending on different conditions.

Most programming languages including Python provide functionality to control the flow of execution of instructions. Normally, there are two type of control flow statements.

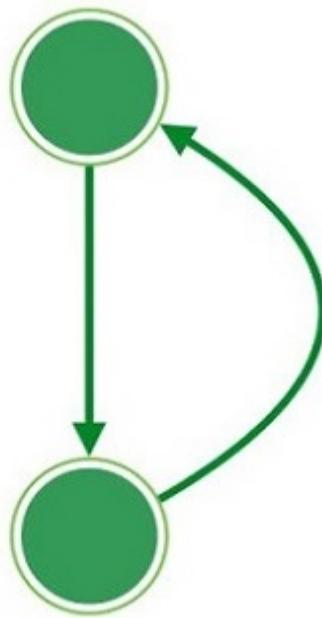
**Decision-making** – The program is able to decide which of the alternative group of instructions to be executed, depending on value of a certain Boolean expression.

The following diagram illustrates how decision-making statements work –



**Looping or Iteration** – Most of the processes require a group of instructions to be repeatedly executed. In programming terminology, it is called a **loop**. Instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

The following diagram illustrates how the looping works –



If the control goes back unconditionally, it forms an infinite loop which is not desired as the rest of the code would never get executed.

In a conditional loop, the repeated iteration of block of statements goes on till a certain condition is met.

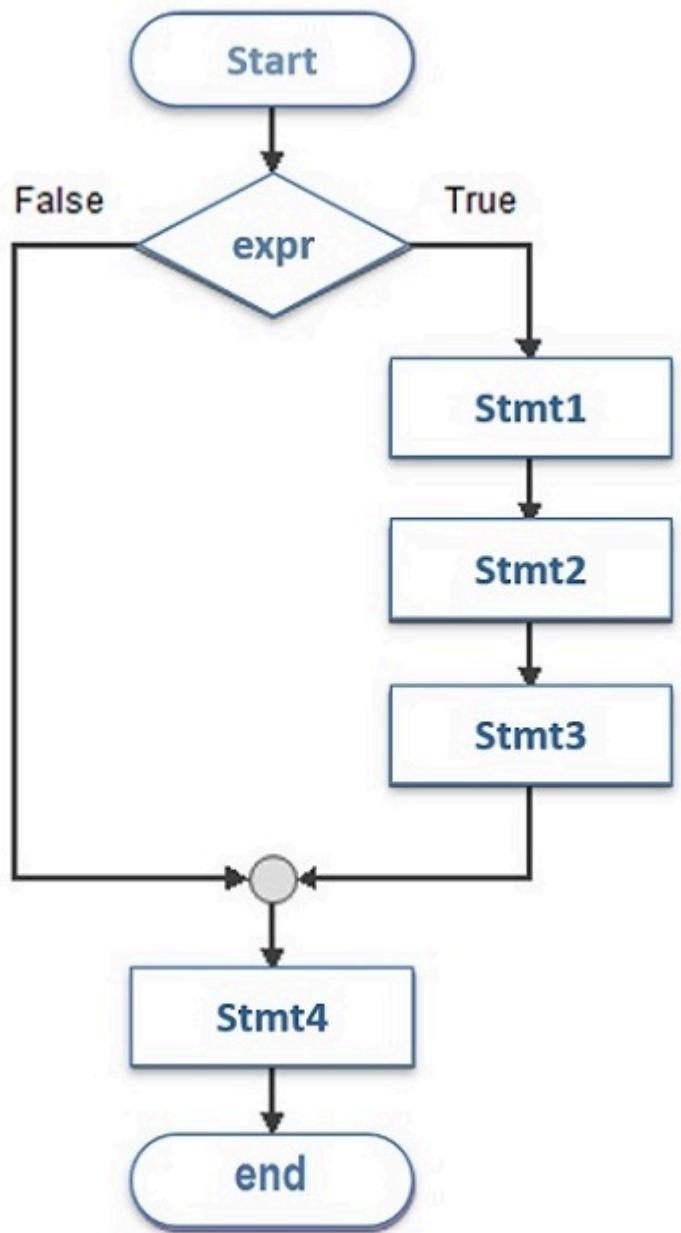
## Python - Decision Making

Python's decision making functionality is in its keywords – if, else and elif. The if keyword requires a boolean expression, followed by colon symbol.

The colon (:) symbol starts an indented block. The statements with the same level of indentation are executed if the boolean expression in if statement is True. If the expression is not True (False), the interpreter bypasses the indented block and proceeds to execute statements at earlier indentation level.

### Python – The if Statement

The following flowchart illustrates how Python **if statement** works –



## Syntax

The logic in the above flowchart is expressed by the following syntax –

```
if expr==True:  
    stmt1  
    stmt2  
    stmt3  
    ..  
    ..  
    Stmt4
```

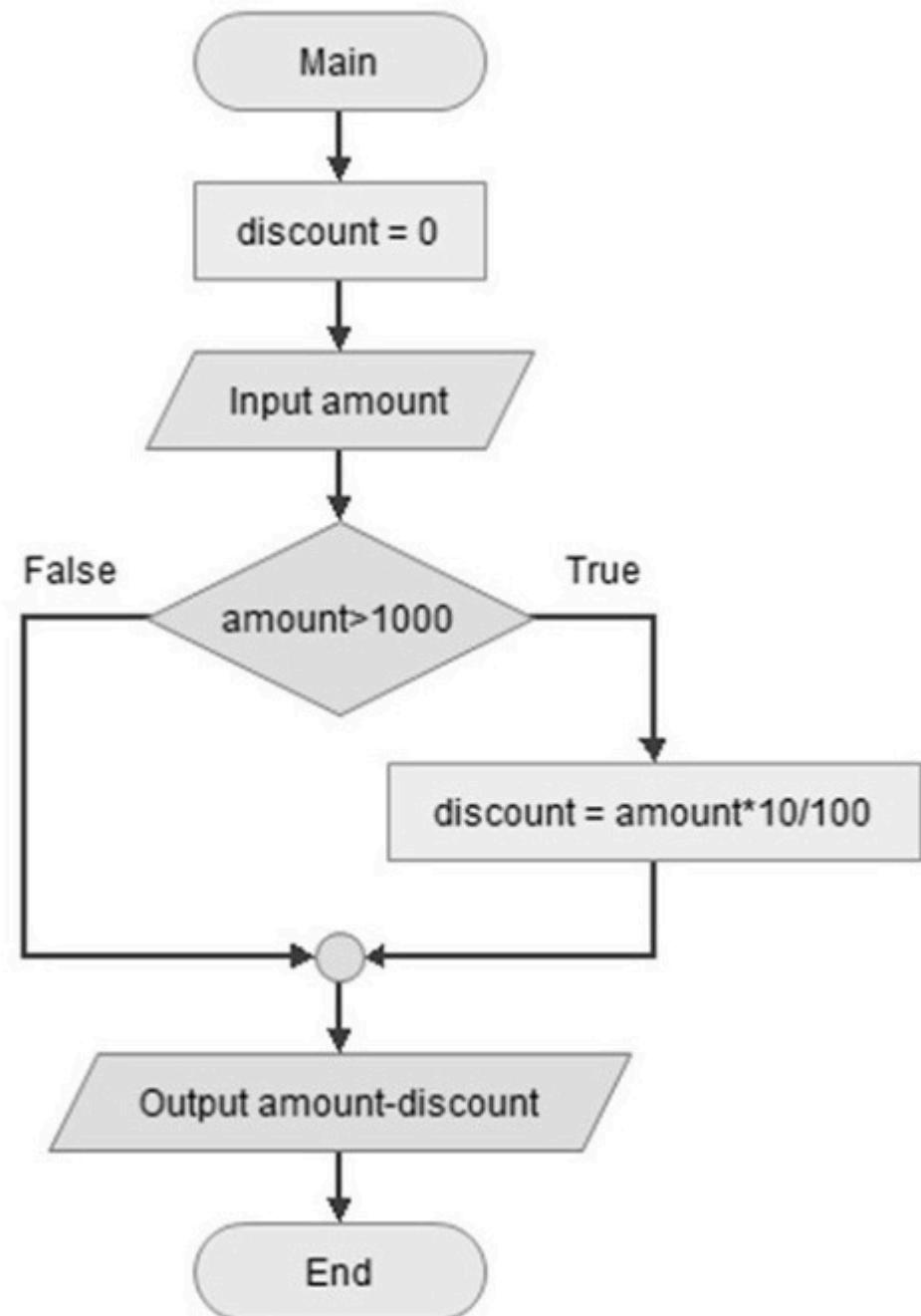
The **if** statement is similar to that of other languages. The **if** statement contains a boolean expression using which the data is compared and a decision is made based on the result of

the comparison.

If the boolean expression evaluates to True, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the ":" symbol. If boolean expression evaluates to False, then the first set of code after the end of block is executed.

## Example

Let us consider an example of a customer entitled to 10% discount if his purchase amount is  $>1000$ ; if not, no discount applicable. This flowchart shows the process.



In Python, we first set a discount variable to 0 and accept the amount as input from user.

Then comes the conditional statement if amt>1000. Put : symbol that starts conditional block wherein discount applicable is calculated. Obviously, discount or not, next statement by default prints amount-discount. If applied, it will be subtracted, if not it is 0.

```
</>  
  
discount = 0  
amount = 1200  
if amount > 1000:  
    discount = amount * 10 / 100  
print("amount = ",amount - discount)
```

[Open Compiler](#)

Here the amount is 1200, hence discount 120 is deducted. On executing the code, you will get the following **output** –

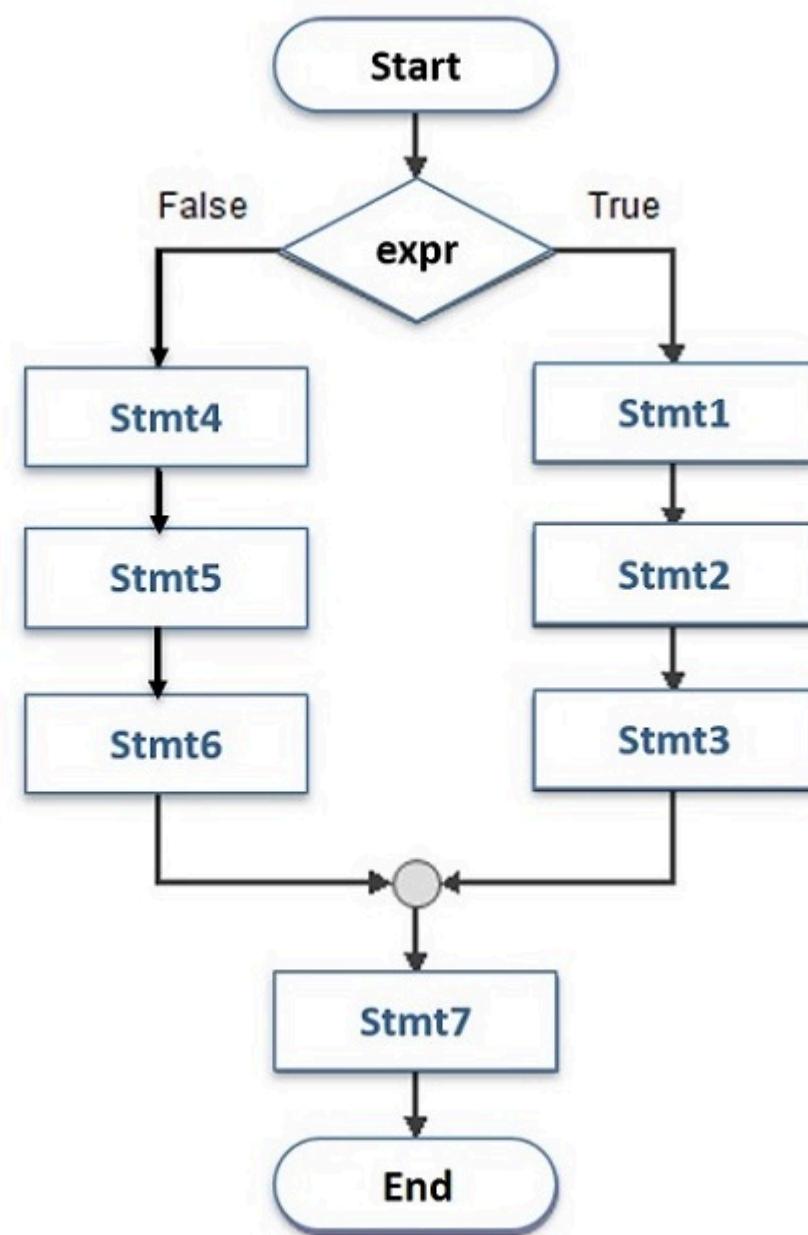
```
amount = 1080.0
```

Change the variable amount to 800, and run the code again. This time, no discount is applicable. And, you will get the following output –

```
amount = 800
```

## Python - The if-else Statement

Along with the **if** statement, **else** keyword can also be optionally used. It provides an alternate block of statements to be executed if the Boolean expression (in if statement) is not true. this flowchart shows how else block is used.



If the expr is True, block of stmt1,2,3 is executed then the default flow continues with stmt7. However, the If expr is False, block stmt4,5,6 runs then the default flow continues.

## Syntax

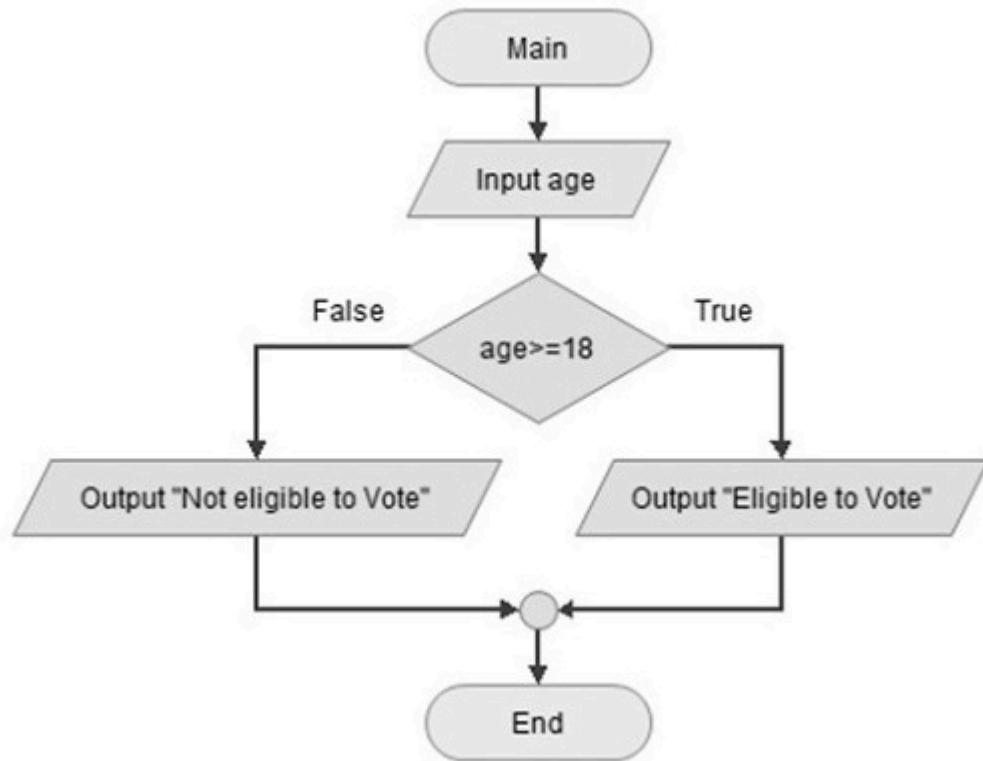
Python implementation of the above flowchart is as follows –

```

if expr==True:
    stmt1
    stmt2
    stmt3
else:
    stmt4
    stmt5
    stmt6
stmt7
  
```

## Example

Let us understand the use of else clause with following example. The variable age can take different values. If the expression "age > 18" is true, message you are eligible to vote is displayed otherwise not eligible message should be displayed. Following flowchart illustrates this logic.



Its Python implementation is simple.

```

</>
Open Compiler

age=25
print ("age: ", age)
if age>=18:
    print ("eligible to vote")
else:
    print ("not eligible to vote")
  
```

To begin, set the integer variable "age" to 25.

Then use the **if** statement with "age>18" expression followed by ":" which starts a block; this will come in action if "age>=18" is true.

To provide **else** block, use "else:" the ensuing indented block containing message **not eligible** will be in action when "age>=18" is false.

On executing this code, you will get the following **output** –

```
age: 25
eligible to vote
```

To test the the **else** block, change the **age** to 12, and run the code again.

```
age: 12
not eligible to vote
```

## Python – elif Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else** statement, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement; there can be an arbitrary number of **elif** statements following an **if**.

## Syntax

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

## Example

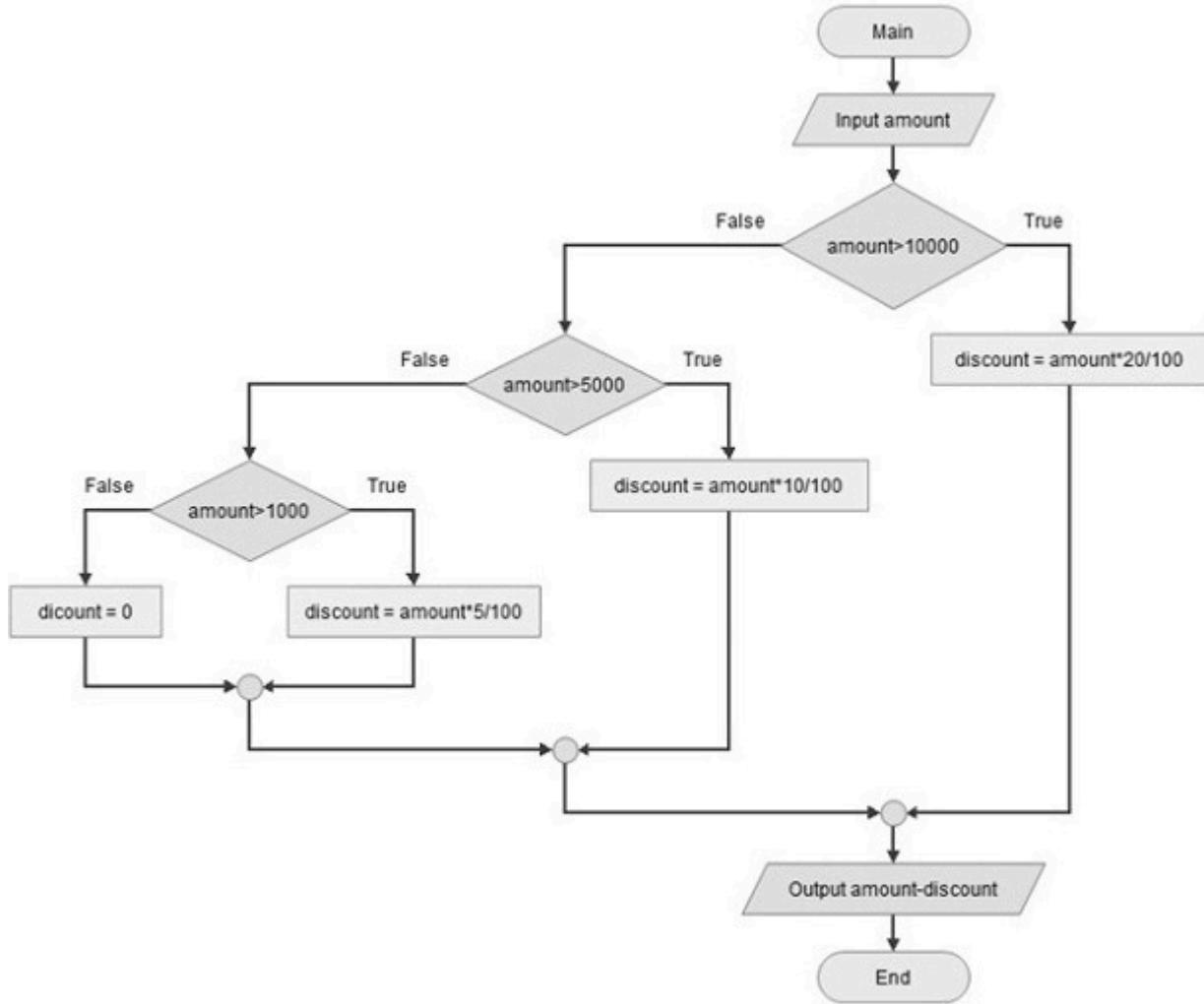
Let us understand how **elif** works, with the help of following example.

The discount structure used in an earlier example is modified to different slabs of discount

–

- 20% on amount exceeding 10000,
- 10% for amount between 5-10000,
- 5% if it is between 1 to 5000.
- no discount if amount<1000

The following flowchart illustrates these conditions –



## Example

We can write a Python code for the above logic with **if-else** statements –

```

amount = int(input('Enter amount: '))
if amount > 10000:
    discount = amount * 20 / 100
else:
    if amount > 5000:
        discount = amount * 10 / 100
    else:
        discount = 0
print(amount - discount)
  
```

```
if amount > 1000:  
    discount = amount * 5 / 100  
else:  
    dicount = 0  
print('amount: ',amount - discount)
```

While the code will work perfectly ok, if you look at the increasing level of indentation at each if and else statement, it will become difficult to manage if there are still more conditions.

The **elif** statement makes the code easy to read and comprehend.

**Elif** is short for "else if". It allows the logic to be arranged in a cascade of **elif** statements after the first if statement. If the first **if** statement evaluates to false, subsequent elif statements are evaluated one by one and comes out of the cascade if any one is satisfied.

Last in the cascade is the **else** block which will come in picture when all preceding if/elif conditions fail.

```
amount = 800  
print('amount = ',amount)  
if amount > 10000:  
    discount = amount * 20 / 100  
elif amount > 5000:  
    discount = amount * 10 / 100  
elif amount > 1000:  
    discount = amount * 5 / 100  
else:  
    discount=0  
  
print('payable amount = ',amount - discount)
```

Set **amount** to test all possible conditions: 800, 2500, 7500 and 15000. The **outputs** will vary accordingly –

```
amount: 800  
payable amount = 800  
amount: 2500  
payable amount = 2375.0  
amount: 7500  
payable amount = 6750.0  
amount: 15000  
payable amount = 12000.0
```

## Nested If Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

## Syntax

The syntax of the nested **if...elif...else** construct will be like this –

```
if expression1:  
    statement(s)  
    if expression2:  
        statement(s)  
    elif expression3:  
        statement(s)  
    else:  
        statement(s)  
    elif expression4:  
        statement(s)  
    else:  
        statement(s)
```

## Example

Now let's take a Python code to understand how it works –

```
</> Open Compiler  
  
#!/usr/bin/python3  
num=8  
print ("num = ",num)  
if num%2==0:  
    if num%3==0:  
        print ("Divisible by 3 and 2")  
    else:  
        print ("divisible by 2 not divisible by 3")  
else:  
    if num%3==0:  
        print ("divisible by 3 not divisible by 2")
```

```
else:  
    print ("not Divisible by 2 not divisible by 3")
```

When the above code is executed, it produces the following **output** –

```
num = 8  
divisible by 2 not divisible by 3  
num = 15  
divisible by 3 not divisible by 2  
num = 12  
Divisible by 3 and 2  
num = 5  
not Divisible by 2 not divisible by 3
```

## Python - MatchCase Statement

Before its 3.10 version, Python lacked a feature similar to switch-case in C or C++. In Python 3.10, a pattern matching technique called match-case has been introduced, which is similar to the "switch case" construct.

A **match** statement takes an expression and compares its value to successive patterns given as one or more case blocks. The usage is more similar to pattern matching in languages like Rust or Haskell than a switch statement in C or C++. Only the first pattern that matches gets executed. It is also possible to extract components (sequence elements or object attributes) from the value into variables.

### Syntax

The basic usage of match-case is to compare a variable against one or more values.

```
match variable_name:  
    case 'pattern 1' : statement 1  
    case 'pattern 2' : statement 2  
    ...  
    case 'pattern n' : statement n
```

### Example

The following code has a function named `weekday()`. It receives an integer argument, matches it with all possible weekday number values, and returns the corresponding name of day.

```
</>

def weekday(n):
    match n:
        case 0: return "Monday"
        case 1: return "Tuesday"
        case 2: return "Wednesday"
        case 3: return "Thursday"
        case 4: return "Friday"
        case 5: return "Saturday"
        case 6: return "Sunday"
        case _: return "Invalid day number"
print (weekday(3))
print (weekday(6))
print (weekday(7))
```

## Output

On executing, this code will produce the following output –

```
Thursday
Sunday
Invalid day number
```

The last case statement in the function has "\_" as the value to compare. It serves as the wildcard case, and will be executed if all other cases are not true.

## Combined Cases

Sometimes, there may be a situation where for more than one cases, a similar action has to be taken. For this, you can combine cases with the OR operator represented by "|" symbol.

## Example

```
</>

def access(user):
    match user:
        case "admin" | "manager": return "Full access"
        case "Guest": return "Limited access"
        case _: return "No access"
```

```
print (access("manager"))
print (access("Guest"))
print (access("Ravi"))
```

## Output

The above code defines a function named `access()` and has one string argument, representing the name of the user. For admin or manager user, the system grants full access; for Guest, the access is limited; and for the rest, there's no access.

Full access

Limited access

No access

## List as the Argument

Since Python can match the expression against any literal, you can use a list as a case value. Moreover, for variable number of items in the list, they can be parsed to a sequence with "\*" operator.

## Example

</>

Open Compiler

```
def greeting(details):
    match details:
        case [time, name]:
            return f'Good {time} {name}!'
        case [time, *names]:
            msg=''
            for name in names:
                msg+=f'Good {time} {name}!\n'
            return msg

    print (greeting(["Morning", "Ravi"]))
    print (greeting(["Afternoon", "Guest"]))
    print (greeting(["Evening", "Kajal", "Praveen", "Lata"]))
```

## Output

On executing, this code will produce the following output –

Good Morning Ravi!  
Good Afternoon Guest!  
Good Evening Kajal!  
Good Evening Praveen!  
Good Evening Lata!

## Using "if" in "Case" Clause

Normally Python matches an expression against literal cases. However, it allows you to include if statement in the case clause for conditional computation of match variable.

In the following example, the function argument is a list of amount and duration, and the interest is to be calculated for amount less than or more than 10000. The condition is included in the **case** clause.

### Example

&lt;/&gt;

Open Compiler

```
def intr(details):
    match details:
        case [amt, duration] if amt<10000:
            return amt*10*duration/100
        case [amt, duration] if amt>=10000:
            return amt*15*duration/100
    print ("Interest = ", intr([5000,5]))
    print ("Interest = ", intr([15000,3]))
```

### Output

On executing, this code will produce the following output –

Interest = 2500.0  
Interest = 6750.0

## Python - The for Loop

The **for** loop in Python has the ability to iterate over the items of any sequence, such as a list or a string.

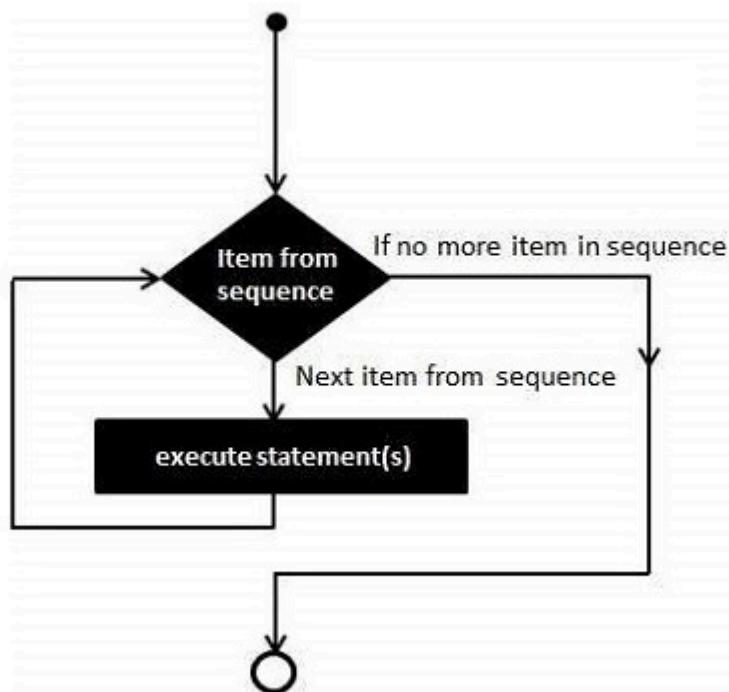
## Syntax

```
for iterating_var in sequence:
    statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item (at 0th index) in the sequence is assigned to the iterating variable `iterating_var`.

Next, the `statements` block is executed. Each item in the list is assigned to `iterating_var`, and the `statement(s)` block is executed until the entire sequence is exhausted.

The following flow diagram illustrates the working of **for** loop –



Since the loop is executed for each member element in a sequence, there is no need for explicit verification of Boolean expression controlling the loop (as in **while** loop).

The sequence objects such as list, tuple or string are called **iterables**, as the for loop iterates through the collection. Any iterator object can be iterated by the **for** loop.

The view objects returned by `items()`, `keys()` and `values()` methods of dictionary are also iterables, hence we can run a **for** loop with them.

Python's built-in `range()` function returns an iterator object that streams a sequence of numbers. We can run a for loop with range.

## Using "for" with a String

A string is a sequence of Unicode letters, each having a positional index. The following example compares each character and displays if it is not a vowel ('a', 'e', 'I', 'o' or 'u')

## Example

```
</> Open Compiler  
  
zen = '''  
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
...  
  
for char in zen:  
    if char not in 'aeiou':  
        print (char, end='')
```

## Output

On executing, this code will produce the following output –

```
Btfl s bttr thn gly.  
Explct s bttr thn mplct.  
Smpl s bttr thn cmplx.  
Cmplx s bttr thn cmplctd.
```

## Using "for" with a Tuple

Python's tuple object is also an indexed sequence, and hence we can traverse its items with a **for** loop.

## Example

In the following example, the **for** loop traverses a tuple containing integers and returns the total of all numbers.

```
</> Open Compiler  
  
numbers = (34,54,67,21,78,97,45,44,80,19)  
total = 0  
for num in numbers:  
    total+=num  
print ("Total =", total)
```

## Output

On executing, this code will produce the following output –

```
Total = 539
```

## Using "for" with a List

Python's list object is also an indexed sequence, and hence we can traverse its items with a **for** loop.

### Example

In the following example, the for loop traverses a list containing integers and prints only those which are divisible by 2.

```
</> Open Compiler  
numbers = [34, 54, 67, 21, 78, 97, 45, 44, 80, 19]  
total = 0  
for num in numbers:  
    if num%2 == 0:  
        print (num)
```

## Output

On executing, this code will produce the following output –

```
34  
54  
78  
44  
80
```

## Using "for" with a Range Object

Python's built-in range() function returns a range object. Python's range object is an iterator which generates an integer with each iteration. The object contains integers from start to stop, separated by step parameter.

## Syntax

The range() function has the following syntax –

```
range(start, stop, step)
```

## Parameters

- **Start** – Starting value of the range. Optional. Default is 0
- **Stop** – The range goes upto stop-1
- **Step** – Integers in the range increment by the step value. Option, default is 1.

## Return Value

The range() function returns a range object. It can be parsed to a list sequence.

## Example

```
</> Open Compiler  
  
numbers = range(5)  
...  
start is 0 by default,  
step is 1 by default,  
range generated from 0 to 4  
...  
print (list(numbers))  
# step is 1 by default, range generated from 10 to 19  
numbers = range(10,20)  
print (list(numbers))  
# range generated from 1 to 10 increment by step of 2  
numbers = range(1, 10, 2)  
print (list(numbers))
```

## Output

On executing, this code will produce the following output –

```
[0, 1, 2, 3, 4]  
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
[1, 3, 5, 7, 9]
```

## Example

Once we obtain the range, we can use the **for** loop with it.

```
</>
```

Open Compiler

```
for num in range(5):
    print (num, end=' ')
print()
for num in range(10,20):
    print (num, end=' ')
print()
for num in range(1, 10, 2):
    print (num, end=' ')
```

## Output

On executing, this code will produce the following output –

```
0 1 2 3 4
10 11 12 13 14 15 16 17 18 19
1 3 5 7 9
```

## Example: Factorial of a Number

Factorial is a product of all numbers from 1 to that number say n. It can also be defined as product of 1, 2, up to n.

```
Factorial of a number n! = 1 * 2 * . . . . * n
```

We use the range() function to get the sequence of numbers from 1 to n-1 and perform cumulative multiplication to get the factorial value.

```
</>
```

Open Compiler

```
fact=1
N = 5
for x in range(1, N+1):
```

```
fact=fact*x  
print ("factorial of {} is {}".format(N, fact))
```

## Output

On executing, this code will produce the following output –

```
factorial of 5 is 120
```

In the above program, change the value of N to obtain factorial value of different numbers.

## Using "for" Loop with Sequence Index

To iterate over a sequence, we can obtain the list of indices using the range() function

```
Indices = range(len(sequence))
```

We can then form a **for** loop as follows:

```
</>
```

Open Compiler

```
numbers = [34,54,67,21,78]  
indices = range(len(numbers))  
for index in indices:  
    print ("index:",index, "number:",numbers[index])
```

On executing, this code will produce the following **output** –

```
index: 0 number: 34  
index: 1 number: 54  
index: 2 number: 67  
index: 3 number: 21  
index: 4 number: 78
```

## Using "for" with Dictionaries

Unlike a list, tuple or a string, dictionary data type in Python is not a sequence, as the items do not have a positional index. However, traversing a dictionary is still possible with

different techniques.

Running a simple **for** loop over the dictionary object traverses the keys used in it.

```
</>
```

[Open Compiler](#)

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers:  
    print (x)
```

On executing, this code will produce the following **output** –

```
10  
20  
30  
40
```

Once we are able to get the key, its associated value can be easily accessed either by using square brackets operator or with the **get()** method. Take a look at the following example –

```
</>
```

[Open Compiler](#)

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers:  
    print (x, ":", numbers[x])
```

It will produce the following **output** –

```
10 : Ten  
20 : Twenty  
30 : Thirty  
40 : Forty
```

The **items()**, **keys()** and **values()** methods of **dict** class return the view objects **dict\_items**, **dict\_keys** and **dict\_values** respectively. These objects are iterators, and hence we can run a for loop over them.

The **dict\_items** object is a list of key-value tuples over which a for loop can be run as follows –

```
</>  
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers.items():  
    print (x)
```

It will produce the following **output** –

```
(10, 'Ten')  
(20, 'Twenty')  
(30, 'Thirty')  
(40, 'Forty')
```

Here, "x" is the tuple element from the dict\_items iterator. We can further unpack this tuple in two different variables. Check the following code –

```
</>  
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x,y in numbers.items():  
    print (x,":", y)
```

It will produce the following **output** –

```
10 : Ten  
20 : Twenty  
30 : Thirty  
40 : Forty
```

Similarly, the collection of keys in dict\_keys object can be iterated over. Take a look at the following example –

```
</>  
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers.keys():  
    print (x, ":", numbers[x])
```

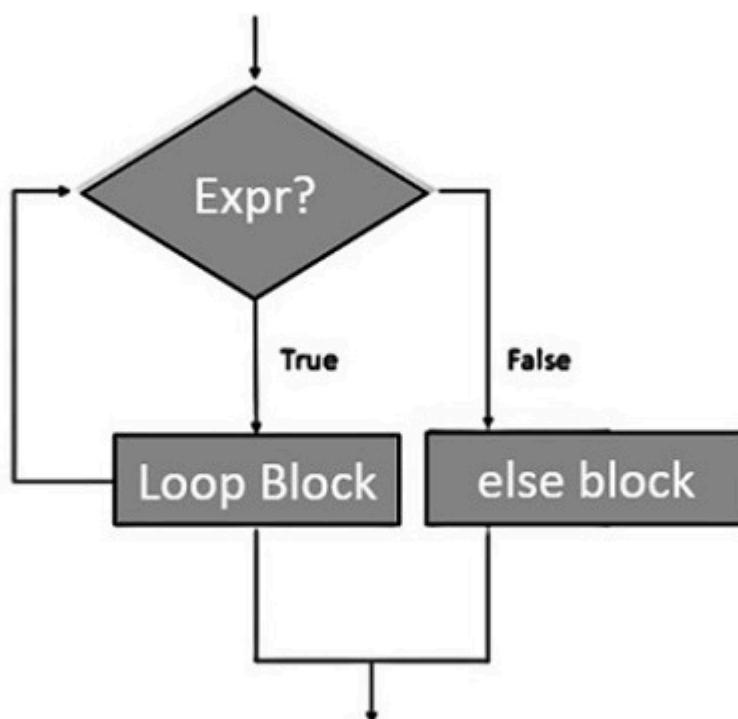
It will produce the same **output** –

```
10 : Ten
20 : Twenty
30 : Thirty
40 : Forty
```

## Python - The **for** else Loop

Python supports having an "else" statement associated with a "for" loop statement. If the "else" statement is used with a "for" loop, the "else" statement is executed when the sequence is exhausted before the control shifts to the main line of execution.

The following flow diagram illustrates how to use **else** statement with **for** loop –



### Example

The following example illustrates the combination of an else statement with a for statement. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```
</>

for count in range(6):
    print ("Iteration no. {}".format(count))
else:
```

Open Compiler

```
print ("for loop over. Now in else block")
print ("End of for loop")
```

On executing, this code will produce the following **output** –

```
Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
for loop over. Now in else block
End of for loop
```

## Nested Loops

Python programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

### Syntax

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
        statements(s)
```

The syntax for a nested **while** loop statement in Python programming language is as follows –

```
while expression:
    while expression:
        statement(s)
    statement(s)
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example a for loop can be inside a while loop or vice versa.

### Example

The following program uses a nested-for loop to display multiplication tables from 1-10.

</>

Open Compiler

```
#!/usr/bin/python3
for i in range(1,11):
    for j in range(1,11):
        k=i*j
        print ("{:3d}".format(k), end=' ')
print()
```

The `print()` function inner loop has `end=' '` which appends a space instead of default newline. Hence, the numbers will appear in one row.

The last `print()` will be executed at the end of inner **for** loop.

When the above code is executed, it produces the following **output** –

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

## Python - The while Loop

Normally, flow of execution of steps in a computer program goe from start to end. However, instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given boolean expression is true.

### Syntax

The syntax of a **while** loop in Python programming language is –

```
while expression:
    statement(s)
```

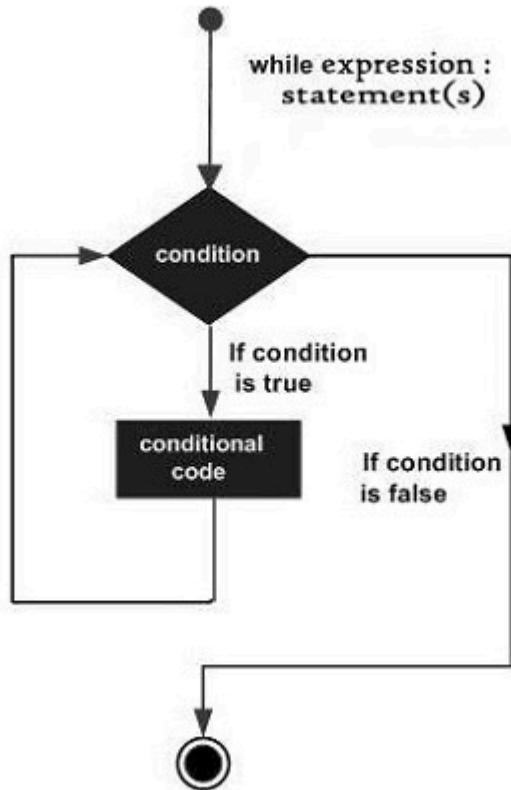
The **while** keyword is followed by a boolean expression, and then by colon symbol, to start an indented block of statements. Here, `statement(s)` may be a single statement or a

block of statements with uniform indent. The condition may be any expression, and true is any non-zero value. The loop iterates while the boolean expression is true.

As soon as the expression becomes false, the program control passes to the line immediately following the loop.

If it fails to turn false, the loop continues to run, and doesn't stop unless forcefully stopped. Such a loop is called infinite loop, which is undesired in a computer program.

The following flow diagram illustrates the **while** loop –



## Example 1

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

```
</>  
count=0  
while count<5:  
    count+=1  
    print ("Iteration no. {}".format(count))  
  
print ("End of while loop")
```

Open Compiler

We initialize count variable to 0, and the loop runs till "count<5". In each iteration, count is incremented and checked. If it's not 5 next repetition takes place. Inside the looping block, instantaneous value of count is printed. When the **while** condition becomes false, the loop terminates, and next statement is executed, here it is End of **while** loop message.

## Output

On executing, this code will produce the following output –

```
Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
End of while loop
```

## Example 2

Here is another example of using the **while** loop. For each iteration, the program asks for user input and keeps repeating till the user inputs a non-numeric string. The `isnumeric()` function that returns true if input is an integer, false otherwise.

```
var='0'
while var.isnumeric()==True:
    var=input('enter a number..')
    if var.isnumeric()==True:
        print ("Your input", var)
    print ("End of while loop")
```

## Output

On executing, this code will produce the following output –

```
enter a number..10
Your input 10
enter a number..100
Your input 100
enter a number..543
Your input 543
enter a number..qwer
End of while loop
```

## The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

### Example 3

Let's take an example to understand how the infinite loop works in Python –

```
#!/usr/bin/python3
var = 1
while var == 1 : # This constructs an infinite loop
    num = int(input("Enter a number :"))
    print ("You entered: ", num)
print ("Good bye!")
```

### Output

On executing, this code will produce the following output –

```
Enter a number :20
You entered: 20
Enter a number :29
You entered: 29
Enter a number :3
You entered: 3
Enter a number :11
You entered: 11
Enter a number :22
You entered: 22
Enter a number :Traceback (most recent call last):
  File "examples\test.py", line 5, in
    num = int(input("Enter a number :"))
KeyboardInterrupt
```

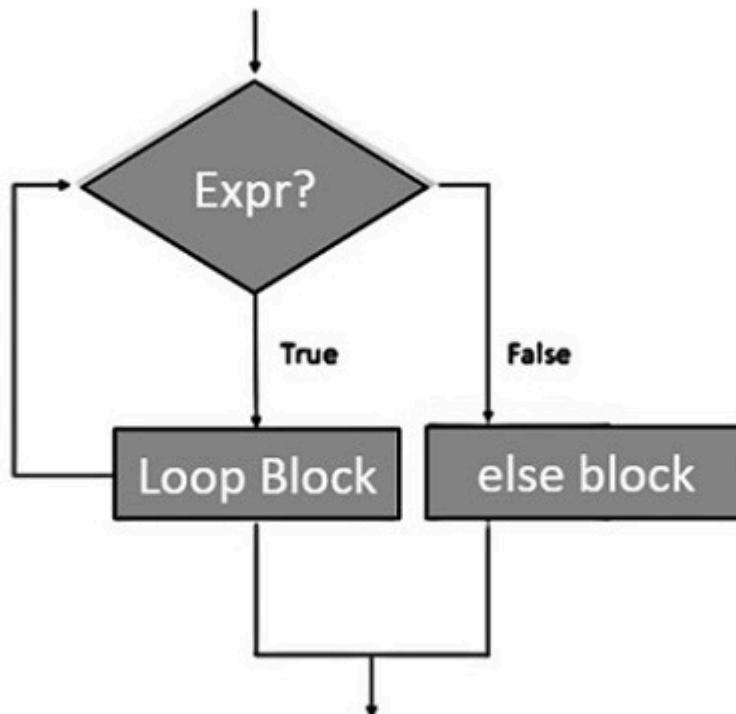
The above example goes in an infinite loop and you need to use CTRL+C to exit the program.

## The while-else Loop

Python supports having an **else** statement associated with a **while** loop statement.

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false before the control shifts to the main line of execution.

The following flow diagram shows how to use **else** with **while** statement –



## Example

The following example illustrates the combination of an else statement with a while statement. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```
</>
count=0
while count<5:
    count+=1
    print ("Iteration no. {}".format(count))
else:
    print ("While loop over. Now in else block")
    print ("End of while loop")
```

Open Compiler

## Output

On executing, this code will produce the following **output** –

```
Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
While loop over. Now in else block
End of while loop
```

# Python - The break Statement

## Loop Control Statements

The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements –

Sr.No.	Control Statement & Description
1	<b>break statement</b> Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	<b>continue statement</b> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	<b>pass statement</b> The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly.

## Python – The break Statement

The **break** statement is used for premature termination of the current loop. After abandoning the loop, execution at the next statement is resumed, just like the traditional

break statement in C.

The most common use of break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both while and for loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

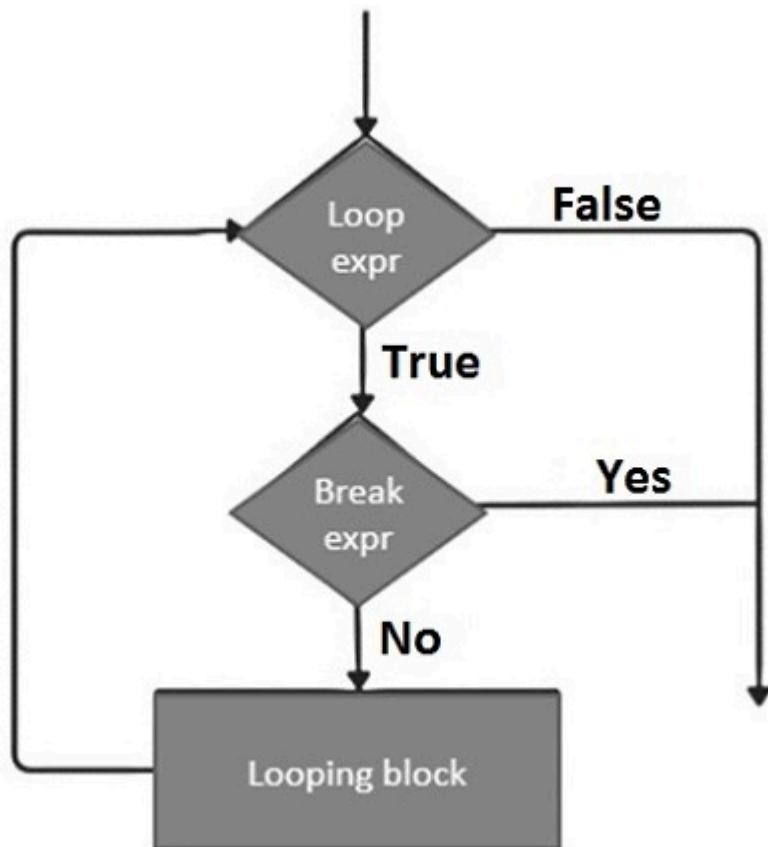
## Syntax

The syntax for a **break** statement in Python is as follows –

```
break
```

## Flow Diagram

Its flow diagram looks like this –



## Example 1

Now let's take an example to understand how the "break" statement works in Python –

```
</>
```

Open Compiler

```
#!/usr/bin/python3
print ('First example')
for letter in 'Python': # First Example
    if letter == 'h':
        break
    print ('Current Letter :', letter)
print ('Second example')
var = 10 # Second Example
while var > 0:
    print ('Current variable value :', var)
    var = var -1
    if var == 5:
        break
print ("Good bye!")
```

When the above code is executed, it produces the following **output** –

```
First example
Current Letter : P
Current Letter : y
Current Letter : t
Second example
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

## Example 2

The following program demonstrates the use of **break** in a **for** loop iterating over a list. User inputs a number, which is searched in the list. If it is found, then the loop terminates with the 'found' message.

```
#!/usr/bin/python3
no=int(input('any number: '))
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num==no:
        print ('number found in list')
```

```
break  
else:  
    print ('number not found in list')
```

The above program will produce the following **output** –

```
any number: 33  
number found in list  
any number: 5  
number not found in list
```

### Example 3: Checking for Prime Number

Note that when the `break` statement is encountered, Python abandons the remaining statements in the loop, including the `else` block.

The following example takes advantage of this behaviour to find whether a number is prime or not. By definition, a number is prime if it is not divisible by any other number except 1 and itself.

The following code runs a `for` loop over numbers from 2 to the desired number-1. If it is divisible by any value of looping variable, the number is not prime, hence the program breaks from the loop. If the number is not divisible by any number between 2 and x-1, the `else` block prints the message that the given number is prime.

```
</> Open Compiler  
  
num = 37  
print ("Number: ", num)  
for x in range(2,num):  
    if num%x==0:  
        print ("{} is not prime".format(num))  
        break  
    else:  
        print ("{} is prime".format(num))
```

### Output

Assign different values to `num` to check if it is a prime number or not.

```
Number: 37  
37 is prime
```

Number: 49  
49 is not prime

# Python - The Continue Statement

The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

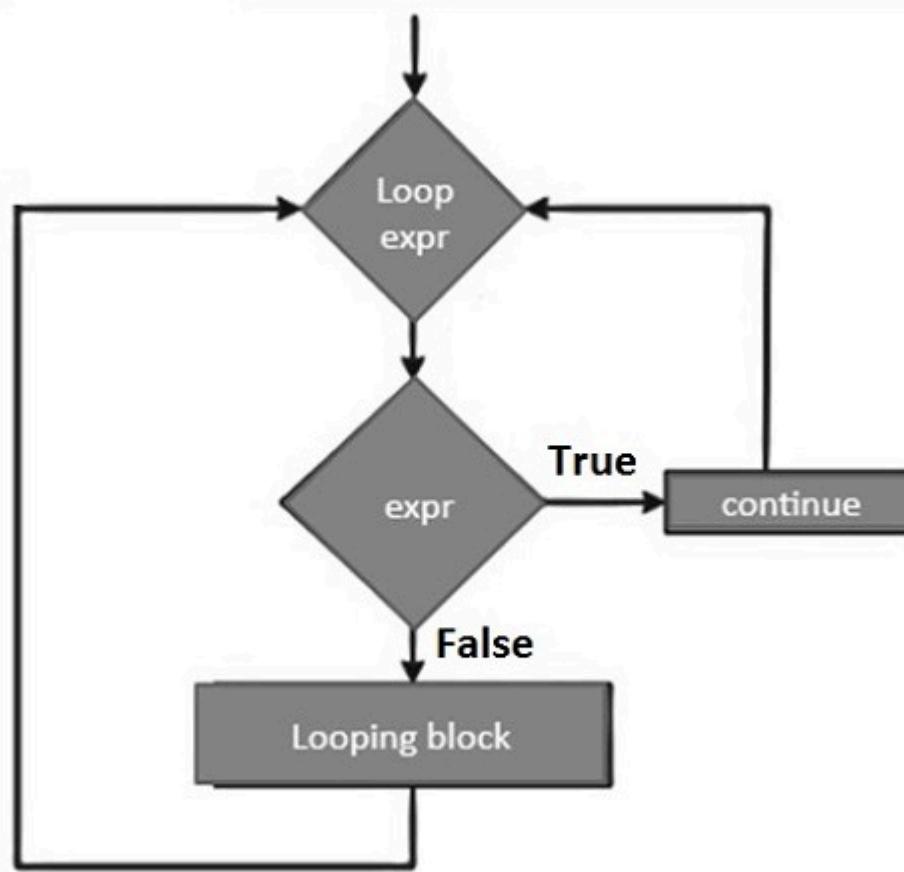
The continue statement can be used in both **while** and **for** loops.

## Syntax

```
continue
```

## Flow Diagram

The flow diagram of the **continue** statement looks like this –



The **continue** statement is just the opposite to that of **break**. It skips the remaining statements in the current loop and starts the next iteration.

## Example 1

Now let's take an example to understand how the **continue** statement works in Python –

</>

Open Compiler

```
for letter in 'Python': # First Example
    if letter == 'h':
        continue
    print ('Current Letter :', letter)
var = 10 # Second Example
while var > 0:
    var = var -1
    if var == 5:
        continue
    print ('Current variable value :', var)
print ("Good bye!")
```

When the above code is executed, it produces the following **output** –

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

## Example 2: Checking Prime Factors

Following code uses continue to find the prime factors of a given number. To find prime factors, we need to successively divide the given number starting with 2, increment the divisor and continue the same process till the input reduces to 1.

The algorithm for finding prime factors is as follows –

- Accept input from user (n)
- Set divisor (d) to 2
- Perform following till  $n > 1$
- Check if given number (n) is divisible by divisor (d).
- If  $n \% d == 0$ 
  - a. Print d as a factor
  - Set new value of n as  $n/d$
  - Repeat from 4
- If not
- Increment d by 1
- Repeat from 3

Given below is the Python code for the purpose –

</>

Open Compiler

```
num = 60
print ("Prime factors for: ", num)
d=2
while num>1:
    if num%d==0:
        print (d)
        num=num/d
        continue
    d=d+1
```

On executing, this code will produce the following **output** –

Prime factors for: 60

2  
2  
3  
5

Assign different value (say 75) to num in the above program and test the result for its prime factors.

Prime factors for: 75

3  
5  
5

## Python - The pass Statement

The **pass** statement is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a null operation; nothing happens when it executes. The **pass** statement is also useful in places where your code will eventually go, but has not been written yet, i.e., in stubs).

### Syntax

```
pass
```

### Example

The following code shows how you can use the **pass** statement in Python –

```
</>  
  
for letter in 'Python':  
    if letter == 'h':  
        pass  
        print ('This is pass block')  
    print ('Current Letter :', letter)  
print ("Good bye!")
```

Open Compiler

When the above code is executed, it produces the following **output** –

Current Letter : P  
Current Letter : y  
Current Letter : t  
This is pass block  
Current Letter : h

Current Letter : o

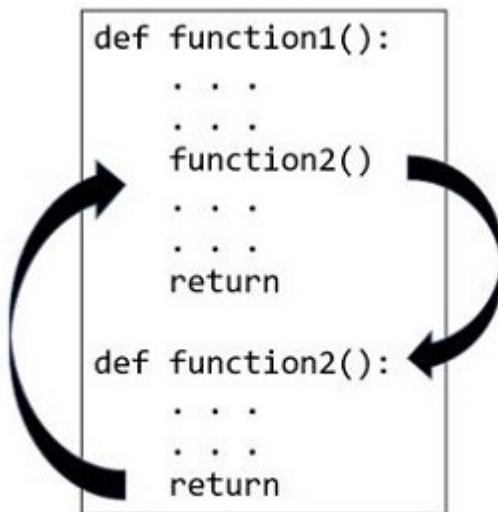
Current Letter : n

Good bye!

## Python - Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

A top-to-down approach towards building the processing logic involves defining blocks of independent reusable functions. A function may be invoked from any other function by passing required data (called **parameters** or **arguments**). The called function returns its result back to the calling environment.



### Types of Python Functions

Python provides the following types of functions –

- Built-in functions
- Functions defined in built-in modules
- User-defined functions

Python's standard library includes number of built-in functions. Some of Python's built-in functions are `print()`, `int()`, `len()`, `sum()`, etc. These functions are always available, as they are loaded into computer's memory as soon as you start Python interpreter.

The standard library also bundles a number of modules. Each module defines a group of functions. These functions are not readily available. You need to import them into the

memory from their respective modules.

In addition to the built-in functions and functions in the built-in modules, you can also create your own functions. These functions are called user-defined functions.

## Python Defining a Function

You can define custom functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( () ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement; the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement **return [expression]** exits a function, optionally passing back an expression to the caller. A **return** statement with no arguments is the same as `return None`.

## Syntax

```
def functionname( parameters ):  
    "function_docstring"  
    function_suite  
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Once the function is defined, you can execute it by calling it from another function or directly from the Python prompt.

## Example

The following example shows how to define a function `greetings()`. The bracket is empty so there aren't any parameters.

The first line is the docstring. Function block ends with return statement. when this function is called, **Hello world** message will be printed.

```
def greetings():
    "This is docstring of greetings function"
    print ("Hello World")
    return
greetings()
```

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call `printme()` function –

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

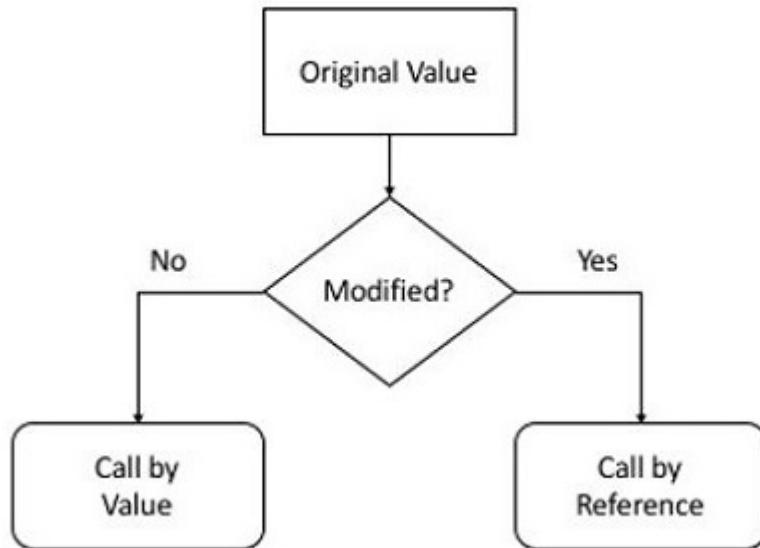
When the above code is executed, it produces the following **output** –

```
I'm first call to user defined function!
Again second call to the same function
```

## Pass by Reference vs Value

The **function calling mechanism** of Python differs from that of C and C++. There are two main function calling mechanisms: **Call by Value** and **Call by Reference**.

When a variable is passed to a function, what does the function do to it? If any changes to its variable does not get reflected in the actual argument, then it uses call by value mechanism. On the other hand, if the change is reflected, then it becomes call by reference mechanism.



**C/C++ functions are said to be called by value.** When a function in C/C++ is called, the value of actual arguments is copied to the variables representing the formal arguments. If the function modifies the value of formal argument, it doesn't reflect the variable that was passed to it.

**Python uses pass by reference mechanism.** As variable in Python is a label or reference to the object in the memory, the both the variables used as actual argument as well as formal arguments really refer to the same object in the memory. We can verify this fact by checking the `id()` of the passed variable before and after passing.

&lt;/&gt;

Open Compiler

```

def testfunction(arg):
    print ("ID inside the function:", id(arg))
var="Hello"
print ("ID before passing:", id(var))
testfunction(var)
  
```

If the above code is executed, the `id()` before passing and inside the function is same.

```

ID before passing: 1996838294128
ID inside the function: 1996838294128
  
```

The behaviour also depends on whether the passed object is mutable or immutable. Python numeric object is immutable. When a numeric object is passed, and then the function changes the value of the formal argument, it actually creates a new object in the memory, leaving the original variable unchanged.

&lt;/&gt;

[Open Compiler](#)

```
def testfunction(arg):
    print ("ID inside the function:", id(arg))
    arg=arg+1
    print ("new object after increment", arg, id(arg))

var=10
print ("ID before passing:", id(var))
testfunction(var)
print ("value after function call", var)
```

It will produce the following **output** –

```
ID before passing: 140719550297160
ID inside the function: 140719550297160
new object after increment 11 140719550297192
value after function call 10
```

Let us now pass a mutable object (such as a list or dictionary) to a function. It is also passed by reference, as the id() of list before and after passing is same. However, if we modify the list inside the function, its global representation also reflects the change.

Here we pass a list, append a new item, and see the contents of original list object, which we will find has changed.

&lt;/&gt;

[Open Compiler](#)

```
def testfunction(arg):
    print ("Inside function:",arg)
    print ("ID inside the function:", id(arg))
    arg.append(100)

var=[10, 20, 30, 40]
print ("ID before passing:", id(var))
testfunction(var)
print ("list after function call", var)
```

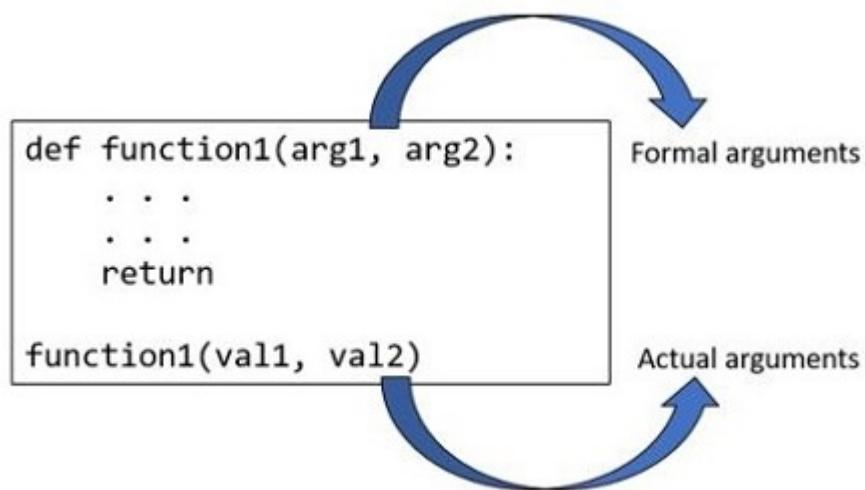
It will produce the following **output** –

```
ID before passing: 2716006372544
Inside function: [10, 20, 30, 40]
ID inside the function: 2716006372544
list after function call [10, 20, 30, 40, 100]
```

## Function Arguments

The process of a function often depends on certain data provided to it while calling it. While defining a function, you must give a list of variables in which the data passed to it is collected. The variables in the parentheses are called formal arguments.

When the function is called, value to each of the formal arguments must be provided. Those are called actual arguments.



## Example

Let's modify greetings function and have name an argument. A string passed to it whilcalling becomes name variable inside the function.

```
</> Open Compiler
```

```
def greetings(name):
    "This is docstring of greetings function"
    print ("Hello {}".format(name))
    return

greetings("Samay")
greetings("Pratima")
greetings("Steven")
```

It will produce the following **output** –

```
Hello Samay  
Hello Pratima  
Hello Steven
```

## Function with Return Value

The **return** keyword as the last statement in function definition indicates end of function block, and the program flow goes back to the calling function. Although reduced indent after the last statement in the block also implies return but using explicit return is a good practice.

Along with the flow control, the function can also return value of an expression to the calling function. The value of returned expression can be stored in a variable for further processing.

### Example

Let us define the add() function. It adds the two values passed to it and returns the addition. The returned value is stored in a variable called result.

```
</> Open Compiler  
  
def add(x,y):  
    z=x+y  
    return z  
  
a=10  
b=20  
result = add(a,b)  
print ("a = {} b = {} a+b = {}".format(a, b, result))
```

It will produce the following output –

```
a = 10 b = 20 a+b = 30
```

## Types of Function Arguments

Based on how the arguments are declared while defining a Python function, there are classified into the following categories –

- Positional or required arguments
- Keyword arguments
- Default arguments
- Positional-only arguments
- Keyword-only arguments
- Arbitrary or variable-length arguments

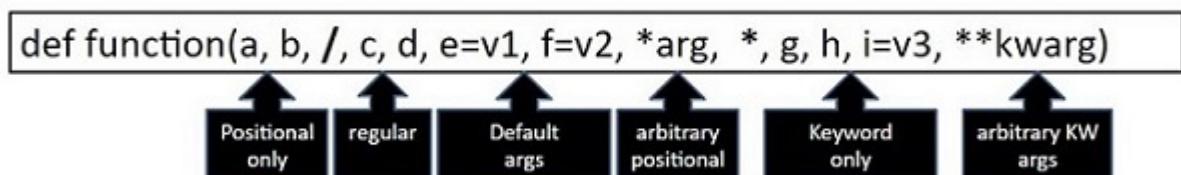
In the next few chapters, we will discuss these function arguments at length.

## Order of Arguments

A function can have arguments of any of the types defined above. However, the arguments must be declared in the following order –

- The argument list begins with the positional-only args, followed by the slash (/) symbol.
- It is followed by regular positional args that may or may not be called as keyword arguments.
- Then there may be one or more args with default values.
- Next, arbitrary positional arguments represented by a variable prefixed with single asterisk, that is treated as tuple. It is the next.
- If the function has any keyword-only arguments, put an asterisk before their names start. Some of the keyword-only arguments may have a default value.
- Last in the bracket is argument with two asterisks \*\* to accept arbitrary number of keyword arguments.

The following diagram shows the order of formal arguments –



## Python - Default Arguments

You can define a function with default value assigned to one or more formal arguments. Python uses the default value for such an argument if no value is passed to it. If any value is passed, the default is overridden.

## Example

```
</> Open Compiler

# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

It will produce the following **output** –

```
Name: miki
Age 50
Name: miki
Age 35
```

In the above example, the second call to the function doesn't pass value to age argument, hence its default value 35 is used.

Let us look at another example that assigns default value to a function argument. The function percent() is defined as below –

```
def percent(phy, maths, maxmarks=200):
    val = (phy+maths)*100/maxmarks
    return val
```

Assuming that marks given for each subject are out of 100, the argument maxmarks is set to 200. Hence, we can omit the value of third argument while calling percent() function.

```
phy = 60
maths = 70
result = percent(phy,maths)
```

However, if maximum marks for each subject is not 100, then we need to put the third argument while calling the percent() function.

```
phy = 40
maths = 46
result = percent(phy,maths, 100)
```

## Example

Here is the complete example –

```
</>
```

Open Compiler

```
def percent(phy, maths, maxmarks=200):
    val = (phy+maths)*100/maxmarks
    return val

phy = 60
maths = 70
result = percent(phy,maths)
print ("percentage:", result)

phy = 40
maths = 46
result = percent(phy,maths, 100)
print ("percentage:", result)
```

It will produce the following **output** –

```
percentage: 65.0
percentage: 86.0
```

## Python - Keyword Arguments

Keyword argument are also called named arguments. Variables in the function definition are used as keywords. When the function is called, you can explicitly mention the name and its value.

## Example

```
</>
```

Open Compiler

```
# Function definition is here
def printinfo( name, age ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return

# Now you can call printinfo function
# by positional arguments
printinfo ("Naveen", 29)

# by keyword arguments
printinfo(name="miki", age = 30)
```

By default, the function assigns the values to arguments in the order of appearance. In the second function call, we have assigned the value to a specific argument

It will produce the following **output** –

```
Name: Naveen
Age 29
Name: miki
Age 30
```

Let us try to understand more about keyword argument with the help of following function definition –

```
</> Open Compiler

def division(num, den):
    quotient = num/den
    print ("num:{} den:{} quotient:{}".format(num, den, quotient))

division(10,5)
division(5,10)
```

Since the values are assigned as per the position, the output is as follows –

```
num:10 den:5 quotient:2.0
num:5 den:10 quotient:0.5
```

Instead of passing the values with positional arguments, let us call the function with keyword arguments –

```
division(num=10, den=5)
division(den=5, num=10)
```

It will produce the following **output** –

```
num:10 den:5 quotient:2.0
num:10 den:5 quotient:2.0
```

When using keyword arguments, it is not necessary to follow the order of formal arguments in function definition.

Using keyword arguments is optional. You can use mixed calling. You can pass values to some arguments without keywords, and for others with keyword.

```
division(10, den=5)
```

However, the positional arguments must be before the keyword arguments while using mixed calling.

Try to call the division() function with the following statement.

```
division(num=5, 10)
```

As the Positional argument cannot appear after keyword arguments, Python raises the following error message –

```
division(num=5, 10)
^
```

```
SyntaxError: positional argument follows keyword argument
```

## Python - Keyword-Only Arguments

You can use the variables in formal argument list as keywords to pass value. Use of keyword arguments is optional. But, you can force the function be given arguments by keyword only. You should put an asterisk (\*) before the keyword-only arguments list.

Let us say we have a function with three arguments, out of which we want second and third arguments to be keyword-only. For that, put \* after the first argument.

The built-in print() function is an example of keyword-only arguments. You can give list of expressions to be printed in the parentheses. The printed values are separated by a white space by default. You can specify any other separation character instead with sep argument.

&lt;/&gt;

[Open Compiler](#)

```
print ("Hello", "World", sep="-")
```

It will print –

Hello-World

The **sep** argument is keyword-only. Try using it as non-keyword argument.

&lt;/&gt;

[Open Compiler](#)

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

You'll get different output – not as desired.

Hello World -

## Example

In the following user defined function intr() with two arguments, amt and rate. To make the **rate** argument keyword-only, put "\*" before it.

```
def intr(amt, *, rate):
    val = amt*rate/100
    return val
```

To call this function, the value for **rate** must be passed by keyword.

```
interest = intr(1000, rate=10)
```

However, if you try to use the default positional way of calling the function, you get an error.

```
interest = intr(1000, 10)
^^^^^^^^^^^^^^^^^
```

TypeError: intr() takes 1 positional argument but 2 were given

## Python - Positional Arguments

The list of variables declared in the parentheses at the time of defining a function are the **formal arguments**. A function may be defined with any number of formal arguments.

While calling a function –

- All the arguments are required
- The number of actual arguments must be equal to the number of formal arguments.
- Formal arguments are positional. They Pick up values in the order of definition.
- The type of arguments must match.
- Names of formal and actual arguments need not be same.

### Example

</>

Open Compiler

```
def add(x,y):
    z=x+y
    print ("x={} y={} x+y={}".format(x,y,z))

a=10
b=20
add(a,b)
```

It will produce the following **output** –

x=10 y=20 x+y=30

Here, the add() function has two formal arguments, both are numeric. When integers 10 and 20 passed to it. The variable a takes 10 and b takes 20, in the order of declaration. The add() function displays the addition.

Python also raises error when the number of arguments don't match. Give only one argument and check the result.

```
add(b)  
TypeError: add() missing 1 required positional argument: 'y'
```

Pass more than number of formal arguments and check the result –

```
add(10, 20, 30)  
TypeError: add() takes 2 positional arguments but 3 were given
```

Data type of corresponding actual and formal arguments must match. Change a to a string value and see the result.

```
a="Hello"  
b=20  
add(a,b)
```

It will produce the following **output** –

```
z=x+y  
~^~  
TypeError: can only concatenate str (not "int") to str
```

## Python - Positional-Only Arguments

It is possible to define a function in which one or more arguments can not accept their value with keywords. Such arguments may be called positional-only arguments.

Python's built-in `input()` function is an example of positional-only arguments. The syntax of `input` function is –

```
input(prompt = "")
```

Prompt is an explanatory string for the benefit of the user. For example –

```
name = input("enter your name ")
```

However, you cannot use the `prompt` keyword inside the parentheses.

```
name = input (prompt="Enter your name ")
^~~~~~
```

TypeError: input() takes no keyword arguments

To make an argument positional-only, use the "/" symbol. All the arguments before this symbol will be treated as position-only.

## Example

We make both the arguments of intr() function as positional-only by putting "/" at the end.

```
def intr(amt, rate, /):
    val = amt*rate/100
    return val
```

If we try to use the arguments as keywords, Python raises following error message –

```
interest = intr(amt=1000, rate=10)
^~~~~~
```

TypeError: intr() got some positional-only arguments passed as keyword arguments: 'am

A function may be defined in such a way that it has some keyword-only and some positional-only arguments.

```
def myfunction(x, /, y, *, z):
    print (x, y, z)
```

In this function, x is a required positional-only argument, y is a regular positional argument (you can use it as keyword if you want), and z is a keyword-only argument.

The following function calls are valid –

```
myfunction(10, y=20, z=30)
myfunction(10, 20, z=30)
```

However, these calls raise errors –

```
myfunction(x=10, y=20, z=30)
TypeError: myfunction() got some positional-only arguments passed as keyword arguments:
```

```
myfunction(10, 20, 30)
```

```
TypeError: myfunction() takes 2 positional arguments but 3 were given
```

## Python - Arbitrary Arguments

You may want to define a function that is able to accept arbitrary or variable number of arguments. Moreover, the arbitrary number of arguments might be positional or keyword arguments.

- An argument prefixed with a single asterisk \* for arbitrary positional arguments.
- An argument prefixed with two asterisks \*\* for arbitrary keyword arguments.

### Example

Given below is an example of arbitrary or variable length positional arguments –

```
</> Open Compiler  
  
# sum of numbers  
def add(*args):  
    s=0  
    for x in args:  
        s=s+x  
    return s  
result = add(10,20,30,40)  
print (result)  
  
result = add(1,2,3)  
print (result)
```

The **args** variable prefixed with "\*" stores all the values passed to it. Here, args becomes a tuple. We can run a loop over its items to add the numbers.

It will produce the following **output** –

```
100  
6
```

It is also possible to have a function with some required arguments before the sequence of variable number of values.

## Example

The following example has **avg()** function. Assume that a student can take any number of tests. First test is mandatory. He can take as many tests as he likes to better his score. The function calculates the average of marks in first test and his maximum score in the rest of tests.

The function has two arguments, first is the required argument and second to hold any number of values.

&lt;/&gt;

Open Compiler

```
#avg of first test and best of following tests
def avg(first, *rest):
    second=max(rest)
    return (first+second)/2

result=avg(40,30,50,25)
print (result)
```

Following call to avg() function passes first value to the required argument first, and the remaining values to a tuple named rest. We then find the maximum and use it to calculate the average.

It will produce the following **output** –

45.0

If a variable in the argument list has two asterisks prefixed to it, the function can accept arbitrary number of keyword arguments. The variable becomes a dictionary of keyword:value pairs.

## Example

The following code is an example of a function with arbitrary keyword arguments. The addr() function has an argument **\*\*kwargs** which is able to accept any number of address elements like name, city, phno, pin, etc. Inside the function kwargs dictionary of kw:value pairs is traversed using items() method.

&lt;/&gt;

Open Compiler

```
def addr(**kwargs):
    for k,v in kwargs.items():
```

```

print ("{}:{}".format(k,v))

print ("pass two keyword args")
addr(Name="John", City="Mumbai")
print ("pass four keyword args")

# pass four keyword args
addr(Name="Raam", City="Mumbai", ph_no="9123134567", PIN="400001")

```

It will produce the following **output** –

```

pass two keyword args
Name:John
City:Mumbai
pass four keyword args
Name:Raam
City:Mumbai
ph_no:9123134567
PIN:400001

```

If the function uses mixed types of arguments, the arbitrary keyword arguments should be after positional, keyword and arbitrary positional arguments in the argument list.

## Example

Imagine a case where science and maths are mandatory subjects, in addition to which student may choose any number of elective subjects.

The following code defines a **percent()** function where marks in science and marks are stored in required arguments, and the marks in variable number of elective subjects in **\*\*optional** argument.

&lt;/&gt;

Open Compiler

```

def percent(math, sci, **optional):
    print ("maths:", math)
    print ("sci:", sci)
    s=math+sci
    for k,v in optional.items():
        print ("{}:{}".format(k,v))
        s=s+v
    return s/(len(optional)+2)

```

```
result=percent(math=80, sci=75, Eng=70, Hist=65, Geo=72)
print ("percentage:", result)
```

It will produce the following **output** –

```
maths: 80
sci: 75
Eng:70
Hist:65
Geo:72
percentage: 72.4
```

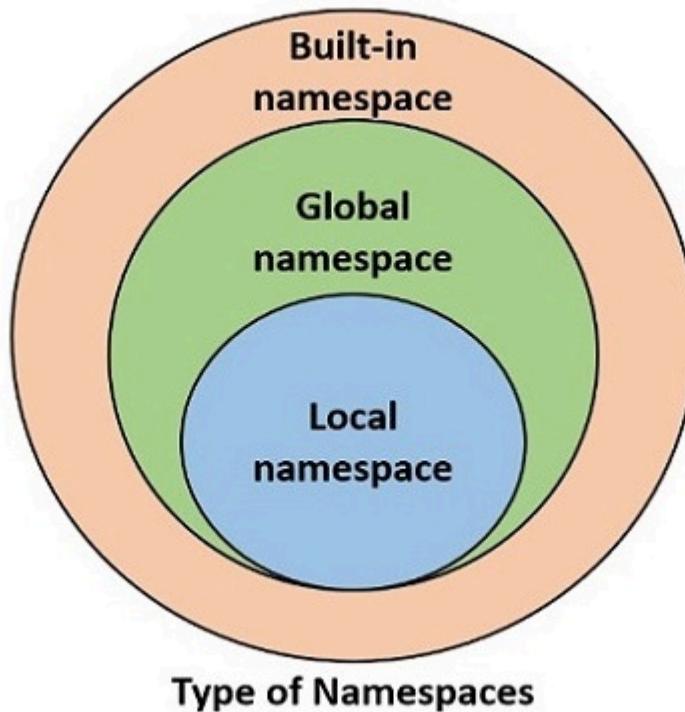
## Python - Variable Scope

A variable in Python is a symbols name to the object in computer's memory. Python works on the concept of namespaces to define the context for various identifiers such as functions, variables etc. A namespace is a collection of symbolic names defined in the current context.

Python provides the following types of namespaces –

- **Built-in namespace** contains built-in functions and built-in exceptions. They are loaded in the memory as soon as Python interpreter is loaded and remain till the interpreter is running.
- **Global namespace** contains any names defined in the main program. These names remain in memory till the program is running.
- **Local namespace** contains names defined inside a function. They are available till the function is running.

These namespaces are nested one inside the other. Following diagram shows relationship between namespaces.



The life of a certain variable is restricted to the namespace in which it is defined. As a result, it is not possible to access a variable present in the inner namespace from any outer namespace.

## globals() Function

Python's standard library includes a built-in function `globals()`. It returns a dictionary of symbols currently available in global namespace.

Run the `globals()` function directly from the Python prompt.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '__
  ↪
```

It can be seen that the `builtins` module which contains definitions of all built-in functions and built-in exceptions is loaded.

Save the following code that contains few variables and a function with few more variables inside it.

```
</>
```

Open Compiler

```
name = 'TutorialsPoint'
marks = 50
result = True
```

```
def myfunction():
    a = 10
    b = 20
    return a+b

print (globals())
```

Calling `globals()` from inside this script returns following dictionary object –

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen...
```

The global namespace now contains variables in the program and their values and the function object in it (and not the variables in the function).

## locals() Function

Python's standard library includes a built-in function `locals()`. It returns a dictionary of symbols currently available in namespace of the function.

Modify the above script to print dictionary of global and local namespaces from within the function.

```
</>
```

Open Compiler

```
name = 'TutorialsPoint'
marks = 50
result = True
def myfunction():
    a = 10
    b = 20
    c = a+b
    print ("globals():", globals())
    print ("locals():", locals())
    return c
myfunction()
```

The **output** shows that `locals()` returns a dictionary of variables and their values currently available in the function.

```
globals(): {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':  
locals(): {'a': 10, 'b': 20, 'c': 30}
```

Since both `globals()` and `locals()` functions return dictionary, you can access value of a variable from respective namespace with dictionary `get()` method or index operator.

```
print (globals()['name']) #displays TutorialsPoint  
print (locals().get('a')) #displays 10
```

## Namespace Conflict

If a variable of same name is present in global as well as local scope, Python interpreter gives priority to the one in local namespace.

</>

Open Compiler

```
marks = 50 # this is a global variable  
def myfunction():  
    marks = 70 # this is a local variable  
    print (marks)  
  
myfunction()  
print (marks) # prints global value
```

It will produce the following **output** –

70

50

If you try to manipulate value of a global variable from inside a function, Python raises **UnboundLocalError**.

</>

Open Compiler

```
marks = 50 # this is a global variable  
def myfunction():  
    marks = marks + 20  
    print (marks)
```

```
myfunction()  
print (marks) # prints global value
```

It will produce the following **output** –

```
marks = marks + 20
```

~~~~~

UnboundLocalError: cannot access local variable 'marks' where it is not associated with a

To modify a global variable, you can either update it with a dictionary syntax, or use the **global** keyword to refer it before modifying.

</>

Open Compiler

```
var1 = 50 # this is a global variable
```

```
var2 = 60 # this is a global variable
```

```
def myfunction():
```

```
    "Change values of global variables"
```

```
    globals()['var1'] = globals()['var1']+10
```

```
    global var2
```

```
    var2 = var2 + 20
```

```
myfunction()
```

```
print ("var1:",var1, "var2:",var2) #shows global variables with changed values
```

It will produce the following **output** –

```
var1: 60 var2: 80
```

Lastly, if you try to access a local variable in global scope, Python raises NameError as the variable in local scope can't be accessed outside it.

</>

Open Compiler

```
var1 = 50 # this is a global variable
```

```
var2 = 60 # this is a global variable
```

```
def myfunction(x, y):
```

```
    total = x+y
```

```
    print ("Total is a local variable: ", total)
```

```
myfunction(var1, var2)
print (total) # This gives NameError
```

It will produce the following output –

```
Total is a local variable: 110
Traceback (most recent call last):
  File "C:\Users\user\examples\main.py", line 9, in <module>
    print (total) # This gives NameError
               ^
NameError: name 'total' is not defined
```

## Python - Function Annotations

The function annotation feature of Python enables you to add additional explanatory metadata about the arguments declared in a function definition, and also the return data type.

Although you can use the docstring feature of Python for documentation of a function, it may be obsolete if certain changes in the function's prototype are made. Hence, the annotation feature was introduced in Python as a result of PEP 3107.

The annotations are not considered by Python interpreter while executing the function. They are mainly for the Python IDEs for providing a detailed documentation to the programmer.

Annotations are any valid Python expressions added to the arguments or return data type. Simplest example of annotation is to prescribe the data type of the arguments. Annotation is mentioned as an expression after putting a colon in front of the argument.

```
def myfunction(a: int, b: int):
    c = a+b
    return c
```

Remember that Python is a dynamically typed language, and doesn't enforce any type checking at runtime. Hence annotating the arguments with data types doesn't have any effect while calling the function. Even if non-integer arguments are given, Python doesn't detect any error.

</>

Open Compiler

```
def myfunction(a: int, b: int):
    c = a+b
    return c

print (myfunction(10,20))
print (myfunction("Hello ", "Python"))
```

It will produce the following **output** –

```
30
Hello Python
```

Annotations are ignored at runtime, but are helpful for the IDEs and static type checker libraries such as mypy.

You can give annotation for the return data type as well. After the parentheses and before the colon symbol, put an arrow (->) followed by the annotation. For example –

```
def myfunction(a: int, b: int) -> int:
    c = a+b
    return c
```

As using the data type as annotation is ignored at runtime, you can put any expression which acts as the metadata for the arguments. Hence, function may have any arbitrary expression as annotation as in following example –

```
def total(x : 'marks in Physics', y: 'marks in chemistry'):
    return x+y
```

If you want to specify a default argument along with the annotation, you need to put it after the annotation expression. Default arguments must come after the required arguments in the argument list.

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
    c = a+b
    return c
print (myfunction(10))
```

The function in Python is also an object, and one of its attributes is `__annotations__`. You can check with `dir()` function.

```
print (dir(myfunction))
```

This will print the list of myfunction object containing `__annotations__` as one of the attributes.

```
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__', '__code__', '__d']
```

The `__annotations__` attribute itself is a dictionary in which arguments are keys and annotations their values.

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
    c = a+b
    return c
print (myfunction.__annotations__)
```

It will produce the following **output** –

```
{'a': 'physics', 'b': 'Maths', 'return': <class 'int'>}
```

You may have arbitrary positional and/or arbitrary keyword arguments for a function. Annotations can be given for them also.

</>

[Open Compiler](#)

```
def myfunction(*args: "arbitrary args", **kwargs: "arbitrary keyword args") -> int:
    pass
print (myfunction.__annotations__)
```

It will produce the following **output** –

```
{'args': 'arbitrary args', 'kwargs': 'arbitrary keyword args', 'return': <class 'int'>}
```

In case you need to provide more than one annotation expressions to a function argument, give it in the form of a dictionary object in front of the argument itself.

</>

[Open Compiler](#)

```
def division(num: dict(type=float, msg='numerator'), den: dict(type=float, msg='de
    return num/den
print (division.__annotations__)
```

It will produce the following **output** –

```
{'num': {'type': <class 'float'>, 'msg': 'numerator'}, 'den': {'type': <class 'float'>, 'msg':
```

## Python - Modules

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

The concept of module in Python further enhances the modularity. You can define more than one related functions together and load required functions. A module is a file containing definition of functions, classes, variables, constants or any other Python object. Contents of this file can be made available to any other program. Python has the import keyword for this purpose.

### Example

```
</>
```

Open Compiler

```
import math
print ("Square root of 100:", math.sqrt(100))
```

It will produce the following **output** –

```
Square root of 100: 10.0
```

## Built in Modules

Python's standard library comes bundled with a large number of modules. They are called built-in modules. Most of these built-in modules are written in C (as the reference implementation of Python is in C), and pre-compiled into the library. These modules pack useful functionality like system-specific OS management, disk IO, networking, etc.

Here is a select list of built-in modules –

| Sr.No. | Name & Brief Description |
|--------|--------------------------|
|--------|--------------------------|

|    |                                                                                                                                                                                            |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <b>os</b>                                                                                                                                                                                  |
| 1  | This module provides a unified interface to a number of operating system functions.                                                                                                        |
| 2  | <b>string</b><br>This module contains a number of functions for string processing                                                                                                          |
| 3  | <b>re</b><br>This module provides a set of powerful regular expression facilities. Regular expression (RegEx), allows powerful string search and matching for a pattern in a string        |
| 4  | <b>math</b><br>This module implements a number of mathematical operations for floating point numbers. These functions are generally thin wrappers around the platform C library functions. |
| 5  | <b>cmath</b><br>This module contains a number of mathematical operations for complex numbers.                                                                                              |
| 6  | <b>datetime</b><br>This module provides functions to deal with dates and the time within a day. It wraps the C runtime library.                                                            |
| 7  | <b>gc</b><br>This module provides an interface to the built-in garbage collector.                                                                                                          |
| 8  | <b>asyncio</b><br>This module defines functionality required for asynchronous processing                                                                                                   |
| 9  | <b>Collections</b><br>This module provides advanced Container datatypes.                                                                                                                   |
| 10 | <b>functools</b><br>This module has Higher-order functions and operations on callable objects. Useful in functional programming                                                            |
| 11 | <b>operator</b><br>Functions corresponding to the standard operators.                                                                                                                      |
| 12 | <b>pickle</b><br>Convert Python objects to streams of bytes and back.                                                                                                                      |
| 13 | <b>socket</b><br>Low-level networking interface.                                                                                                                                           |
| 14 | <b>sqlite3</b>                                                                                                                                                                             |

|    |                                                                |
|----|----------------------------------------------------------------|
|    | A DB-API 2.0 implementation using SQLite 3.x.                  |
| 15 | <b>statistics</b><br>Mathematical statistics functions         |
| 16 | <b>typing</b><br>Support for type hints                        |
| 17 | <b>venv</b><br>Creation of virtual environments.               |
| 18 | <b>json</b><br>Encode and decode the JSON format.              |
| 19 | <b>wsgiref</b><br>WSGI Utilities and Reference Implementation. |
| 20 | <b>unittest</b><br>Unit testing framework for Python.          |
| 21 | <b>random</b><br>Generate pseudo-random numbers                |

## User Defined Modules

Any text file with .py extension and containing Python code is basically a module. It can contain definitions of one or more functions, variables, constants as well as classes. Any Python object from a module can be made available to interpreter session or another Python script by import statement. A module can also include runnable code.

## Create a Module

Creating a module is nothing but saving a Python code with the help of any editor. Let us save the following code as **mymodule.py**

```
def SayHello(name):
    print ("Hi {}! How are you?".format(name))
    return
```

You can now import mymodule in the current Python terminal.

```
>>> import mymodule
>>> mymodule.SayHello("Harish")
Hi Harish! How are you?
```

You can also import one module in another Python script. Save the following code as `example.py`

```
import mymodule
mymodule.SayHello("Harish")
```

Run this script from command terminal

```
C:\Users\user\examples> python example.py
Hi Harish! How are you?
```

## The import Statement

In Python, the **import** keyword has been provided to load a Python object from one module. The object may be a function, class, a variable etc. If a module contains multiple definitions, all of them will be loaded in the namespace.

Let us save the following code having three functions as **mymodule.py**.

```
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

The **import mymodule** statement loads all the functions in this module in the current namespace. Each function in the imported module is an attribute of this module object.

```
>>> dir(mymodule)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', 'average', 'power', 'sum']
```

To call any function, use the module object's reference. For example, `mymodule.sum()`.

```
import mymodule
print ("sum:",mymodule.sum(10,20))
print ("average:",mymodule.average(10,20))
print ("power:",mymodule.power(10, 2))
```

It will produce the following **output** –

```
sum:30  
average:15.0  
power:100
```

## The from ... import Statement

The import statement will load all the resources of the module in the current namespace. It is possible to import specific objects from a module by using this syntax. For example –

Out of three functions in **mymodule**, only two are imported in following executable script **example.py**

```
from mymodule import sum, average  
print ("sum:",sum(10,20))  
print ("average:",average(10,20))
```

It will produce the following output –

```
sum: 30  
average: 15.0
```

Note that function need not be called by prefixing name of its module to it.

## The from...import \* Statement

It is also possible to import all the names from a module into the current namespace by using the following import statement –

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

## The import ... as Statement

You can assign an alias name to the imported module.

```
from modulename as alias
```

The **alias** should be prefixed to the function while calling.

Take a look at the following **example** –

```
import mymodule as x
print ("sum:",x.sum(10,20))
print ("average:", x.average(10,20))
print ("power:", x.power(10, 2))
```

## Module Attributes

In Python, a module is an object of module class, and hence it is characterized by attributes.

Following are the module attributes –

- `__file__` returns the physical name of the module.
- `__package__` returns the package to which the module belongs.
- `__doc__` returns the docstring at the top of the module if any
- `__dict__` returns the entire scope of the module
- `__name__` returns the name of the module

## Example

Assuming that the following code is saved as **mymodule.py**

```
"The docstring of mymodule"
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

Let us check the attributes of mymodule by importing it in the following script –

```
import mymodule

print ("__file__ attribute:", mymodule.__file__)
```

```
print ("__doc__ attribute:", mymodule.__doc__)
print ("__name__ attribute:", mymodule.__name__)
```

It will produce the following **output** –

```
__file__ attribute: C:\Users\mlath\examples\mymodule.py
__doc__ attribute: The docstring of mymodule
__name__ attribute: mymodule
```

## The \_\_name\_\_ Attribute

The \_\_name\_\_ attribute of a Python module has great significance. Let us explore it in more detail.

In an interactive shell, \_\_name\_\_ attribute returns '\_\_main\_\_'

```
>>> __name__
'__main__'
```

If you import any module in the interpreter session, it returns the name of the module as the \_\_name\_\_ attribute of that module.

```
>>> import math
>>> math.__name__
'math'
```

From inside a Python script, the \_\_name\_\_ attribute returns '\_\_main\_\_'

```
#example.py
print ("__name__ attribute within a script:", __name__)
```

Run this in the command terminal –

```
__name__ attribute within a script: __main__
```

This attribute allows a Python script to be used as executable or as a module. Unlike in C++, Java, C# etc., in Python, there is no concept of the main() function. The Python program script with .py extension can contain function definitions as well as executable statements.

Save **mymodule.py** and with the following code –

```
</>  
"The docstring of mymodule"  
def sum(x,y):  
    return x+y  
  
print ("sum:",sum(10,20))
```

You can see that `sum()` function is called within the same script in which it is defined.

```
C:\Users\user\examples> python mymodule.py  
sum: 30
```

Now let us import this function in another script **example.py**.

```
import mymodule  
print ("sum:",mymodule.sum(10,20))
```

It will produce the following **output** –

```
C:\Users\user\examples> python example.py  
sum: 30  
sum: 30
```

The output "sum:30" appears twice. Once when `mymodule` module is imported. The executable statements in imported module are also run. Second output is from the calling script, i.e., **example.py** program.

What we want to happen is that when a module is imported, only the function should be imported, its executable statements should not run. This can be done by checking the value of `__name__`. If it is `__main__`, means it is being run and not imported. Include the executable statements like function calls conditionally.

Add **if** statement in **mymodule.py** as shown –

```
</>  
"The docstring of mymodule"  
def sum(x,y):  
    return x+y
```

```
if __name__ == "__main__":
    print ("sum:",sum(10,20))
```

Now if you run **example.py** program, you will find that the sum:30 output appears only once.

```
C:\Users\user\examples> python example.py
sum: 30
```

## The reload() Function

Sometimes you may need to reload a module, especially when working with the interactive interpreter session of Python.

Assume that we have a test module (test.py) with the following function –

```
def SayHello(name):
    print ("Hi {}! How are you?".format(name))
    return
```

We can import the module and call its function from Python prompt as –

```
>>> import test
>>> test.SayHello("Deepak")
Hi Deepak! How are you?
```

However, suppose you need to modify the SayHello() function, such as –

```
def SayHello(name, course):
    print ("Hi {}! How are you?".format(name))
    print ("Welcome to {} Tutorial by TutorialsPoint".format(course))
    return
```

Even if you edit the test.py file and save it, the function loaded in the memory won't update. You need to reload it, using reload() function in imp module.

```
>>> import imp
>>> imp.reload(test)
>>> test.SayHello("Deepak", "Python")
```

Hi Deepak! How are you?

Welcome to Python Tutorial by TutorialsPoint

# Python - Built-in Functions

As of Python 3.11.2 version, there are 71 built-in functions in Pyhthon. The list of built-in functions is given below –

| Sr.No. | Function & Description                                                                                           |
|--------|------------------------------------------------------------------------------------------------------------------|
| 1      | <b>abs()</b><br>Returns absolute value of a number                                                               |
| 2      | <b>aiter()</b><br>Returns an asynchronous iterator for an asynchronous iterable                                  |
| 3      | <b>all()</b><br>Returns true when all elements in iterable is true                                               |
| 4      | <b>anext()</b><br>Returns the next item from the given asynchronous iterator                                     |
| 5      | <b>any()</b><br>Checks if any Element of an Iterable is True                                                     |
| 6      | <b>ascii()</b><br>Returns String Containing Printable Representation                                             |
| 7      | <b>bin()</b><br>Converts integer to binary string                                                                |
| 8      | <b>bool()</b><br>Converts a Value to Boolean                                                                     |
| 9      | <b>breakpoint()</b><br>This function drops you into the debugger at the call site and calls sys.breakpointhook() |
| 10     | <b>bytearray()</b><br>returns array of given byte size                                                           |
| 11     | <b>bytes()</b><br>returns immutable bytes object                                                                 |
| 12     | <b>callable()</b><br>Checks if the Object is Callable                                                            |

|    |                                                                     |
|----|---------------------------------------------------------------------|
| 13 | <b>chr()</b><br>Returns a Character (a string) from an Integer      |
| 14 | <b>classmethod()</b><br>Returns class method for given function     |
| 15 | <b>compile()</b><br>Returns a code object                           |
| 16 | <b>complex()</b><br>Creates a Complex Number                        |
| 17 | <b>delattr()</b><br>Deletes Attribute From the Object               |
| 18 | <b>dict()</b><br>Creates a Dictionary                               |
| 19 | <b>dir()</b><br>Tries to Return Attributes of Object                |
| 20 | <b>divmod()</b><br>Returns a Tuple of Quotient and Remainder        |
| 21 | <b>enumerate()</b><br>Returns an Enumerate Object                   |
| 22 | <b>eval()</b><br>Runs Code Within Program                           |
| 23 | <b>exec()</b><br>Executes Dynamically Created Program               |
| 24 | <b>filter()</b><br>Constructs iterator from elements which are true |
| 25 | <b>float()</b><br>Returns floating point number from number, string |
| 26 | <b>format()</b><br>Returns formatted representation of a value      |
| 27 | <b>frozenset()</b><br>Returns immutable frozenset object            |
| 28 | <b>getattr()</b><br>Returns value of named attribute of an object   |

|    |                                                                       |
|----|-----------------------------------------------------------------------|
| 29 | <b>globals()</b><br>Returns dictionary of current global symbol table |
| 30 | <b>hasattr()</b><br>Returns whether object has named attribute        |
| 31 | <b>hash()</b><br>Returns hash value of an object                      |
| 32 | <b>help()</b><br>Invokes the built-in Help System                     |
| 33 | <b>hex()</b><br>Converts to Integer to Hexadecimal                    |
| 34 | <b>id()</b><br>Returns Identify of an Object                          |
| 35 | <b>input()</b><br>Reads and returns a line of string                  |
| 36 | <b>int()</b><br>Returns integer from a number or string               |
| 37 | <b>isinstance()</b><br>Checks if a Object is an Instance of Class     |
| 38 | <b>issubclass()</b><br>Checks if a Class is Subclass of another Class |
| 39 | <b>iter()</b><br>Returns an iterator                                  |
| 40 | <b>len()</b><br>Returns Length of an Object                           |
| 41 | <b>list()</b><br>Creates a list in Python                             |
| 42 | <b>locals()</b><br>Returns dictionary of a current local symbol table |
| 43 | <b>map()</b><br>Applies Function and Returns a List                   |
| 44 | <b>max()</b><br>Returns the largest item                              |

|    |                                                                   |
|----|-------------------------------------------------------------------|
| 45 | <b>memoryview()</b><br>Returns memory view of an argument         |
| 46 | <b>min()</b><br>Returns the smallest value                        |
| 47 | <b>next()</b><br>Retrieves next item from the iterator            |
| 48 | <b>object()</b><br>Creates a featureless object                   |
| 49 | <b>oct()</b><br>Returns the octal representation of an integer    |
| 50 | <b>open()</b><br>Returns a file object                            |
| 51 | <b>ord()</b><br>Returns an integer of the Unicode character       |
| 52 | <b>pow()</b><br>Returns the power of a number                     |
| 53 | <b>print()</b><br>Prints the Given Object                         |
| 54 | <b>property()</b><br>Returns the property attribute               |
| 55 | <b>range()</b><br>Returns a sequence of integers                  |
| 56 | <b>repr()</b><br>Returns a printable representation of the object |
| 57 | <b>reversed()</b><br>Returns the reversed iterator of a sequence  |
| 58 | <b>round()</b><br>Rounds a number to specified decimals           |
| 59 | <b>set()</b><br>Constructs and returns a set                      |
| 60 | <b>setattr()</b><br>Sets the value of an attribute of an object   |

|    |                                                                                |
|----|--------------------------------------------------------------------------------|
| 61 | <b><code>slice()</code></b><br>Returns a slice object                          |
| 62 | <b><code>sorted()</code></b><br>Returns a sorted list from the given iterable  |
| 63 | <b><code>staticmethod()</code></b><br>Transforms a method into a static method |
| 64 | <b><code>str()</code></b><br>Returns the string version of the object          |
| 65 | <b><code>sum()</code></b><br>Adds items of an Iterable                         |
| 66 | <b><code>super()</code></b><br>Returns a proxy object of the base class        |
| 67 | <b><code>tuple()</code></b><br>Returns a tuple                                 |
| 68 | <b><code>type()</code></b><br>Returns the type of the object                   |
| 69 | <b><code>vars()</code></b><br>Returns the <code>__dict__</code> attribute      |
| 70 | <b><code>zip()</code></b><br>Returns an iterator of tuples                     |
| 71 | <b><code>__import__()</code></b><br>Function called by the import statement    |

## Built-in Mathematical Functions

Following mathematical functions are built into the Python interpreter, hence you don't need to import them from any module.

| Sr.No. | Function & Description                                                                                                                                                           |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b><code>abs()</code> function</b><br>The <code>abs()</code> function returns the absolute value of <code>x</code> , i.e. the positive distance between <code>x</code> and zero. |

|   |                                                                                                                                                                                                  |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|   | <b>max() function</b>                                                                                                                                                                            |
| 2 | The max() function returns the largest of its arguments or largest number from the iterable (list or tuple).                                                                                     |
|   | <b>min() function</b>                                                                                                                                                                            |
| 3 | The function min() returns the smallest of its arguments i.e. the value closest to negative infinity, or smallest number from the iterable (list or tuple)                                       |
|   | <b>pow() function</b>                                                                                                                                                                            |
| 4 | The pow() function returns x raised to y. It is equivalent to $x**y$ . The function has third optional argument mod. If given, it returns $(x**y) \% \text{mod}$ value                           |
|   | <b>round() Function</b>                                                                                                                                                                          |
| 5 | round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.                                                                                               |
|   | <b>sum() function</b>                                                                                                                                                                            |
| 6 | The sum() function returns the sum of all numeric items in any iterable (list or tuple). An optional start argument is 0 by default. If given, the numbers in the list are added to start value. |

## Python - Strings

In Python, a string is an immutable sequence of Unicode characters. Each character has a unique numeric value as per the UNICODE standard. But, the sequence as a whole, doesn't have any numeric value even if all the characters are digits. To differentiate the string from numbers and other identifiers, the sequence of characters is included within single, double or triple quotes in its literal representation. Hence, 1234 is a number (integer) but '1234' is a string.

As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.

```
>>> 'Welcome To TutorialsPoint'
'Welcome To TutorialsPoint'
>>> "Welcome To TutorialsPoint"
'Welcome To TutorialsPoint'
>>> '''Welcome To TutorialsPoint'''
'Welcome To TutorialsPoint'
>>> """Welcome To TutorialsPoint"""
'Welcome To TutorialsPoint'
```

Looking at the above statements, it is clear that, internally Python stores strings as included in single quotes.

A string in Python is an object of str class. It can be verified with type() function.

```
var = "Welcome To TutorialsPoint"
print (type(var))
```

It will produce the following **output** –

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

You want to embed some text in double quotes as a part of string, the string itself should be put in single quotes. To embed a single quoted text, string should be written in double quotes.

```
var = 'Welcome to "Python Tutorial" from TutorialsPoint'
print ("var:", var)

var = "Welcome to 'Python Tutorial' from TutorialsPoint"
print ("var:", var)
```

To form a string with triple quotes, you may use triple single quotes, or triple double quotes – both versions are similar.

```
var = '''Welcome to TutorialsPoint'''
print ("var:", var)

var = """Welcome to TutorialsPoint"""
print ("var:", var)
```

Triple quoted string is useful to form a multi-line string.

```
</>  
  
var = '''
Welcome To
Python Tutorial
from TutorialsPoint
...
print ("var:", var)
```

[Open Compiler](#)

It will produce the following **output** –

```
var:
```

```
Welcome To  
Python Tutorial  
from TutorialsPoint
```

A string is a non-numeric data type. Obviously, we cannot use arithmetic operators with string operands. Python raises `TypeError` in such a case.

```
>>> "Hello"- "World"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

## Python Slicing Strings

In Python, a string is an ordered sequence of Unicode characters. Each character in the string has a unique index in the sequence. The index starts with 0. First character in the string has its positional index 0. The index keeps incrementing towards the end of string.

If a string variable is declared as `var="HELLO PYTHON"`, index of each character in the string is as follows –

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| H | E | L | L | O |   | P | Y | T | H | O  | N  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Python allows you to access any individual character from the string by its index. In this case, 0 is the lower bound and 11 is the upper bound of the string. So, `var[0]` returns H, `var[6]` returns P. If the index in square brackets exceeds the upper bound, Python raises `IndexError`.

```
>>> var="HELLO PYTHON"  
>>> var[0]  
'H'  
>>> var[7]  
'Y'  
>>> var[11]  
'N'  
>>> var[12]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range
```

One of the unique features of Python sequence types (and therefore a string object) it has a negative indexing scheme also. In the example above, a positive indexing scheme is used where the index increments from left to right. In case of negative indexing, the character at the end has -1 index and the index decrements from right to left, as a result the first character H has -12 index.

|     |     |     |    |    |    |    |    |    |    |    |    |
|-----|-----|-----|----|----|----|----|----|----|----|----|----|
| H   | E   | L   | L  | O  |    | P  | Y  | T  | H  | O  | N  |
| -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

Let us use negative indexing to fetch N, Y, and H characters.

```
>>> var[-1]
'N'
>>> var[-5]
'Y'
>>> var[-12]
'H'
>>> var[-13]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Once again, if the index goes beyond the range, IndexError is encountered.

We can therefore use positive or negative index to retrieve a character from the string.

```
>>> var[0], var[-12]
('H', 'H')
>>> var[7], var[-5]
('Y', 'Y')
>>> var[11], var[-1]
('N', 'N')
```

In Python, string is an immutable object. The object is immutable if it cannot be modified in-place, once stored in a certain memory location. You can retrieve any character from the string with the help of its index, but you cannot replace it with another character. In our example, character Y is at index 7 in HELLO PYTHON. Try to replace Y with y and see what happens.

&lt;/&gt;

Open Compiler

```
var="HELLO PYTHON"
var[7]="y"
```

```
print (var)
```

It will produce the following **output** –

```
Traceback (most recent call last):
File "C:\Users\users\example.py", line 2, in <module>
    var[7]="y"
    ~~~~^~~^
TypeError: 'str' object does not support item assignment
```

The TypeError is because the string is immutable.

Python defines ":" as string slicing operator. It returns a substring from the original string. Its general usage is –

```
substr=var[x:y]
```

The ":" operator needs two integer operands (both of which may be omitted, as we shall see in subsequent examples). The first operand x is the index of the first character of the desired slice. The second operand y is the index of the character next to the last in the desired string. So var(x:y] separates characters from xth position to (y-1)th position from the original string.

```
</>
```

Open Compiler

```
var="HELLO PYTHON"

print ("var:",var)
print ("var[3:8]:", var[3:8])
```

It will produce the following **output** –

```
var: HELLO PYTHON
var[3:8]: LO PY
```

Negative indexes can also be used for slicing.

```
</>
```

Open Compiler

```
var="HELLO PYTHON"
print ("var:",var)
```

```
print ("var[3:8]:", var[3:8])
print ("var[-9:-4]:", var[-9:-4])
```

It will produce the following **output** –

```
var: HELLO PYTHON
var[3:8]: LO PY
var[-9:-4]: LO PY
```

Both the operands for Python's Slice operator are optional. The first operand defaults to zero, which means if we do not give the first operand, the slice starts at 0th index, i.e. the first character. It slices the leftmost substring up to "y-1" characters.

&lt;/&gt;

Open Compiler

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[0:5]:", var[0:5])
print ("var[:5]:", var[:5])
```

It will produce the following **output** –

```
var: HELLO PYTHON
var[0:5]: HELLO
var[:5]: HELLO
```

Similarly, y operand is also optional. By default, it is "-1", which means the string will be sliced from the xth position up to the end of string.

&lt;/&gt;

Open Compiler

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[6:12]:", var[6:12])
print ("var[6:]:", var[6:])
```

It will produce the following output –

```
var: HELLO PYTHON
var[6:12]: PYTHON
```

```
var[6:]: PYTHON
```

Naturally, if both the operands are not used, the slice will be equal to the original string. That's because "x" is 0, and "y" is the last index+1 (or -1) by default.

```
</>
```

[Open Compiler](#)

```
var="HELLO PYTHON"  
print ("var:",var)  
print ("var[0:12]:", var[0:12])  
print ("var[:]:", var[:])
```

It will produce the following **output** –

```
var: HELLO PYTHON  
var[0:12]: HELLO PYTHON  
var[:]: HELLO PYTHON
```

The left operand must be smaller than the operand on right, for getting a substring of the original string. Python doesn't raise any error, if the left operand is greater, bu returns a null string.

```
</>
```

[Open Compiler](#)

```
var="HELLO PYTHON"  
print ("var:",var)  
print ("var[-1:7]:", var[-1:7])  
print ("var[7:0]:", var[7:0])
```

It will produce the following **output** –

```
var: HELLO PYTHON  
var[-1:7]:  
var[7:0]:
```

Slicing returns a new string. You can very well perform string operations like concatenation, or slicing on the sliced string.

```
</>
```

[Open Compiler](#)

```

var="HELLO PYTHON"

print ("var:",var)
print ("var[:6][:2]:", var[:6][:2])

var1=var[:6]
print ("slice:", var1)
print ("var1[:2]:", var1[:2])

```

It will produce the following **output** –

```

var: HELLO PYTHON
var[:6][:2]: HE
slice: HELLO
var1[:2]: HE

```

## Python - Modify Strings

In Python, a string (object of **str** class) is of immutable type. An immutable object is the one which can be modified in place, one created in the memory. Hence, unlike a list, any character in the sequence cannot be overwritten, nor can we insert or append characters to it unless we use certain string method that returns a new string object.

However, we can use one of the following tricks as a workaround to modify a string.

### Converting a String to a List

Since both string and list objects are sequences, they are interconvertible. Hence, if we cast a string object to a list, modify the list either by `insert()`, `append()` or `remove()` methods and convert the list back to a string, to get back the modified version.

We have a string variable `s1` with `WORD` as its value. With `list()` built-in function, let us convert it to a `l1` list object, and insert a character `L` at index 3. Then we use the `join()` method in `str` class to concatenate all the characters.

&lt;/&gt;

Open Compiler

```

s1="WORD"
print ("original string:", s1)
l1=list(s1)

```

```
l1.insert(3,"L")  
  
print (l1)  
  
s1=''.join(l1)  
print ("Modified string:", s1)
```

It will produce the following **output** –

```
original string: WORD  
['W', 'O', 'R', 'L', 'D']  
Modified string: WORLD
```

## Using the Array Module

To modify a string, construct an array object. Python standard library includes array module. We can have an array of Unicode type from a string variable.

```
import array as ar  
s1="WORD"  
sar=ar.array('u', s1)
```

Items in the array have a zero based index. So, we can perform array operations such as append, insert, remove etc. Let us insert L before the character D

```
sar.insert(3,"L")
```

Now, with the help of tounicode() method, get back the modified string

```
</>  
  
import array as ar  
  
s1="WORD"  
print ("original string:", s1)  
  
sar=ar.array('u', s1)  
sar.insert(3,"L")  
s1=sar.tounicode()
```

Open Compiler

```
print ("Modified string:", s1)
```

It will produce the following **output** –

```
original string: WORD  
Modified string: WORLD
```

## Using the StringIO Class

Python's io module defines the classes to handle streams. The StringIO class represents a text stream using an in-memory text buffer. A StringIO object obtained from a string behaves like a File object. Hence we can perform read/write operations on it. The getvalue() method of StringIO class returns a string.

Let us use this principle in the following program to modify a string.

```
</> Open Compiler  
  
import io  
  
s1="WORD"  
print ("original string:", s1)  
  
sio=io.StringIO(s1)  
sio.seek(3)  
sio.write("LD")  
s1=sio.getvalue()  
  
print ("Modified string:", s1)
```

It will produce the following **output** –

```
original string: WORD  
Modified string: WORLD
```

## Python - String Concatenation

The "+" operator is well-known as an addition operator, returning the sum of two numbers. However, the "+" symbol acts as string **concatenation operator** in Python. It

works with two string operands, and results in the concatenation of the two.

The characters of the string on the right of plus symbol are appended to the string on its left. Result of concatenation is a new string.

&lt;/&gt;

Open Compiler

```
str1="Hello"  
str2="World"  
print ("String 1:",str1)  
print ("String 2:",str2)  
str3=str1+str2  
print("String 3:",str3)
```

It will produce the following **output** –

```
String 1: Hello  
String 2: World  
String 3: HelloWorld
```

To insert a whitespace between the two, use a third empty string.

&lt;/&gt;

Open Compiler

```
str1="Hello"  
str2="World"  
blank=" "  
print ("String 1:",str1)  
print ("String 2:",str2)  
str3=str1+blank+str2  
print("String 3:",str3)
```

It will produce the following **output** –

```
String 1: Hello  
String 2: World  
String 3: Hello World
```

Another symbol \*, which we normally use for multiplication of two numbers, can also be used with string operands. Here, \* acts as a repetition operator in Python. One of the

operands must be an integer, and the second a string. The operator concatenates multiple copies of the string. For example –

```
>>> "Hello"*3  
'HelloHelloHello'
```

The integer operand is the number of copies of the string operand to be concatenated.

Both the string operators, (\*) the repetition operator and (+) the concatenation operator, can be used in a single expression. The "\*" operator has a higher precedence over the "+" operator.

```
str1="Hello"  
str2="World"  
print ("String 1:",str1)  
print ("String 2:",str2)  
str3=str1+str2*3  
print("String 3:",str3)  
str4=(str1+str2)*3  
print ("String 4:", str4)
```

To form **str3** string, Python concatenates 3 copies of World first, and then appends the result to Hello

String 3: HelloWorldWorldWorld

In the second case, the strings str1 and str2 are inside parentheses, hence their concatenation takes place first. Its result is then replicated three times.

String 4: HelloWorldHelloWorldHelloWorld

Apart from + and \*, no other arithmetic operator symbols can be used with string operands.

## Python - String Formatting

String formatting is the process of building a string representation dynamically by inserting the value of numeric expressions in an already existing string. Python's string concatenation operator doesn't accept a non-string operand. Hence, Python offers following string formatting techniques –

- Using % operator for substitution

- Using `format()` method of `str` class
- Using f-string syntax
- Using String Template class

## Python - Escape Characters

In Python, a string becomes a raw string if it is prefixed with "r" or "R" before the quotation symbols. Hence 'Hello' is a normal string whereas r'Hello' is a raw string.

```
>>> normal="Hello"
>>> print (normal)
Hello
>>> raw=r"Hello"
>>> print (raw)
Hello
```

In normal circumstances, there is no difference between the two. However, when the escape character is embedded in the string, the normal string actually interprets the escape sequence, whereas the raw string doesn't process the escape character.

```
>>> normal="Hello\nWorld"
>>> print (normal)
Hello
World
>>> raw=r"Hello\nWorld"
>>> print (raw)
Hello\nWorld
```

In the above example, when a normal string is printed the escape character '\n' is processed to introduce a newline. However, because of the raw string operator 'r' the effect of escape character is not translated as per its meaning.

The newline character \n is one of the escape sequences identified by Python. Escape sequence invokes an alternative implementation character subsequence to "\". In Python, "\\" is used as escape character. Following table shows list of escape sequences.

Unless an 'r' or 'R' prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are –

| Sr.No | Escape Sequence & Meaning |
|-------|---------------------------|
| 1     | \<newline>                |

|    |                                               |
|----|-----------------------------------------------|
|    | Backslash and newline ignored                 |
| 2  | <b>\\"</b><br>Backslash (\")                  |
| 3  | <b>\'</b><br>Single quote ('')                |
| 4  | <b>\\"</b><br>Double quote (")                |
| 5  | <b>\a</b><br>ASCII Bell (BEL)                 |
| 6  | <b>\b</b><br>ASCII Backspace (BS)             |
| 7  | <b>\f</b><br>ASCII Formfeed (FF)              |
| 8  | <b>\n</b><br>ASCII Linefeed (LF)              |
| 9  | <b>\r</b><br>ASCII Carriage Return (CR)       |
| 10 | <b>\t</b><br>ASCII Horizontal Tab (TAB)       |
| 11 | <b>\v</b><br>ASCII Vertical Tab (VT)          |
| 12 | <b>\ooo</b><br>Character with octal value ooo |
| 13 | <b>\xhh</b><br>Character with hex value hh    |

## Example

The following code shows the usage of escape sequences listed in the above table –

&lt;/&gt;

Open Compiler

```
# ignore \
s = 'This string will not include \
backslashes or newline characters.'
```

```
print (s)

# escape backslash
s=s = 'The \\character is called backslash'
print (s)

# escape single quote
s='Hello \'Python\''
print (s)

# escape double quote
s="Hello \"Python\""
print (s)

# escape \b to generate ASCII backspace
s='Hel\blo'
print (s)

# ASCII Bell character
s='Hello\a'
print (s)

# newline
s='Hello\nPython'
print (s)

# Horizontal tab
s='Hello\tPython'
print (s)

# form feed
s= "hello\fworld"
print (s)

# Octal notation
s="\101"
print(s)

# Hexadecimal notation
s="\x41"
print (s)
```

It will produce the following **output** –

This string will not include backslashes or newline characters.

The \character is called backslash

Hello 'Python'

Hello "Python"

Helo

Hello

Hello

Python

Hello Python

hello

world

A

A

## Python - String Methods

Python's built-in **str** class defines different methods. They help in manipulating strings. Since string is an immutable object, these methods return a copy of the original string, performing the respective processing on it.

The string methods can be classified in following categories –

- Case conversion
- Alignment
- Split and join
- Boolean
- Find and replace
- Formatting
- Translate

## Python - String Exercises

### Example 1

Python program to find number of vowels in a given string.

</>

Open Compiler

```
mystr = "All animals are equal. Some are more equal"
vowels = "aeiou"
count=0
for x in mystr:
    if x.lower() in vowels: count+=1
print ("Number of Vowels:", count)
```

It will produce the following **output** –

Number of Vowels: 18

## Example 2

Python program to convert a string with binary digits to integer.

```
</> Open Compiler

mystr = '10101'

def strtoint(mystr):
    for x in mystr:
        if x not in '01': return "Error. String with non-binary characters"
    num = int(mystr, 2)
    return num
print ("binary:{} integer: {}".format(mystr,strtoint(mystr)))
```

It will produce the following **output** –

binary:10101 integer: 21

Change **mystr** to '10, 101'

binary:10,101 integer: Error. String with non-binary characters

## Example 3

Python program to drop all digits from a string.

```
</> Open Compiler
```

```
digits = [str(x) for x in range(10)]
mystr = 'He121lo, Py00th55on!'
chars = []
for x in mystr:
    if x not in digits:
        chars.append(x)
newstr = ''.join(chars)
print (newstr)
```

It will produce the following **output** –

Hello, Python!

## Exercise Programs

- Python program to sort the characters in a string
- Python program to remove duplicate characters from a string
- Python program to list unique characters with their count in a string
- Python program to find number of words in a string
- Python program to remove all non-alphabetic characters from a string

# Python - Lists

List is one of the built-in data types in Python. A Python list is a sequence of comma separated items, enclosed in square brackets [ ]. The items in a Python list need not be of the same data type.

Following are some examples of Python lists –

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d"]
list4 = [25.50, True, -55, 1+2j]
```

In Python, a list is a sequence data type. It is an ordered collection of items. Each item in a list has a unique position index, starting from 0.

A list in Python is similar to an array in C, C++ or Java. However, the major difference is that in C/C++/Java, the array elements must be of same type. On the other hand, Python

lists may have objects of different data types.

A Python list is mutable. Any item from the list can be accessed using its index, and can be modified. One or more objects from the list can be removed or added. A list may have same item at more than one index positions.

## Python List Operations

In Python, List is a sequence. Hence, we can concatenate two lists with "+" operator and concatenate multiple copies of a list with "\*" operator. The membership operators "in" and "not in" work with list object.

| Python Expression     | Results                      | Description   |
|-----------------------|------------------------------|---------------|
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6]           | Concatenation |
| ['Hi!'] * 4           | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition    |
| 3 in [1, 2, 3]        | True                         | Membership    |

## Python - Access List Items

In Python, a list is a sequence. Each object in the list is accessible with its index. The index starts from 0. Index of the last item in the list is "length-1". To access the values in a list, use the square brackets for slicing along with the index or indices to obtain value available at that index.

The slice operator fetches one or more items from the list. Put index on square brackets to retrieve item at its position.

```
obj = list1[i]
```

### Example 1

Take a look at the following example –

</>

Open Compiler

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]

print ("Item at 0th index in list1: ", list1[0])
print ("Item at index 2 in list2: ", list2[2])
```

It will produce the following **output** –

Item at 0th index in list1: Rohan

Item at index 2 in list2: 3

Python allows negative index to be used with any sequence type. The "-1" index refers to the last item in the list.

## Example 2

Let's take another example –

</>

Open Compiler

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Item at 0th index in list1: ", list1[-1])
print ("Item at index 2 in list2: ", list2[-3])
```

It will produce the following **output** –

Item at 0th index in list1: d

Item at index 2 in list2: True

The slice operator extracts a sublist from the original list.

Sublist = list1[i:j]

## Parameters

- **i** – index of the first item in the sublist
- **j** – index of the item next to the last in the sublist

This will return a slice from  $i^{\text{th}}$  to  $(j-1)^{\text{th}}$  items from the **list1**.

## Example 3

While slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the list. Negative index can be used in slicing. Take a look at the following example

&lt;/&gt;

[Open Compiler](#)

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Items from index 1 to 2 in list1: ", list1[1:3])
print ("Items from index 0 to 1 in list2: ", list2[0:2])
```

It will produce the following **output** –

```
Items from index 1 to 2 in list1: ['b', 'c']
Items from index 0 to 1 in list2: [25.5, True]
```

## Example 4

&lt;/&gt;

[Open Compiler](#)

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]
list4 = ["Rohan", "Physics", 21, 69.75]
list3 = [1, 2, 3, 4, 5]

print ("Items from index 1 to last in list1: ", list1[1:])
print ("Items from index 0 to 1 in list2: ", list2[:2])
print ("Items from index 2 to last in list3", list3[2:-1])
print ("Items from index 0 to index last in list4", list4[:])
```

It will produce the following **output** –

```
Items from index 1 to last in list1: ['b', 'c', 'd']
Items from index 0 to 1 in list2: [25.5, True]
Items from index 2 to last in list3 [3, 4]
Items from index 0 to index last in list4 ['Rohan', 'Physics', 21, 69.75]
```

## Python - Change List Items

List is a mutable data type in Python. It means, the contents of list can be modified in place, after the object is stored in the memory. You can assign a new value at a given index position in the list

## Syntax

```
list1[i] = newvalue
```

## Example 1

In the following code, we change the value at index 2 of the given list.

```
</>
```

[Open Compiler](#)

```
list3 = [1, 2, 3, 4, 5]
print ("Original list ", list3)
list3[2] = 10
print ("List after changing value at index 2: ", list3)
```

It will produce the following **output** –

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

```
List after changing value at index 2: [1, 2, 10, 4, 5]
```

You can replace more consecutive items in a list with another sublist.

## Example 2

In the following code, items at index 1 and 2 are replaced by items in another sublist.

```
</>
```

[Open Compiler](#)

```
list1 = ["a", "b", "c", "d"]

print ("Original list: ", list1)

list2 = ['Y', 'Z']
list1[1:3] = list2

print ("List after changing with sublist: ", list1)
```

It will produce the following **output** –

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'Y', 'Z', 'd']
```

## Example 3

If the source sublist has more items than the slice to be replaced, the extra items in the source will be inserted. Take a look at the following code –

```
</> Open Compiler
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['X','Y', 'Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following **output** –

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'X', 'Y', 'Z', 'd']
```

## Example 4

If the sublist with which a slice of original list is to be replaced, has lesser items, the items with match will be replaced and rest of the items in original list will be removed.

In the following code, we try to replace "b" and "c" with "Z" (one less item than items to be replaced). It results in Z replacing b and c removed.

```
</> Open Compiler
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following **output** –

Original list: ['a', 'b', 'c', 'd']

List after changing with sublist: ['a', 'Z', 'd']

## Python - Add List Items

There are two methods of the **list** class, `append()` and `insert()`, that are used to add items to an existing list.

### Example 1

The **append()** method adds the item at the end of an existing list.

&lt;/&gt;

Open Compiler

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list1.append('e')
print ("List after appending: ", list1)
```

### Output

Original list: ['a', 'b', 'c', 'd']

List after appending: ['a', 'b', 'c', 'd', 'e']

### Example 2

The **insert()** method inserts the item at a specified index in the list.

&lt;/&gt;

Open Compiler

```
list1 = ["Rohan", "Physics", 21, 69.75]
print ("Original list ", list1)

list1.insert(2, 'Chemistry')
print ("List after appending: ", list1)

list1.insert(-1, 'Pass')
print ("List after appending: ", list1)
```

## Output

```
Original list ['Rohan', 'Physics', 21, 69.75]
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 69.75]
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 'Pass', 69.75]
```

We know that "-1" index points to the last item in the list. However, note that, the item at index "-1" in the original list is 69.75. This index is not refreshed after appending 'chemistry'. Hence, 'Pass' is not inserted at the updated index "-1", but the previous index "-1".

## Python - Remove List Items

The list class methods **remove()** and **pop()** both can remove an item from a list. The difference between them is that remove() removes the object given as argument, while pop() removes an item at the given index.

### Using the remove() Method

The following example shows how you can use the remove() method to remove list items –

&lt;/&gt;

Open Compiler

```
list1 = ["Rohan", "Physics", 21, 69.75]
print ("Original list: ", list1)

list1.remove("Physics")
print ("List after removing: ", list1)
```

It will produce the following **output** –

```
Original list: ['Rohan', 'Physics', 21, 69.75]
List after removing: ['Rohan', 21, 69.75]
```

### Using the pop() Method

The following example shows how you can use the pop() method to remove list items –

```
</>

list2 = [25.50, True, -55, 1+2j]
print ("Original list: ", list2)
list2.pop(2)
print ("List after popping: ", list2)
```

It will produce the following **output** –

```
Original list: [25.5, True, -55, (1+2j)]
List after popping: [25.5, True, (1+2j)]
```

## Using the "del" Keyword

Python has the "del" keyword that deletes any Python object from the memory.

### Example

We can use "del" to delete an item from a list. Take a look at the following example –

```
</>

list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
del list1[2]
print ("List after deleting: ", list1)
```

It will produce the following **output** –

```
Original list: ['a', 'b', 'c', 'd']
List after deleting: ['a', 'b', 'd']
```

### Example

You can delete a series of consecutive items from a list with the slicing operator. Take a look at the following example –

```
</>
```

```
list2 = [25.50, True, -55, 1+2j]
print ("List before deleting: ", list2)
del list2[0:2]
print ("List after deleting: ", list2)
```

It will produce the following **output** –

```
List before deleting: [25.5, True, -55, (1+2j)]
List after deleting: [-55, (1+2j)]
```

## Python - Loop Lists

You can traverse the items in a list with Python's **for** loop construct. The traversal can be done, using list as an iterator or with the help of index.

### Syntax

Python list gives an iterator object. To iterate a list, use the for statement as follows –

```
for obj in list:
    ...
    ...
```

### Example 1

Take a look at the following example –

</>

Open Compiler

```
lst = [25, 12, 10, -21, 10, 100]
for num in lst:
    print (num, end = ' ')
```

### Output

```
25 12 10 -21 10 100
```

### Example 2

To iterate through the items in a list, obtain the range object of integers "0" to "len-1". See the following example –

```
</> Open Compiler  
  
lst = [25, 12, 10, -21, 10, 100]  
indices = range(len(lst))  
for i in indices:  
    print ("lst[{}]: {}".format(i), lst[i])
```

## Output

```
lst[0]: 25  
lst[1]: 12  
lst[2]: 10  
lst[3]: -21  
lst[4]: 10  
lst[5]: 100
```

# Python - List Comprehension

List comprehension is a very powerful programming tool. It is similar to set builder notation in mathematics. It is a concise way to create new list by performing some kind of process on each item on existing list. List comprehension is considerably faster than processing a list by **for** loop.

## Example 1

Suppose we want to separate each letter in a string and put all non-vowel letters in a list object. We can do it by a **for** loop as shown below –

```
</> Open Compiler  
  
chars=[]  
for ch in 'TutorialsPoint':  
    if ch not in 'aeiou':  
        chars.append(ch)  
print (chars)
```

The chars list object is displayed as follows –

```
[T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

## List Comprehension Technique

We can easily get the same result by a list comprehension technique. A general usage of list comprehension is as follows –

```
listObj = [x for x in iterable]
```

Applying this, chars list can be constructed by the following statement –

```
</>
```

[Open Compiler](#)

```
chars = [char for char in 'TutorialsPoint' if char not in 'aeiou']  
print(chars)
```

The chars list will be displayed as before –

```
[T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

## Example 2

The following example uses list comprehension to build a list of squares of numbers between 1 to 10

```
</>
```

[Open Compiler](#)

```
squares = [x*x for x in range(1,11)]  
print(squares)
```

The squares list object is –

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

## Nested Loops in List Comprehension

In the following example, all combinations of items from two lists in the form of a tuple are added in a third list object.

## Example 3

```
</>  
list1=[1,2,3]  
list2=[4,5,6]  
CombLst=[(x,y) for x in list1 for y in list2]  
print (CombLst)
```

[Open Compiler](#)

It will produce the following **output** –

```
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
```

## Condition in List Comprehension

The following statement will create a list of all even numbers between 1 to 20.

## Example 4

```
</>  
list1=[x for x in range(1,21) if x%2==0]  
print (list1)
```

[Open Compiler](#)

It will produce the following **output** –

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

# Python - Sort Lists

The `sort()` method of list class rearranges the items in ascending or descending order with the use of lexicographical ordering mechanism. The sorting is in-place, in the sense the rearrangement takes place in the same list object, and that it doesn't return a new object.

## Syntax

```
list1.sort(key, reverse)
```

## Parameters

- **Key** – The function applied to each item in the list. The return value is used to perform sort. Optional
- **reverse** – Boolean value. If set to True, the sort takes place in descending order. Optional

## Return value

This method returns None.

## Example 1

Now let's take a look at some examples to understand how we can sort lists in Python –

&lt;/&gt;

Open Compiler

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
print ("list before sort", list1)
list1.sort()
print ("list after sort : ", list1)

print ("Descending sort")

list2 = [10,16, 9, 24, 5]
print ("list before sort", list2)
list2.sort()
print ("list after sort : ", list2)
```

It will produce the following **output** –

```
list before sort ['physics', 'Biology', 'chemistry', 'maths']
list after sort: ['Biology', 'chemistry', 'maths', 'physics']
Descending sort
list before sort [10, 16, 9, 24, 5]
list after sort : [5, 9, 10, 16, 24]
```

## Example 2

In this example, the str.lower() method is used as key parameter in sort() method.

&lt;/&gt;

[Open Compiler](#)

```
list1 = ['Physics', 'biology', 'Biomechanics', 'psychology']
print ("list before sort", list1)
list1.sort(key=str.lower)
print ("list after sort : ", list1)
```

It will produce the following **output** –

```
list before sort ['Physics', 'biology', 'Biomechanics', 'psychology']
list after sort : ['biology', 'Biomechanics', 'Physics', 'psychology']
```

## Example 3

Let us use a user-defined function as the key parameter in `sort()` method. The `myfunction()` uses `%` operator to return the remainder, based on which the sort is done.

&lt;/&gt;

[Open Compiler](#)

```
def myfunction(x):
    return x%10
list1 = [17, 23, 46, 51, 90]
print ("list before sort", list1)
list1.sort(key=myfunction)
print ("list after sort : ", list1)
```

It will produce the following **output** –

```
list before sort [17, 23, 46, 51, 90]
list after sort: [90, 51, 23, 46, 17]
```

## Python - Copy Lists

In Python, a variable is just a label or reference to the object in the memory. Hence, the assignment "`lst1 = lst`" refers to the same list object in the memory. Take a look at the following example –

&lt;/&gt;

[Open Compiler](#)

```
lst = [10, 20]
print ("lst:", lst, "id(lst):", id(lst))
lst1 = lst
print ("lst1:", lst1, "id(lst1):", id(lst1))
```

It will produce the following **output** –

```
lst: [10, 20] id(lst): 1677677188288
lst1: [10, 20] id(lst1): 1677677188288
```

As a result, if we update "lst", it will automatically reflect in "lst1". Change lst[0] to 100

```
lst[0]=100
print ("lst:", lst, "id(lst):", id(lst))
print ("lst1:", lst1, "id(lst1):", id(lst1))
```

It will produce the following **output** –

```
lst: [100, 20] id(lst): 1677677188288
lst1: [100, 20] id(lst1): 1677677188288
```

Hence, we can say that "lst1" is not the physical copy of "lst".

## Using the Copy Method of List Class

Python's list class has a `copy()` method to create a new physical copy of a list object.

### Syntax

```
lst1 = lst.copy()
```

The new list object will have a different `id()` value. The following example demonstrates this –

```
lst = [10, 20]
lst1 = lst.copy()
print ("lst:", lst, "id(lst):", id(lst))
print ("lst1:", lst1, "id(lst1):", id(lst1))
```

It will produce the following **output** –

```
lst: [10, 20] id(lst): 1677678705472
lst1: [10, 20] id(lst1): 1677678706304
```

Even if the two lists have same data, they have different `id()` value, hence they are two different objects and "lst1" is a copy of "lst".

If we try to modify "lst", it will not reflect in "lst1". See the following example –

```
lst[0]=100
print ("lst:", lst, "id(lst):",id(lst))
print ("lst1:", lst1, "id(lst1):",id(lst1))
```

It will produce the following **output** –

```
lst: [100, 20] id(lst): 1677678705472
lst1: [10, 20] id(lst1): 1677678706304
```

## Python - Join Lists

In Python, List is classified as a sequence type object. It is a collection of items, which may be of different data types, with each item having a positional index starting with 0. You can use different ways to join two Python lists.

All the sequence type objects support concatenation operator, with which two lists can be joined.

```
</> Open Compiler
L1 = [10,20,30,40]
L2 = ['one', 'two', 'three', 'four']
L3 = L1+L2
print ("Joined list:", L3)
```

It will produce the following **output** –

```
Joined list: [10, 20, 30, 40, 'one', 'two', 'three', 'four']
```

You can also use the augmented concatenation operator with "+=" symbol to append L2 to L1

[Open Compiler](#)

```
</>  
L1 = [10,20,30,40]  
L2 = ['one', 'two', 'three', 'four']  
L1+=L2  
print ("Joined list:", L1)
```

The same result can be obtained by using the extend() method. Here, we need to extend L1 so as to add elements from L2 in it.

```
</>  
L1 = [10,20,30,40]  
L2 = ['one', 'two', 'three', 'four']  
L1.extend(L2)  
print ("Joined list:", L1)
```

To add items from one list to another, a classical iterative solution also works. Traverse items of second list with a for loop, and append each item in the first.

```
</>  
L1 = [10,20,30,40]  
L2 = ['one', 'two', 'three', 'four']  
  
for x in L2:  
    L1.append(x)  
  
print ("Joined list:", L1)
```

A slightly complex approach for merging two lists is using list comprehension, as following code shows –

```
L1 = [10,20,30,40]  
L2 = ['one', 'two', 'three', 'four']  
L3 = [y for x in [L1, L2] for y in x]  
print ("Joined list:", L3)
```

## Python - List Methods

Python's **list** class includes the following methods using which you can add, update, and delete list items –

| Sr.No | Methods & Description                                                             |
|-------|-----------------------------------------------------------------------------------|
| 1     | <b>list.append(obj)</b><br>Appends object obj to list                             |
| 2     | <b>list.clear()</b><br>Clears the contents of list                                |
| 3     | <b>list.copy()</b><br>Returns a copy of the list object                           |
| 4     | <b>list.count(obj)</b><br>Returns count of how many times obj occurs in list      |
| 5     | <b>list.extend(seq)</b><br>Appends the contents of seq to list                    |
| 6     | <b>list.index(obj)</b><br>Returns the lowest index in list that obj appears       |
| 7     | <b>list.insert(index, obj)</b><br>Inserts object obj into list at offset index    |
| 8     | <b>list.pop(obj=list[-1])</b><br>Removes and returns last object or obj from list |
| 9     | <b>list.remove(obj)</b><br>Removes object obj from list                           |
| 10    | <b>list.reverse()</b><br>Reverses objects of list in place                        |
| 11    | <b>list.sort([func])</b><br>Sorts objects of list, use compare func if given      |

## Python - List Exercises

### Example 1

Python program to find unique numbers in a given list.

&lt;/&gt;

Open Compiler

```
L1 = [1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2]
L2 = []
for x in L1:
    if x not in L2:
        L2.append(x)
print (L2)
```

It will produce the following **output** –

[1, 9, 6, 3, 4, 5, 2, 7, 8]

## Example 2

Python program to find sum of all numbers in a list.

</>

Open Compiler

```
L1 = [1, 9, 1, 6, 3, 4]
ttl = 0
for x in L1:
    ttl+=x
print ("Sum of all numbers Using loop:", ttl)
ttl = sum(L1)
print ("Sum of all numbers sum() function:", ttl)
```

It will produce the following **output** –

Sum of all numbers Using loop: 24  
 Sum of all numbers sum() function: 24

## Example 3

Python program to create a list of 5 random integers.

</>

Open Compiler

```
import random
L1 = []
for i in range(5):
    x = random.randint(0, 100)
```

```
L1.append(x)
print (L1)
```

It will produce the following **output** –

```
[77, 3, 20, 91, 85]
```

## Exercise Programs

- Python program to remove all odd numbers from a list.
- Python program to sort a list of strings on the number of alphabets in each word.
- Python program non-numeric items in a list in a separate list.
- Python program to create a list of integers representing each character in a string
- Python program to find numbers common in two lists.

## Python - Tuples

Tuple is one of the built-in data types in Python. A Python tuple is a sequence of comma separated items, enclosed in parentheses (). The items in a Python tuple need not be of same data type.

Following are some examples of Python tuples –

```
tup1 = ("Rohan", "Physics", 21, 69.75)
tup2 = (1, 2, 3, 4, 5)
tup3 = ("a", "b", "c", "d")
tup4 = (25.50, True, -55, 1+2j)
```

In Python, tuple is a sequence data type. It is an ordered collection of items. Each item in the tuple has a unique position index, starting from 0.

In C/C++/Java array, the array elements must be of same type. On the other hand, Python tuple may have objects of different data types.

Python tuple and list both are sequences. One major difference between the two is, Python list is mutable, whereas tuple is immutable. Although any item from the tuple can be accessed using its index, and cannot be modified, removed or added.

## Python Tuple Operations

In Python, Tuple is a sequence. Hence, we can concatenate two tuples with `+` operator and concatenate multiple copies of a tuple with `*` operator. The membership operators "`in`" and "`not in`" work with tuple object.

| Python Expression                  | Results                                   | Description   |
|------------------------------------|-------------------------------------------|---------------|
| <code>(1, 2, 3) + (4, 5, 6)</code> | <code>(1, 2, 3, 4, 5, 6)</code>           | Concatenation |
| <code>('Hi!',) * 4</code>          | <code>('Hi!', 'Hi!', 'Hi!', 'Hi!')</code> | Repetition    |
| <code>3 in (1, 2, 3)</code>        | <code>True</code>                         | Membership    |

Note that even if there is only one object in a tuple, you must give a comma after it. Otherwise, it is treated as a string.

## Python - Access Tuple Items

In Python, Tuple is a sequence. Each object in the list is accessible with its index. The index starts from "0". Index or the last item in the tuple is "length-1". To access values in tuples, use the square brackets for slicing along with the index or indices to obtain value available at that index.

The slice operator fetches one or more items from the tuple.

```
obj = tup1(i)
```

### Example 1

Put the index inside square brackets to retrieve the item at its position.

</>

Open Compiler

```
tup1 = ("Rohan", "Physics", 21, 69.75)
tup2 = (1, 2, 3, 4, 5)

print ("Item at 0th index in tup1: ", tup1[0])
print ("Item at index 2 in list2: ", tup2[2])
```

It will produce the following **output** –

Item at 0th index in tup1: Rohan  
 Item at index 2 in tup2: 3

## Example 2

Python allows negative index to be used with any sequence type. The "-1" index refers to the last item in the tuple.

&lt;/&gt;

Open Compiler

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)
print ("Item at 0th index in tup1: ", tup1[-1])
print ("Item at index 2 in tup2: ", tup2[-3])
```

It will produce the following **output** –

```
Item at 0th index in tup1: d
Item at index 2 in tup2: True
```

## Extracting a Subtuple from a Tuple

The slice operator extracts a subtuple from the original tuple.

```
Subtup = tup1[i:j]
```

### Parameters

- **i** – index of the first item in the subtup
- **j** – index of the item next to the last in the subtup

This will return a slice from  $i^{\text{th}}$  to  $(j-1)^{\text{th}}$  items from the `tup1`.

## Example 3

Take a look at the following example –

&lt;/&gt;

Open Compiler

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)
```

```
print ("Items from index 1 to 2 in tup1: ", tup1[1:3])
print ("Items from index 0 to 1 in tup2: ", tup2[0:2])
```

It will produce the following **output** –

```
Items from index 1 to 2 in tup1: ('b', 'c')
Items from index 0 to 1 in tup2: (25.5, True)
```

## Example 4

While slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the tuple. Negative index can be used in slicing. See the following example –

</>

Open Compiler

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)
tup4 = ("Rohan", "Physics", 21, 69.75)
tup3 = (1, 2, 3, 4, 5)

print ("Items from index 1 to last in tup1: ", tup1[1:])
print ("Items from index 0 to 1 in tup2: ", tup2[:2])
print ("Items from index 2 to last in tup3: ", tup3[2:-1])
print ("Items from index 0 to index last in tup4: ", tup4[:])
```

It will produce the following **output** –

```
Items from index 1 to last in tup1: ('b', 'c', 'd')
Items from index 0 to 1 in tup2: (25.5, True)
Items from index 2 to last in tup3: (3, 4)
Items from index 0 to index last in tup4: ('Rohan', 'Physics', 21, 69.75)
```

## Python - Update Tuples

In Python, tuple is an immutable data type. An immutable object cannot be modified once it is created in the memory.

## Example 1

If we try to assign a new value to a tuple item with slice operator, Python raises `TypeError`. See the following example –

```
tup1 = ("a", "b", "c", "d")
tup1[2] = 'Z'
print ("tup1: ", tup1)
```

It will produce the following **output** –

```
Traceback (most recent call last):
File "C:\Users\mlath\examples\main.py", line 2, in <module>
  tup1[2] = 'Z'
  ~~~~~^~~^
TypeError: 'tuple' object does not support item assignment
```

Hence, it is not possible to update a tuple. Therefore, the tuple class doesn't provide methods for adding, inserting, deleting, sorting items from a tuple object, as the list class.

## How to Update a Python Tuple?

You can use a work-around to update a tuple. Using the `list()` function, convert the tuple to a list, perform the desired append/insert/remove operations and then parse the list back to tuple object.

### Example 2

Here, we convert the tuple to a list, update an existing item, append a new item and sort the list. The list is converted back to tuple.

</>

[Open Compiler](#)

```
tup1 = ("a", "b", "c", "d")
print ("Tuple before update", tup1, "id(): ", id(tup1))

list1 = list(tup1)
list1[2]='F'
list1.append('Z')
list1.sort()
print ("updated list", list1)

tup1 = tuple(list1)
print ("Tuple after update", tup1, "id(): ", id(tup1))
```

It will produce the following **output** –

```
Tuple before update ('a', 'b', 'c', 'd') id(): 2295023084192
updated list ['F', 'Z', 'a', 'b', 'd']
Tuple after update ('F', 'Z', 'a', 'b', 'd') id(): 2295021518128
```

However, note that the `id()` of `tup1` before update and after update are different. It means that a new tuple object is created and the original tuple object is not modified in-place.

## Python - Unpack Tuple Items

The term "unpacking" refers to the process of parsing tuple items in individual variables. In Python, the parentheses are the default delimiters for a literal representation of sequence object.

Following statements to declare a tuple are identical.

```
>>> t1 = (x,y)
>>> t1 = x,y
>>> type (t1)
<class 'tuple'>
```

### Example 1

To store tuple items in individual variables, use multiple variables on the left of assignment operator, as shown in the following example –

```
</> Open Compiler
```

```
tup1 = (10,20,30)
x, y, z = tup1
print ("x: ", x, "y: ", "z: ",z)
```

It will produce the following **output** –

```
x: 10 y: 20 z: 30
```

That's how the tuple is unpacked in individual variables.

### Using to Unpack a Tuple

In the above example, the number of variables on the left of assignment operator is equal to the items in the tuple. What if the number is not equal to the items?

## Example 2

If the number of variables is more or less than the length of tuple, Python raises a `ValueError`.

&lt;/&gt;

Open Compiler

```
tup1 = (10, 20, 30)
x, y = tup1
x, y, p, q = tup1
```

It will produce the following **output** –

```
x, y = tup1
```

```
^^^^^
```

```
ValueError: too many values to unpack (expected 2)
```

```
x, y, p, q = tup1
```

```
^^^^^^^^^^^^^
```

```
ValueError: not enough values to unpack (expected 4, got 3)
```

In such a case, the "\*" symbol is used for unpacking. Prefix "\*" to "y", as shown below –

&lt;/&gt;

Open Compiler

```
tup1 = (10, 20, 30)
x, *y = tup1
print ("x: ", "y: ", y)
```

It will produce the following **output** –

```
x: y: [20, 30]
```

The first value in tuple is assigned to "x", and rest of items to "y" which becomes a list.

## Example 3

In this example, the tuple contains 6 values and variables to be unpacked are 3. We prefix "\*" to the second variable.

&lt;/&gt;

[Open Compiler](#)

```
tup1 = (10,20,30, 40, 50, 60)
x, *y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following **output** –

```
x: 10 y: [20, 30, 40, 50] z: 60
```

Here, values are unpacked in "x" and "z" first, and then the rest of values are assigned to "y" as a list.

## Example 4

What if we add "\*" to the first variable?

&lt;/&gt;

[Open Compiler](#)

```
tup1 = (10,20,30, 40, 50, 60)
*x, y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following **output** –

```
x: [10, 20, 30, 40] y: 50 z: 60
```

Here again, the tuple is unpacked in such a way that individual variables take up the value first, leaving the remaining values to the list "x".

# Python - Loop Tuples

You can traverse the items in a tuple with Python's **for** loop construct. The traversal can be done, using tuple as an iterator or with the help of index.

## Syntax

Python tuple gives an iterator object. To iterate a tuple, use the **for** statement as follows –

```
for obj in tuple:  
    . . .  
    . . .
```

## Example 1

The following example shows a simple Python **for** loop construct –

&lt;/&gt;

Open Compiler

```
tup1 = (25, 12, 10, -21, 10, 100)  
for num in tup1:  
    print (num, end = ' ')
```

It will produce the following **output** –

25 12 10 -21 10 100

## Example 2

To iterate through the items in a tuple, obtain the range object of integers "0" to "len-1".

&lt;/&gt;

Open Compiler

```
tup1 = (25, 12, 10, -21, 10, 100)  
indices = range(len(tup1))  
for i in indices:  
    print ("tup1[{}]: {}".format(i), tup1[i])
```

It will produce the following **output** –

```
tup1[0]: 25  
tup1 [1]: 12  
tup1 [2]: 10  
tup1 [3]: -21  
tup1 [4]: 10  
tup1 [5]: 100
```

# Python - Join Tuples

In Python, a Tuple is classified as a sequence type object. It is a collection of items, which may be of different data types, with each item having a positional index starting with 0. Although this definition also applies to a list, there are two major differences in list and tuple. First, while items are placed in square brackets in case of List (example: [10,20,30,40]), the tuple is formed by putting the items in parentheses (example: (10,20,30,40)).

In Python, a Tuple is an immutable object. Hence, it is not possible to modify the contents of a tuple once it is formed in the memory.

However, you can use different ways to join two Python tuples.

## Example 1

All the sequence type objects support concatenation operator, with which two lists can be joined.

```
</> Open Compiler  
  
T1 = (10,20,30,40)  
T2 = ('one', 'two', 'three', 'four')  
T3 = T1+T2  
print ("Joined Tuple:", T3)
```

It will produce the following **output** –

```
Joined Tuple: (10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

## Example 2

You can also use the augmented concatenation operator with the "+=" symbol to append T2 to T1

```
</> Open Compiler  
  
T1 = (10,20,30,40)  
T2 = ('one', 'two', 'three', 'four')  
T1+=T2  
print ("Joined Tuple:", T1)
```

## Example 3

The same result can be obtained by using the `extend()` method. Here, we need cast the two tuple objects to lists, extend so as to add elements from one list to another, and convert the joined list back to a tuple.

```
</> Open Compiler

T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
L1 = list(T1)
L2 = list(T2)
L1.extend(L2)
T1 = tuple(L1)
print ("Joined Tuple:", T1)
```

## Example 4

Python's built-in `sum()` function also helps in concatenating tuples. We use an expression

```
sum((t1, t2), ())
```

The elements of the first tuple are appended to an empty tuple first, and then elements from second tuple are appended and returns a new tuple that is concatenation of the two.

```
</> Open Compiler

T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
T3 = sum((T1, T2), ())
print ("Joined Tuple:", T3)
```

## Example 5

A slightly complex approach for merging two tuples is using list comprehension, as following code shows –

```
</> Open Compiler

T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
L1, L2 = list(T1), list(T2)
L3 = [y for x in [L1, L2] for y in x]
```

```
T3 = tuple(L3)
print ("Joined Tuple:", T3)
```

## Example 6

You can run a **for** loop on the items in second loop, convert each item in a single item tuple and concatenate it to first tuple with the "+=" operator

&lt;/&gt;

Open Compiler

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
for t in T2:
    T1+= (t,)
print (T1)
```

# Python - Tuple Methods

Since a tuple in Python is immutable, the tuple class doesn't define methods for adding or removing items. The tuple class defines only two methods.

| Sr.No | Methods & Description                                                          |
|-------|--------------------------------------------------------------------------------|
| 1     | <b>tuple.count(obj)</b><br>Returns count of how many times obj occurs in tuple |
| 2     | <b>tuple.index(obj)</b><br>Returns the lowest index in tuple that obj appears  |

## Finding the Index of a Tuple Item

The `index()` method of tuple class returns the index of first occurrence of the given item.

### Syntax

```
tuple.index(obj)
```

### Return value

The `index()` method returns an integer, representing the index of the first occurrence of "obj".

## Example

Take a look at the following example –

```
</> Open Compiler
tup1 = (25, 12, 10, -21, 10, 100)
print ("Tup1:", tup1)
x = tup1.index(10)
print ("First index of 10:", x)
```

It will produce the following **output** –

```
Tup1: (25, 12, 10, -21, 10, 100)
First index of 10: 2
```

## Counting Tuple Items

The count() method in tuple class returns the number of times a given object occurs in the tuple.

### Syntax

```
tuple.count(obj)
```

### Return Value

Number of occurrence of the object. The count() method returns an integer.

## Example

```
</> Open Compiler
tup1 = (10, 20, 45, 10, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(10)
print ("count of 10:", c)
```

It will produce the following **output** –

```
Tup1: (10, 20, 45, 10, 30, 10, 55)
count of 10: 3
```

## Example

Even if the items in the tuple contain expressions, they will be evaluated to obtain the count.

```
Tup1 = (10, 20/80, 0.25, 10/40, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(0.25)
print ("count of 10:", c)
```

It will produce the following **output** –

```
Tup1: (10, 0.25, 0.25, 0.25, 30, 10, 55)
count of 10: 3
```

# Python Tuple Exercises

## Example 1

Python program to find unique numbers in a given tuple –

```
</> Open Compiler
T1 = (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
T2 = ()
for x in T1:
    if x not in T2:
        T2+=x
print ("original tuple:", T1)
print ("Unique numbers:", T2)
```

It will produce the following **output** –

```
original tuple: (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
Unique numbers: (1, 9, 6, 3, 4, 5, 2, 7, 8)
```

## Example 2

Python program to find sum of all numbers in a tuple –

```
</> Open Compiler  
  
T1 = (1, 9, 1, 6, 3, 4)  
ttl = 0  
for x in T1:  
    ttl+=x  
  
print ("Sum of all numbers Using loop:", ttl)  
  
ttl = sum(T1)  
print ("Sum of all numbers sum() function:", ttl)
```

It will produce the following **output** –

```
Sum of all numbers Using loop: 24  
Sum of all numbers sum() function: 24
```

## Example 3

Python program to create a tuple of 5 random integers –

```
</> Open Compiler  
  
import random  
t1 = ()  
for i in range(5):  
    x = random.randint(0, 100)  
    t1+=(x,)  
print (t1)
```

It will produce the following **output** –

```
(64, 21, 68, 6, 12)
```

## Exercise Programs

- Python program to remove all duplicates numbers from a list.
- Python program to sort a tuple of strings on the number of alphabets in each word.
- Python program to prepare a tuple of non-numeric items from a given tuple.
- Python program to create a tuple of integers representing each character in a string
- Python program to find numbers common in two tuples.

## Python - Sets

A set is one of the built-in data types in Python. In mathematics, set is a collection of distinct objects. Set data type is Python's implementation of a set. Objects in a set can be of any data type.

Set in Python also a collection data type such as list or tuple. However, it is not an ordered collection, i.e., items in a set are not accessible by its positional index. A set object is a collection of one or more immutable objects enclosed within curly brackets {}.

### Example 1

Some examples of set objects are given below –

```
</> Open Compiler
s1 = {"Rohan", "Physics", 21, 69.75}
s2 = {1, 2, 3, 4, 5}
s3 = {"a", "b", "c", "d"}
s4 = {25.5, True, -55, 1+2j}
print (s1)
print (s2)
print (s3)
print (s4)
```

It will produce the following **output** –

```
{'Physics', 21, 'Rohan', 69.75}
{1, 2, 3, 4, 5}
{'a', 'd', 'c', 'b'}
{25.5, -55, True, (1+2j)}
```

The above result shows that the order of objects in the assignment is not necessarily retained in the set object. This is because Python optimizes the structure of set for set operations.

In addition to the literal representation of set (keeping the items inside curly brackets), Python's built-in `set()` function also constructs set object.

## set() Function

`set()` is one of the built-in functions. It takes any sequence object (list, tuple or string) as argument and returns a set object

### Syntax

```
Obj = set(sequence)
```

### Parameters

- **sequence** – An object of list, tuple or str type

### Return value

The `set()` function returns a set object from the sequence, discarding the repeated elements in it.

### Example 2

```
</>
```

Open Compiler

```
L1 = ["Rohan", "Physics", 21, 69.75]
s1 = set(L1)
T1 = (1, 2, 3, 4, 5)
s2 = set(T1)
string = "TutorialsPoint"
s3 = set(string)

print (s1)
print (s2)
print (s3)
```

It will produce the following **output** –

```
{'Rohan', 69.75, 21, 'Physics'}  
{1, 2, 3, 4, 5}  
{'u', 'a', 'o', 'n', 'r', 's', 'T', 'P', 'i', 't', 'l'}
```

## Example 3

Set is a collection of distinct objects. Even if you repeat an object in the collection, only one copy is retained in it.

```
</> Open Compiler  
  
s2 = {1, 2, 3, 4, 5, 3, 0, 1, 9}  
s3 = {"a", "b", "c", "d", "b", "e", "a"}  
print (s2)  
print (s3)
```

It will produce the following **output** –

```
{0, 1, 2, 3, 4, 5, 9}  
{'a', 'b', 'd', 'c', 'e'}
```

## Example 4

Only immutable objects can be used to form a set object. Any number type, string and tuple is allowed, but you cannot put a list or a dictionary in a set.

```
</> Open Compiler  
  
s1 = {1, 2, [3, 4, 5], 3, 0, 1, 9}  
print (s1)  
s2 = {"Rohan", {"phy":50}}  
print (s2)
```

It will produce the following **output** –

```
s1 = {1, 2, [3, 4, 5], 3, 0, 1, 9}  
^^^^^^^^^^^^^^^^^^^^  
TypeError: unhashable type: 'list'  
s2 = {"Rohan", {"phy":50}}
```

```
^^^^^^^^^^^^^^^^^^^^^
```

TypeError: unhashable type: 'dict'

Python raises TypeError with a message unhashable types 'list' or 'dict'. Hashing generates a unique number for an immutable item that enables quick search inside computer's memory. Python has built-in hash() function. This function is not supported by list or dictionary.

Even though mutable objects are not stored in a set, set itself is a mutable object. Python has a special operators to work with sets, and there are different methods in set class to perform add, remove, update operations on elements of a set object.

## Python - Access Set Items

Since set is not a sequence data type, its items cannot be accessed individually as they do not have a positional index (as in list or tuple). Set items do not have a key either (as in dictionary) to access. You can only traverse the set items using a **for** loop.

### Example 1

```
</>
```

Open Compiler

```
langs = {"C", "C++", "Java", "Python"}  
for lang in langs:  
    print (lang)
```

It will produce the following **output** –

```
Python  
C  
C++  
Java
```

### Example 2

Python's membership operators let you check if a certain item is available in the set. Take a look at the following example –

```
</>
```

Open Compiler

```
langs = {"C", "C++", "Java", "Python"}  
print ("PHP" in langs)  
print ("Java" in langs)
```

It will produce the following **output** –

False  
True

## Python - Add Set Items

Even if a set holds together only immutable objects, set itself is mutable. We can add new items in it with any of the following ways –

### add() Method

The add() method in set class adds a new element. If the element is already present in the set, there is no change in the set.

### Syntax

```
set.add(obj)
```

### Parameters

- **obj** – an object of any immutable type.

### Example

Take a look at the following example –

```
</>
```

Open Compiler

```
lang1 = {"C", "C++", "Java", "Python"}  
lang1.add("Golang")  
print (lang1)
```

It will produce the following **output** –

```
{'Python', 'C', 'Golang', 'C++', 'Java'}
```

## update() Method

The update() method of set class includes the items of the set given as argument. If elements in the other set has one or more items that are already existing, they will not be included.

### Syntax

```
set.update(obj)
```

### Parameters

- **obj** – a set or a sequence object (list, tuple, string)

### Example

The following example shows how the update() method works –

```
</> Open Compiler  
  
lang1 = {"C", "C++", "Java", "Python"}  
lang2 = {"PHP", "C#", "Perl"}  
lang1.update(lang2)  
print (lang1)
```

It will produce the following **output** –

```
{'Python', 'Java', 'C', 'C#', 'PHP', 'Perl', 'C++'}
```

### Example

The update() method also accepts any sequence object as argument. Here, a tuple is the argument for update() method.

```
</>
```

Open Compiler

```
lang1 = {"C", "C++", "Java", "Python"}  
lang2 = ("PHP", "C#", "Perl")  
lang1.update(lang2)  
print (lang1)
```

It will produce the following **output** –

```
{'Java', 'Perl', 'Python', 'C++', 'C#', 'C', 'PHP'}
```

## Example

In this example, a set is constructed from a string, and another string is used as argument for update() method.

</>

Open Compiler

```
set1 = set("Hello")  
set1.update("World")  
print (set1)
```

It will produce the following **output** –

```
{'H', 'r', 'o', 'd', 'W', 'l', 'e'}
```

## union() Method

The union() method of set class also combines the unique items from two sets, but it returns a new set object.

## Syntax

```
set.union(obj)
```

## Parameters

- **obj** – a set or a sequence object (list, tuple, string)

## Return value

The `union()` method returns a set object

## Example

The following example shows how the `union()` method works –

```
</> Open Compiler  
lang1 = {"C", "C++", "Java", "Python"}  
lang2 = {"PHP", "C#", "Perl"}  
lang3 = lang1.union(lang2)  
print (lang3)
```

It will produce the following **output** –

```
{'C#', 'Java', 'Perl', 'C++', 'PHP', 'Python', 'C'}
```

## Example

If a sequence object is given as argument to `union()` method, Python automatically converts it to a set first and then performs union.

```
</> Open Compiler  
lang1 = {"C", "C++", "Java", "Python"}  
lang2 = ["PHP", "C#", "Perl"]  
lang3 = lang1.union(lang2)  
print (lang3)
```

It will produce the following **output** –

```
{'PHP', 'C#', 'Python', 'C', 'Java', 'C++', 'Perl'}
```

## Example

In this example, a set is constructed from a string, and another string is used as argument for `union()` method.

```
</>
```

Open Compiler

```
set1 = set("Hello")
set2 = set1.union("World")
print (set2)
```

It will produce the following **output** –

```
{'e', 'H', 'r', 'd', 'W', 'o', 'l'}
```

## Python - Remove Set Items

Python's set class provides different methods to remove one or more items from a set object.

### remove() Method

The remove() method removes the given item from the set collection, if it is present in it. However, if it is not present, it raises KeyError.

### Syntax

```
set.remove(obj)
```

### Parameters

- **obj** – an immutable object

### Example

```
</>
```

Open Compiler

```
lang1 = {"C", "C++", "Java", "Python"}
print ("Set before removing: ", lang1)
lang1.remove("Java")
print ("Set after removing: ", lang1)
lang1.remove("PHP")
```

It will produce the following **output** –

```
Set before removing: {'C', 'C++', 'Python', 'Java'}
Set after removing: {'C', 'C++', 'Python'}
lang1.remove("PHP")
KeyError: 'PHP'
```

## discard() Method

The `discard()` method in `set` class is similar to `remove()` method. The only difference is, it doesn't raise error even if the object to be removed is not already present in the set collection.

## Syntax

```
set.discard(obj)
```

## Parameters

- **obj** – An immutable object

## Example

```
</> Open Compiler
lang1 = {"C", "C++", "Java", "Python"}
print ("Set before discarding C++: ", lang1)
lang1.discard("C++")
print ("Set after discarding C++: ", lang1)
print ("Set before discarding PHP: ", lang1)
lang1.discard("PHP")
print ("Set after discarding PHP: ", lang1)
```

It will produce the following **output** –

```
Set before discarding C++: {'Java', 'C++', 'Python', 'C'}
Set after discarding C++: {'Java', 'Python', 'C'}
Set before discarding PHP: {'Java', 'Python', 'C'}
Set after discarding PHP: {'Java', 'Python', 'C'}
```

## pop() Method

The pop() method in set class removes an arbitrary item from the set collection. The removed item is returned by the method. Popping from an empty set results in KeyError.

### Syntax

```
obj = set.pop()
```

### Return value

The pop() method returns the object removed from set.

### Example

```
</>
```

Open Compiler

```
lang1 = {"C", "C++"}  
print ("Set before popping: ", lang1)  
obj = lang1.pop()  
print ("object popped: ", obj)  
print ("Set after popping: ", lang1)  
obj = lang1.pop()  
obj = lang1.pop()
```

It will produce the following **output** –

```
Set before popping: {'C++', 'C'}  
object popped: C++  
Set after popping: {'C'}  
Traceback (most recent call last):  
  obj = lang1.pop()  
  ^^^^^^^^^^^^^^  
KeyError: 'pop from an empty set'
```

At the time of call to pop() for third time, the set is empty, hence KeyError is raised.

## clear() Method

The clear() method in set class removes all the items in a set object, leaving an empty set.

## Syntax

```
set.clear()
```

## Example

```
</>
```

Open Compiler

```
lang1 = {"C", "C++", "Java", "Python"}  
print (lang1)  
print ("After clear() method")  
lang1.clear()  
print (lang1)
```

It will produce the following **output** –

```
{'Java', 'C++', 'Python', 'C'}  
After clear() method  
set()
```

## difference\_update() Method

The difference\_update() method in set class updates the set by removing items that are common between itself and another set given as argument.

## Syntax

```
set.difference_update(obj)
```

## Parameters

- **obj** – a set object

## Example

```
</>
```

Open Compiler

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1 before running difference_update: ", s1)
s1.difference_update(s2)
print ("s1 after running difference_update: ", s1)
```

It will produce the following **output** –

```
s1 before running difference_update: {1, 2, 3, 4, 5}
s1 after running difference_update: {1, 2, 3}
set()
```

## difference() Method

The difference() method is similar to difference\_update() method, except that it returns a new set object that contains the difference of the two existing sets.

### Syntax

```
set.difference(obj)
```

### Parameters

- **obj** – a set object

### Return value

The difference() method returns a new set with items remaining after removing those in obj.

### Example

```
</>
Open Compiler

s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1: ", s1, "s2: ", s2)
s3 = s1.difference(s2)
print ("s3 = s1-s2: ", s3)
```

It will produce the following **output** –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}  
s3 = s1-s2: {1, 2, 3}
```

## intersection\_update() Method

As a result of intersection\_update() method, the set object retains only those items which are common in itself and other set object given as argument.

### Syntax

```
set.intersection_update(obj)
```

### Parameters

- **obj** – a set object

### Return value

The intersection\_update() method removes uncommon items and keeps only those items which are common to itself and obj.

### Example

```
</>  
  
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
print ("s1: ", s1, "s2: ", s2)  
s1.intersection_update(s2)  
print ("a1 after intersection: ", s1)
```

Open Compiler

It will produce the following **output** –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}  
s1 after intersection: {4, 5}
```

## intersection() Method

The intersection() method in set class is similar to its intersection\_update() method, except that it returns a new set object that consists of items common to existing sets.

### Syntax

```
set.intersection(obj)
```

### Parameters

- **obj** – a set object

### Return value

The intersection() method returns a set object, retaining only those items common in itself and obj.

### Example

```
</>  
Open Compiler  
  
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
print ("s1: ", s1, "s2: ", s2)  
s3 = s1.intersection(s2)  
print ("s3 = s1 & s2: ", s3)
```

It will produce the following **output** –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}  
s3 = s1 & s2: {4, 5}
```

## symmetric\_difference\_update() method

The symmetric difference between two sets is the collection of all the uncommon items, rejecting the common elements. The symmetric\_difference\_update() method updates a set with symmetric difference between itself and the set given as argument.

## Syntax

```
set.symmetric_difference_update(obj)
```

## Parameters

- **obj** – a set object

## Example

```
</> Open Compiler  
  
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
print ("s1: ", s1, "s2: ", s2)  
s1.symmetric_difference_update(s2)  
print ("s1 after running symmetric difference ", s1)
```

It will produce the following **output** –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}  
s1 after running symmetric difference {1, 2, 3, 6, 7, 8}
```

## symmetric\_difference() Method

The `symmetric_difference()` method in set class is similar to `symmetric_difference_update()` method, except that it returns a new set object that holds all the items from two sets minus the common items.

## Syntax

```
set.symmetric_difference(obj)
```

## Parameters

- **obj** – a set object

## Return value

The `symmetric_difference()` method returns a new set that contains only those items not common between the two set objects.

## Example

&lt;/&gt;

Open Compiler

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1: ", s1, "s2: ", s2)
s3 = s1.symmetric_difference(s2)
print ("s1 = s1^s2 ", s3)
```

It will produce the following **output** –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}
s1 = s1^s2 {1, 2, 3, 6, 7, 8}
```

## Python - Loop Sets

A set in Python is not a sequence, nor is it a mapping type class. Hence, the objects in a set cannot be traversed with index or key. However, you can traverse each item in a set using a **for** loop.

### Example 1

The following example shows how you can traverse through a set using a **for** loop –

&lt;/&gt;

Open Compiler

```
langs = ["C", "C++", "Java", "Python"]
for lang in langs:
    print (lang)
```

It will produce the following **output** –

```
C
Python
C++
Java
```

## Example 2

The following example shows how you can run a **for** loop over the elements of one set, and use the **add()** method of set class to add in another set.

&lt;/&gt;

Open Compiler

```
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
for x in s2:  
    s1.add(x)  
print (s1)
```

It will produce the following **output** –

{1, 2, 3, 4, 5, 6, 7, 8}

## Python - Join Sets

In Python, a Set is an ordered collection of items. The items may be of different types. However, an item in the set must be an immutable object. It means, we can only include numbers, string and tuples in a set and not lists. Python's set class has different provisions to join set objects.

### Using the "|" Operator

The "|" symbol (pipe) is defined as the union operator. It performs the AuB operation and returns a set of items in A, B or both. Set doesn't allow duplicate items.

&lt;/&gt;

Open Compiler

```
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s3 = s1|s2  
print (s3)
```

It will produce the following **output** –

{1, 2, 3, 4, 5, 6, 7, 8}

## Using the union() Method

The set class has union() method that performs the same operation as | operator. It returns a set object that holds all items in both sets, discarding duplicates.

```
</>  
Open Compiler  
  
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s3 = s1.union(s2)  
print (s3)
```

## Using the update() Method

The update() method also joins the two sets, as the union() method. However it doesn't return a new set object. Instead, the elements of second set are added in first, duplicates not allowed.

```
</>  
Open Compiler  
  
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s1.update(s2)  
print (s1)
```

## Using the unpacking Operator

In Python, the "\*" symbol is used as unpacking operator. The unpacking operator internally assign each element in a collection to a separate variable.

```
</>  
Open Compiler  
  
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s3 = {*s1, *s2}  
print (s3)
```

# Python - Copy Sets

The `copy()` method in set class creates a shallow copy of a set object.

## Syntax

```
set.copy()
```

## Return Value

The `copy()` method returns a new set which is a shallow copy of existing set.

## Example

```
</>
```

Open Compiler

```
lang1 = {"C", "C++", "Java", "Python"}  
print ("lang1: ", lang1, "id(lang1): ", id(lang1))  
lang2 = lang1.copy()  
print ("lang2: ", lang2, "id(lang2): ", id(lang2))  
lang1.add("PHP")  
print ("After updating lang1")  
print ("lang1: ", lang1, "id(lang1): ", id(lang1))  
print ("lang2: ", lang2, "id(lang2): ", id(lang2))
```

## Output

```
lang1: {'Python', 'Java', 'C', 'C++'} id(lang1): 2451578196864  
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312  
After updating lang1  
lang1: {'Python', 'C', 'C++', 'PHP', 'Java'} id(lang1): 2451578196864  
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312
```

# Python - Set Operators

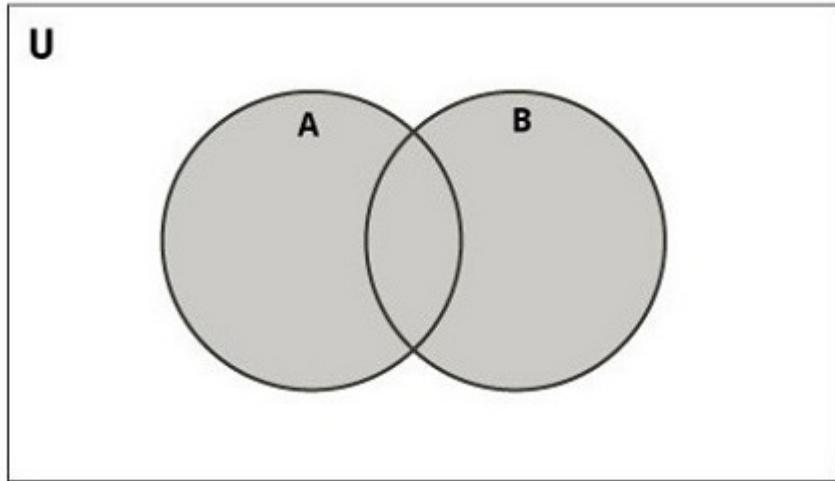
In the Set Theory of Mathematics, the union, intersection, difference and symmetric difference operations are defined. Python implements them with following operators –

## Union Operator (|)

The union of two sets is a set containing all elements that are in A or in B or both. For example,

$$\{1,2\} \cup \{2,3\} = \{1,2,3\}$$

The following diagram illustrates the union of two sets.



Python uses the "|" symbol as a union operator. The following example uses the "|" operator and returns the union of two sets.

## Example

```
</> Open Compiler  
  
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
s3 = s1 | s2  
print ("Union of s1 and s2: ", s3)
```

It will produce the following **output** –

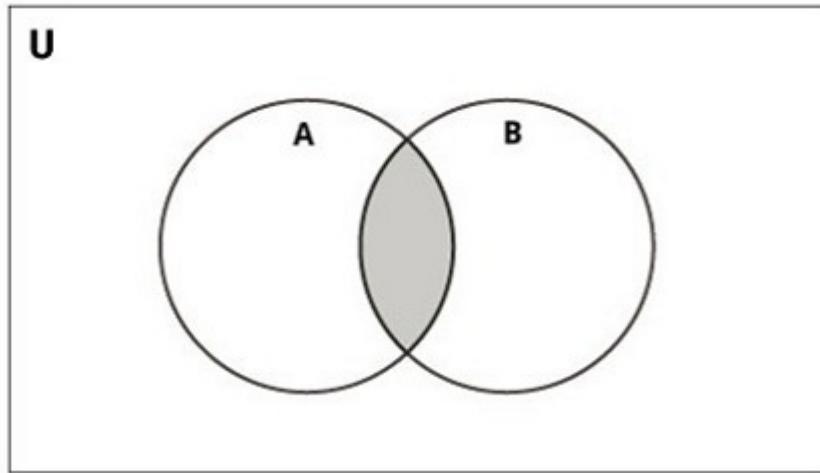
Union of s1 and s2: {1, 2, 3, 4, 5, 6, 7, 8}

## Intersection Operator (&)

The intersection of two sets AA and BB, denoted by  $A \cap B$ , consists of all elements that are both in A and B. For example,

$$\{1,2\} \cap \{2,3\} = \{2\}$$

The following diagram illustrates intersection of two sets.



Python uses the "&" symbol as an intersection operator. Following example uses & operator and returns intersection of two sets.

&lt;/&gt;

Open Compiler

```
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
s3 = s1 & s2  
print ("Intersection of s1 and s2: ", s3)
```

It will produce the following **output** –

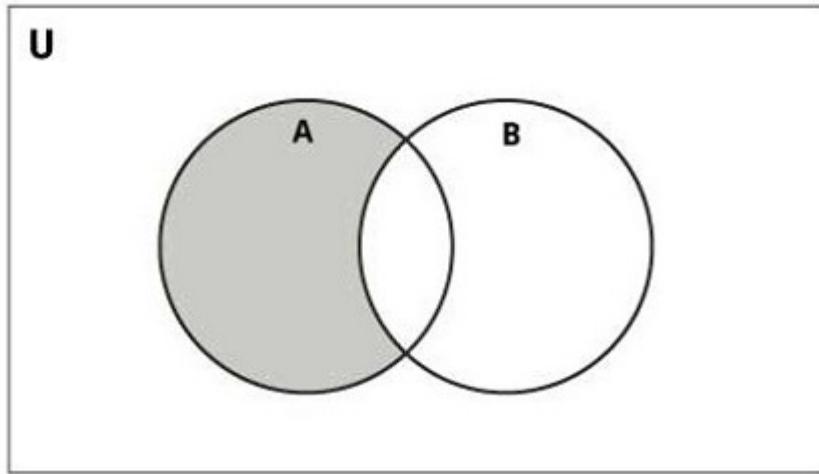
Intersection of s1 and s2: {4, 5}

## Difference Operator (-)

The difference (subtraction) is defined as follows. The set  $A-B$  consists of elements that are in A but not in B. For example,

If  $A=\{1,2,3\}$  and  $B=\{3,5\}$ , then  $A-B=\{1,2\}$

The following diagram illustrates difference of two sets –



Python uses the `"-"` symbol as a difference operator.

## Example

The following example uses the `"-"` operator and returns difference of two sets.

&lt;/&gt;

Open Compiler

```
s1 = {1,2,3,4,5}  
s2 = {4,5,6,7,8}  
s3 = s1 - s2  
print ("Difference of s1 - s2: ", s3)  
s3 = s2 - s1  
print ("Difference of s2 - s1: ", s3)
```

It will produce the following **output** –

Difference of s1 - s2: {1, 2, 3}

Difference of s2 - s1: {8, 6, 7}

Note that " $s1-s2$ " is not the same as " $s2-s1$ ".

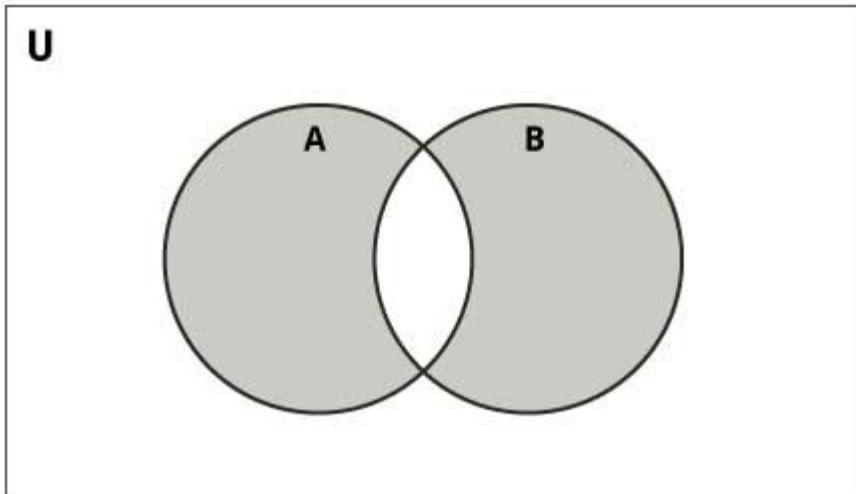
## Symmetric Difference Operator

The symmetric difference of  $A$  and  $B$  is denoted by " $A \Delta B$ " and is defined by

$$A \Delta B = (A - B) \cup (B - A)$$

If  $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$  and  $B = \{1, 3, 5, 6, 7, 8, 9\}$ , then  $A \Delta B = \{2, 4, 9\}$ .

The following diagram illustrates the symmetric difference between two sets –



Python uses the " $\wedge$ " symbol as a symbolic difference operator.

## Example

The following example uses the " $\wedge$ " operator and returns symbolic difference of two sets.

```
</> Open Compiler
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s3 = s1 - s2
print ("Difference of s1 - s2: ", s3)
s3 = s2 - s1
print ("Difference of s2 - s1: ", s3)
s3 = s1 ^ s2
print ("Symmetric Difference in s1 and s2: ", s3)
```

It will produce the following **output** –

```
Difference of s1 - s2: {1, 2, 3}
Difference of s2 - s1: {8, 6, 7}
Symmetric Difference in s1 and s2: {1, 2, 3, 6, 7, 8}
```

## Python - Set Methods

Following methods are defined in Python's set class –

| Sr.No. | Methods & Description |
|--------|-----------------------|
|--------|-----------------------|

|    |                                                                                                           |
|----|-----------------------------------------------------------------------------------------------------------|
| 1  | <b>add()</b><br>Add an element to a set.                                                                  |
| 2  | <b>clear()</b><br>Remove all elements from this set.                                                      |
| 3  | <b>copy()</b><br>Return a shallow copy of a set.                                                          |
| 4  | <b>difference()</b><br>Return the difference of two or more sets as a new set.                            |
| 5  | <b>difference_update()</b><br>Remove all elements of another set from this set.                           |
| 6  | <b>discard()</b><br>Remove an element from a set if it is a member.                                       |
| 7  | <b>intersection()</b><br>Return the intersection of two sets as a new set.                                |
| 8  | <b>intersection_update()</b><br>Update a set with the intersection of itself and another.                 |
| 9  | <b>isdisjoint()</b><br>Return True if two sets have a null intersection.                                  |
| 10 | <b>issubset()</b><br>Return True if another set contains this set.                                        |
| 11 | <b>issuperset()</b><br>Return True this set contains another set.                                         |
| 12 | <b>pop()</b><br>Remove and return an arbitrary set element                                                |
| 13 | <b>remove()</b><br>Remove an element from a set; it must be a member.                                     |
| 14 | <b>symmetric_difference()</b><br>Return the symmetric difference of two sets as a new set.                |
| 15 | <b>symmetric_difference_update()</b><br>Update a set with the symmetric difference of itself and another. |
| 16 | <b>union()</b><br>Return the union of sets as a new set.                                                  |

17

**update()**

Update a set with the union of itself and others.

## Python - Set Exercises

### Example 1

Python program to find common elements in two lists with the help of set operations –

```
</> Open Compiler  
l1=[1,2,3,4,5]  
l2=[4,5,6,7,8]  
s1=set(l1)  
s2=set(l2)  
commons = s1&s2 # or s1.intersection(s2)  
commonlist = list(commons)  
print (commonlist)
```

It will produce the following **output** –

```
[4, 5]
```

### Example 2

Python program to check if a set is a subset of another –

```
</> Open Compiler  
s1={1,2,3,4,5}  
s2={4,5}  
if s2.issubset(s1):  
    print ("s2 is a subset of s1")  
else:  
    print ("s2 is not a subset of s1")
```

It will produce the following **output** –

```
s2 is a subset of s1
```

## Example 3

Python program to obtain a list of unique elements in a list –

&lt;/&gt;

Open Compiler

```
T1 = (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
s1 = set(T1)
print (s1)
```

It will produce the following **output** –

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

## Exercise Programs

- Python program to find the size of a set object.
- Python program that splits a set into two based on odd/even numbers.
- Python program to remove all negative numbers from a set.
- Python program to build another set with absolute value of each number in a set.
- Python program to remove all strings from a set which has elements of different types.

# Python - Dictionaries

Dictionary is one of the built-in data types in Python. Python's dictionary is example of mapping type. A mapping object 'maps' value of one object with another.

In a language dictionary we have pairs of word and corresponding meaning. Two parts of pair are key (word) and value (meaning). Similarly, Python dictionary is also a collection of key:value pairs. The pairs are separated by comma and put inside curly brackets {}.

To establish mapping between key and value, the colon ':' symbol is put between the two.

Given below are some examples of Python dictionary objects –

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}
numbers = {10: "Ten", 20: "Twenty", 30: "Thirty", 40: "Forty"}
marks = {"Savita": 67, "Imtiaz": 88, "Laxman": 91, "David": 49}
```

## Example 1

Only a number, string or tuple can be used as key. All of them are immutable. You can use an object of any type as the value. Hence following definitions of dictionary are also valid –

&lt;/&gt;

Open Compiler

```
d1 = {"Fruit": ["Mango", "Banana"], "Flower": ["Rose", "Lotus"]}
d2 = {('India, USA'): 'Countries', ('New Delhi', 'New York'): 'Capitals'}
print (d1)
print (d2)
```

It will produce the following **output** –

```
{'Fruit': ['Mango', 'Banana'], 'Flower': ['Rose', 'Lotus']}
{'India, USA': 'Countries', ('New Delhi', 'New York'): 'Capitals'}
```

## Example 2

Python doesn't accept mutable objects such as list as key, and raises `TypeError`.

&lt;/&gt;

Open Compiler

```
d1 = {[ "Mango", "Banana"]:"Fruit", "Flower": [ "Rose", "Lotus"]}
print (d1)
```

It will raise a `TypeError` –

Traceback (most recent call last):

```
  File "C:\Users\Sairam\PycharmProjects\pythonProject\main.py", line 8, in <module>
    d1 = {[ "Mango", "Banana"]:"Fruit", "Flower": [ "Rose", "Lotus"]}
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: unhashable type: 'list'
```

## Example 3

You can assign a value to more than one keys in a dictionary, but a key cannot appear more than once in a dictionary.

&lt;/&gt;

[Open Compiler](#)

```
d1 = {"Banana": "Fruit", "Rose": "Flower", "Lotus": "Flower", "Mango": "Fruit"}
d2 = {"Fruit": "Banana", "Flower": "Rose", "Fruit": "Mango", "Flower": "Lotus"}
print (d1)
print (d2)
```

It will produce the following **output** –

```
{'Banana': 'Fruit', 'Rose': 'Flower', 'Lotus': 'Flower', 'Mango': 'Fruit'}
{'Fruit': 'Mango', 'Flower': 'Lotus'}
```

## Python Dictionary Operators

In Python, following operators are defined to be used with dictionary operands. In the example, the following dictionary objects are used.

```
d1 = {'a': 2, 'b': 4, 'c': 30}
d2 = {'a1': 20, 'b1': 40, 'c1': 60}
```

| Operator     | Description                                          | Example                                                                          |
|--------------|------------------------------------------------------|----------------------------------------------------------------------------------|
| dict[key]    | Extract/assign the value mapped with key             | print (d1['b']) retrieves 4<br>d1['b'] = 'Z' assigns new value to key 'b'        |
| dict1 dict2  | Union of two dictionary objects, retuning new object | d3=d1 d2 ; print (d3)<br>{'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60} |
| dict1 =dict2 | Augmented dictionary union operator                  | d1 =d2; print (d1)<br>{'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60}    |

## Python - Access Dictionary Items

### Using the "[ ]" Operator

A dictionary in Python is not a sequence, as the elements in dictionary are not indexed. Still, you can use the square brackets "[ ]" operator to fetch the value associated with a certain key in the dictionary object.

## Example 1

&lt;/&gt;

[Open Compiler](#)

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}  
print ("Capital of Gujarat is : ", capitals['Gujarat'])  
print ("Capital of Karnataka is : ", capitals['Karnataka'])
```

It will produce the following **output** –

```
Capital of Gujarat is: Gandhinagar  
Capital of Karnataka is: Bengaluru
```

## Example 2

Python raises a `KeyError` if the key given inside the square brackets is not present in the dictionary object.

&lt;/&gt;

[Open Compiler](#)

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}  
print ("Capatial of Haryana is : ", capitals['Haryana'])
```

It will produce the following **output** –

```
print ("Capatial of Haryana is : ", capitals['Haryana'])  
~~~~~^~~~~~  
KeyError: 'Haryana'
```

## Using the `get()` Method

The `get()` method in Python's `dict` class returns the value mapped to the given key.

### Syntax

```
Val = dict.get("key")
```

## Parameters

- **key** – An immutable object used as key in the dictionary object

## Return Value

The get() method returns the object mapped with the given key.

### Example 3

&lt;/&gt;

Open Compiler

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}  
print ("Capital of Gujarat is: ", capitals.get('Gujarat'))  
print ("Capital of Karnataka is: ", capitals.get('Karnataka'))
```

It will produce the following **output** –

```
Capital of Gujarat is: Gandhinagar  
Capital of Karnataka is: Bengaluru
```

### Example 4

Unlike the "[]" operator, the get() method doesn't raise error if the key is not found; it return None.

&lt;/&gt;

Open Compiler

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}  
print ("Capital of Haryana is : ", capitals.get('Haryana'))
```

It will produce the following **output** –

```
Capital of Haryana is : None
```

### Example 5

The `get()` method accepts an optional string argument. If it is given, and if the key is not found, this string becomes the return value.

&lt;/&gt;

Open Compiler

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar", "Telangana": "Hyderabad"}  
print ("Capital of Haryana is : ", capitals.get('Haryana', 'Not found'))
```

It will produce the following **output** –

```
Capital of Haryana is: Not found
```

## Python - Change Dictionary Items

Apart from the literal representation of dictionary, where we put comma-separated key:value pairs in curly brackets, we can create dictionary object with built-in `dict()` function.

### Empty Dictionary

Using `dict()` function without any arguments creates an empty dictionary object. It is equivalent to putting nothing between curly brackets.

#### Example

&lt;/&gt;

Open Compiler

```
d1 = dict()  
d2 = {}  
print ('d1: ', d1)  
print ('d2: ', d2)
```

It will produce the following **output** –

```
d1: {}  
d2: {}
```

### Dictionary from List of Tuples

The `dict()` function constructs a dictionary from a list or tuple of two-item tuples. First item in a tuple is treated as key, and the second as its value.

## Example

&lt;/&gt;

[Open Compiler](#)

```
d1=dict([('a', 100), ('b', 200)])
d2 = dict([('a', 'one'), ('b', 'two')])
print ('d1: ', d1)
print ('d2: ', d2)
```

It will produce the following **output** –

```
d1: {'a': 100, 'b': 200}
d2: {'a': 'one', 'b': 'two'}
```

## Dictionary from Keyword Arguments

The `dict()` function can take any number of keyword arguments with name=value pairs. It returns a dictionary object with the name as key and associates it to the value.

## Example

&lt;/&gt;

[Open Compiler](#)

```
d1=dict(a= 100, b=200)
d2 = dict(a='one', b='two')
print ('d1: ', d1)
print ('d2: ', d2)
```

It will produce the following **output** –

```
d1: {'a': 100, 'b': 200}
d2: {'a': 'one', 'b': 'two'}
```

## Python - Add Dictionary Items

## Using the Operator

The "[]" operator (used to access value mapped to a dictionary key) is used to update an existing key-value pair as well as add a new pair.

### Syntax

```
dict["key"] = val
```

If the key is already present in the dictionary object, its value will be updated to val. If the key is not present in the dictionary, a new key-value pair will be added.

### Example

In this example, the marks of "Laxman" are updated to 95.

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: ", marks)
marks['Laxman'] = 95
print ("marks dictionary after update: ", marks)
```

It will produce the following **output** –

```
marks dictionary before update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 95, 'David': 49}
```

### Example

However, an item with 'Krishnan' as its key is not available in the dictionary, hence a new key-value pair is added.

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: ", marks)
marks['Krishan'] = 74
print ("marks dictionary after update: ", marks)
```

It will produce the following **output** –

```
marks dictionary before update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49, 'Kris': 65}
```

## Using the update() Method

You can use the update() method in dict class in three different ways:

### Update with Another Dictionary

In this case, the update() method's argument is another dictionary. Value of keys common in both dictionaries is updated. For new keys, key-value pair is added in the existing dictionary

### Syntax

```
d1.update(d2)
```

### Return value

The existing dictionary is updated with new key-value pairs added to it.

### Example

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
marks.update(marks1)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Update with Iterable

If the argument to update() method is a list or tuple of two item tuples, an item each for it is added in the existing dictionary, or updated if the key is existing.

### Syntax

```
d1.update([(k1, v1), (k2, v2)])
```

### Return value

Existing dictionary is updated with new keys added.

### Example

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = [("Sharad", 51), ("Mushtaq", 61), ("Laxman", 89)]
marks.update(marks1)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Update with Keyword Arguments

Third version of update() method accepts list of keyword arguments in name=value format. New k-v pairs are added, or value of existing key is updated.

### Syntax

```
d1.update(k1=v1, k2=v2)
```

### Return value

Existing dictionary is updated with new key-value pairs added.

## Example

&lt;/&gt;

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks.update(Sharad = 51, Mushtaq = 61, Laxman = 89)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Using the Unpack Operator

The "\*\*\*" symbol prefixed to a dictionary object unpacks it to a list of tuples, each tuple with key and value. Two **dict** objects are unpacked and merged together and obtain a new dictionary.

## Syntax

```
d3 = {**d1, **d2}
```

## Return value

Two dictionaries are merged and a new object is returned.

## Example

&lt;/&gt;

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
```

```
newmarks = {**marks, **marks1}
print ("marks dictionary after update: \n", newmarks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Using the Union Operator (|)

Python introduces the "|" (pipe symbol) as the union operator for dictionary operands. It updates existing keys in dict object on left, and adds new key-value pairs to return a new dict object.

### Syntax

```
d3 = d1 | d2
```

### Return value

The Union operator return a new dict object after merging the two dict operands

### Example

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
newmarks = marks | marks1
print ("marks dictionary after update: \n", newmarks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
```

marks dictionary after update:

```
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Using "|=" Operator

The "|=" operator is an augmented Union operator. It performs in-place update on the dictionary operand on left by adding new keys in the operand on right, and updating the existing keys.

### Syntax

```
d1 |= d2
```

### Example

```
</>
```

Open Compiler

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
marks |= marks1
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

## Python - Remove Dictionary Items

### Using del Keyword

Python's **del** keyword deletes any object from the memory. Here we use it to delete a key-value pair in a dictionary.

### Syntax

```
del dict['key']
```

## Example

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
print ("numbers dictionary before delete operation: \n", numbers)  
del numbers[20]  
print ("numbers dictionary before delete operation: \n", numbers)
```

It will produce the following **output** –

```
numbers dictionary before delete operation:  
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}  
numbers dictionary before delete operation:  
{10: 'Ten', 30: 'Thirty', 40: 'Forty'}
```

## Example

The del keyword with the dict object itself removes it from memory.

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
print ("numbers dictionary before delete operation: \n", numbers)  
del numbers  
print ("numbers dictionary before delete operation: \n", numbers)
```

It will produce the following **output** –

```
numbers dictionary before delete operation:  
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}  
Traceback (most recent call last):  
File "C:\Users\mlath\examples\main.py", line 5, in <module>  
    print ("numbers dictionary before delete operation: \n", numbers)  
                                ^^^^^^  
NameError: name 'numbers' is not defined
```

## Using pop() Method

The pop() method of dict class causes an element with the specified key to be removed from the dictionary.

### Syntax

```
val = dict.pop(key)
```

### Return value

The pop() method returns the value of the specified key after removing the key-value pair.

### Example

```
</> Open Compiler
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
print ("numbers dictionary before pop operation: \n", numbers)
val = numbers.pop(20)
print ("numbers dictionary after pop operation: \n", numbers)
print ("Value popped: ", val)
```

It will produce the following **output** –

```
numbers dictionary before pop operation:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after pop operation:
{10: 'Ten', 30: 'Thirty', 40: 'Forty'}
Value popped: Twenty
```

## Using popitem() Method

The popitem() method in dict() class doesn't take any argument. It pops out the last inserted key-value pair, and returns the same as a tuple

### Syntax

```
val = dict.popitem()
```

## Return Value

The `popitem()` method return a tuple contain key and value of the removed item from the dictionary

### Example

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
print ("numbers dictionary before pop operation: \n", numbers)  
val = numbers.popitem()  
print ("numbers dictionary after pop operation: \n", numbers)  
print ("Value popped: ", val)
```

It will produce the following **output** –

```
numbers dictionary before pop operation:  
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}  
numbers dictionary after pop operation:  
{10: 'Ten', 20: 'Twenty', 30: 'Thirty'}  
Value popped: (40, 'Forty')
```

## Using clear() Method

The `clear()` method in `dict` class removes all the elements from the dictionary object and returns an empty object.

### Syntax

```
dict.clear()
```

### Example

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
print ("numbers dictionary before clear method: \n", numbers)
```

```
numbers.clear()
print ("numbers dictionary after clear method: \n", numbers)
```

It will produce the following **output** –

```
numbers dictionary before clear method:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after clear method:
{}
```

## Python - Dictionary View Objects

The items(), keys() and values() methods of dict class return view objects. These views are refreshed dynamically whenever any change occurs in the contents of their source dictionary object.

### items() Method

The items() method returns a dict\_items view object. It contains a list of tuples, each tuple made up of respective key, value pairs.

### Syntax

```
Obj = dict.items()
```

### Return value

The items() method returns dict\_items object which is a dynamic view of (key,value) tuples.

### Example

In the following example, we first obtain the dict\_items() object with items() method and check how it is dynamically updated when the dictionary object is updated.

</>
Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
obj = numbers.items()
print ('type of obj: ', type(obj))
print (obj)
```

```
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** –

```
type of obj: <class 'dict_items'>
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty')])
update numbers dictionary
View automatically updated
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty'), (50, 'Fifty')])
```

## keys() Method

The `keys()` method of `dict` class returns `dict_keys` object which is a list of all keys defined in the dictionary. It is a view object, as it gets automatically updated whenever any update action is done on the dictionary object.

## Syntax

```
Obj = dict.keys()
```

## Return value

The `keys()` method returns `dict_keys` object which is a view of keys in the dictionary.

## Example

</>

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
obj = numbers.keys()
print ('type of obj: ', type(obj))
print (obj)
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** –

```
type of obj: <class 'dict_keys'>
dict_keys([10, 20, 30, 40])
update numbers dictionary
View automatically updated
dict_keys([10, 20, 30, 40, 50])
```

## values() Method

The values() method returns a view of all the values present in the dictionary. The object is of dict\_values type, which gets automatically updated.

### Syntax

```
Obj = dict.values()
```

### Return value

The values() method returns a dict\_values view of all the values present in the dictionary.

### Example

```
</>
```

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
obj = numbers.values()
print ('type of obj: ', type(obj))
print (obj)
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** –

```
type of obj: <class 'dict_values'>
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty'])
update numbers dictionary
```

```
View automatically updated  
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty', 'Fifty'])
```

## Python - Loop Dictionaries

Unlike a list, tuple or a string, dictionary data type in Python is not a sequence, as the items do not have a positional index. However, traversing a dictionary is still possible with different techniques.

### Example 1

Running a simple **for** loop over the dictionary object traverses the keys used in it.

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers:  
    print (x)
```

It will produce the following **output** –

```
10  
20  
30  
40
```

### Example 2

Once we are able to get the key, its associated value can be easily accessed either by using square brackets operator or with get() method.

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers:  
    print (x, ":", numbers[x])
```

It will produce the following **output** –

```
10 : Ten  
20 : Twenty  
30 : Thirty  
40 : Forty
```

The `items()`, `keys()` and `values()` methods of `dict` class return the view objects `dict_items`, `dict_keys` and `dict_values` respectively. These objects are iterators, and hence we can run a for loop over them.

### Example 3

The `dict_items` object is a list of key-value tuples over which a for loop can be run as follows:

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers.items():  
    print (x)
```

It will produce the following **output** –

```
(10, 'Ten')  
(20, 'Twenty')  
(30, 'Thirty')  
(40, 'Forty')
```

Here, "x" is the tuple element from the `dict_items` iterator. We can further unpack this tuple in two different variables.

### Example 4

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x,y in numbers.items():  
    print (x,":", y)
```

It will produce the following **output** –

```
10 : Ten  
20 : Twenty  
30 : Thirty  
40 : Forty
```

## Example 5

Similarly, the collection of keys in dict\_keys object can be iterated over.

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
for x in numbers.keys():  
    print (x, ":", numbers[x])
```

Respective Keys and values in dict\_keys and dict\_values are at same index. In the following example, we have a for loop that runs from 0 to the length of the dict, and use the looping variable as index and print key and its corresponding value.

## Example 6

&lt;/&gt;

Open Compiler

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
l = len(numbers)  
for x in range(l):  
    print (list(numbers.keys())[x], ":", list(numbers.values())[x])
```

The above two code snippets produce identical **output** –

```
10 : Ten  
20 : Twenty  
30 : Thirty  
40 : Forty
```

# Python - Copy Dictionaries

Since a variable in Python is merely a label or reference to an object in the memory, a simple assignment operator will not create copy of object.

## Example 1

In this example, we have a dictionary "d1" and we assign it to another variable "d2". If "d1" is updated, the changes also reflect in "d2".

```
</> Open Compiler  
  
d1 = {"a":11, "b":22, "c":33}  
d2 = d1  
print ("id:", id(d1), "dict: ",d1)  
print ("id:", id(d2), "dict: ",d2)  
  
d1["b"] = 100  
print ("id:", id(d1), "dict: ",d1)  
print ("id:", id(d2), "dict: ",d2)
```

## Output

```
id: 2215278891200 dict: {'a': 11, 'b': 22, 'c': 33}  
id: 2215278891200 dict: {'a': 11, 'b': 22, 'c': 33}  
id: 2215278891200 dict: {'a': 11, 'b': 100, 'c': 33}  
id: 2215278891200 dict: {'a': 11, 'b': 100, 'c': 33}
```

To avoid this, and make a shallow copy of a dictionary, use the `copy()` method instead of assignment.

## Example 2

```
</> Open Compiler  
  
d1 = {"a":11, "b":22, "c":33}  
d2 = d1.copy()  
print ("id:", id(d1), "dict: ",d1)  
print ("id:", id(d2), "dict: ",d2)  
d1["b"] = 100  
print ("id:", id(d1), "dict: ",d1)  
print ("id:", id(d2), "dict: ",d2)
```

## Output

When "d1" is updated, "d2" will not change now because "d2" is the copy of dictionary object, not merely a reference.

```
id: 1586671734976 dict: {'a': 11, 'b': 22, 'c': 33}
id: 1586673973632 dict: {'a': 11, 'b': 22, 'c': 33}
id: 1586671734976 dict: {'a': 11, 'b': 100, 'c': 33}
id: 1586673973632 dict: {'a': 11, 'b': 22, 'c': 33}
```

## Python - Nested Dictionaries

A Python dictionary is said to have a nested structure if value of one or more keys is another dictionary. A nested dictionary is usually employed to store a complex data structure.

The following code snippet represents a nested dictionary:

```
marklist = {
    "Mahesh" : {"Phy" : 60, "maths" : 70},
    "Madhavi" : {"phy" : 75, "maths" : 68},
    "Mitchell" : {"phy" : 67, "maths" : 71}
}
```

### Example 1

You can also constitute a for loop to traverse nested dictionary, as in the previous section.

</>

[Open Compiler](#)

```
marklist = {
    "Mahesh" : {"Phy" : 60, "maths" : 70},
    "Madhavi" : {"phy" : 75, "maths" : 68},
    "Mitchell" : {"phy" : 67, "maths" : 71}
}
for k,v in marklist.items():
    print (k, ":", v)
```

It will produce the following **output** –

```
Mahesh : {'Phy': 60, 'maths': 70}
Madhavi : {'phy': 75, 'maths': 68}
```

```
Mitchell : {'phy': 67, 'maths': 71}
```

## Example 2

It is possible to access value from an inner dictionary with [] notation or get() method.

```
print (marklist.get("Madhavi")['maths'])
obj=marklist['Mahesh']
print (obj.get('Phy'))
print (marklist['Mitchell'].get('maths'))
```

It will produce the following **output** –

```
68
60
71
```

# Python - Dictionary Methods

A dictionary in Python is an object of the built-in **dict** class, which defines the following methods –

| Sr.No. | Method and Description                                                                                                |
|--------|-----------------------------------------------------------------------------------------------------------------------|
| 1      | <b>dict.clear()</b><br>Removes all elements of dictionary dict.                                                       |
| 2      | <b>dict.copy()</b><br>Returns a shallow copy of dictionary dict.                                                      |
| 3      | <b>dict.fromkeys()</b><br>Create a new dictionary with keys from seq and values set to value.                         |
| 4      | <b>dict.get(key, default=None)</b><br>For key key, returns value or default if key not in dictionary.                 |
| 5      | <b>dict.has_key(key)</b><br>Returns true if a given key is available in the dictionary, otherwise it returns a false. |
| 6      | <b>dict.items()</b><br>Returns a list of dict's (key, value) tuple pairs.                                             |
| 7      | <b>dict.keys()</b>                                                                                                    |

|    |                                                                                                                              |
|----|------------------------------------------------------------------------------------------------------------------------------|
|    | Returns list of dictionary dict's keys.                                                                                      |
| 8  | <b>dict.pop()</b><br>Removes the element with specified key from the collection                                              |
| 9  | <b>dict.popitem()</b><br>Removes the last inserted key-value pair                                                            |
| 10 | <b>dict.setdefault(key, default=None)</b><br>Similar to get(), but will set dict[key]=default if key is not already in dict. |
| 11 | <b>dict.update(dict2)</b><br>Adds dictionary dict2's key-values pairs to dict.                                               |
| 12 | <b>dict.values()</b><br>Returns list of dictionary dict's values.                                                            |

## Python - Dictionary Exercises

### Example 1

Python program to create a new dictionary by extracting the keys from a given dictionary.

```
</> Open Compiler
d1 = {"one":11, "two":22, "three":33, "four":44, "five":55}
keys = [ 'two', 'five' ]
d2={}
for k in keys:
    d2[k]=d1[k]
print (d2)
```

It will produce the following **output** –

```
{'two': 22, 'five': 55}
```

### Example 2

Python program to convert a dictionary to list of (k,v) tuples.

```
</> Open Compiler
```

```
d1 = {"one":11, "two":22, "three":33, "four":44, "five":55}
L1 = list(d1.items())
print (L1)
```

It will produce the following **output** –

```
[('one', 11), ('two', 22), ('three', 33), ('four', 44), ('five', 55)]
```

### Example 3

Python program to remove keys with same values in a dictionary.

</>

[Open Compiler](#)

```
d1 = {"one":"eleven", "2":2, "three":3, "11":"eleven", "four":44, "two":2}
vals = list(d1.values())#all values
uvals = [v for v in vals if vals.count(v)==1]#unique values
d2 = {}
for k,v in d1.items():
    if v in uvals:
        d = {k:v}
        d2.update(d)
print ("dict with unique value:",d2)
```

It will produce the following **output** –

```
dict with unique value: {'three': 3, 'four': 44}
```

### Exercise Programs

- Python program to sort list of dictionaries by values
- Python program to extract dictionary with each key having non-numeric value from a given dictionary.
- Python program to build a dictionary from list of two item (k,v) tuples.
- Python program to merge two dictionary objects, using unpack operator.

## Python - Arrays

Python's standard data types **list**, **tuple** and **string** are sequences. A sequence object is an ordered collection of items. Each item is characterized by incrementing index starting with zero. Moreover, items in a sequence need not be of same type. In other words, a list or tuple may consist of items of different data type.

This feature is different from the concept of an array in C or C++. In C/C++, an array is also an indexed collection of items, but the items must be of similar data type. In C/C++, you have an array of integers or floats, or strings, but you cannot have an array with some elements of integer type and some of different type. A C/C++ array is therefore a homogenous collection of data types.

Python's standard library has array module. The array class in it allows you to construct an array of three basic types, integer, float and Unicode characters.

## Syntax

The syntax of creating array is –

```
import array  
obj = array.array(typecode[, initializer])
```

## Parameters

- **typecode** – The typecode character used to create the array.
- **initializer** – array initialized from the optional value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

## Return type

The array() constructor returns an object of array.array class

## Example

```
</>
```

Open Compiler

```
import array as arr  
  
# creating an array with integer type  
a = arr.array('i', [1, 2, 3])  
print (type(a), a)  
  
# creating an array with char type
```

```
a = arr.array('u', 'BAT')
print (type(a), a)

# creating an array with float type
a = arr.array('d', [1.1, 2.2, 3.3])
print (type(a), a)
```

It will produce the following **output** –

```
<class 'array.array'> array('i', [1, 2, 3])
<class 'array.array'> array('u', 'BAT')
<class 'array.array'> array('d', [1.1, 2.2, 3.3])
```

Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained.

Python array type is decided by a single character Typecode argument. The type codes and the intended data type of array is listed below –

| <b>typecode</b> | <b>Python data type</b> | <b>Byte size</b> |
|-----------------|-------------------------|------------------|
| 'b'             | signed integer          | 1                |
| 'B'             | unsigned integer        | 1                |
| 'u'             | Unicode character       | 2                |
| 'h'             | signed integer          | 2                |
| 'H'             | unsigned integer        | 2                |
| 'i'             | signed integer          | 2                |
| 'I'             | unsigned integer        | 2                |
| 'l'             | signed integer          | 4                |
| 'L'             | unsigned integer        | 4                |
| 'q'             | signed integer          | 8                |
| 'Q'             | unsigned integer        | 8                |
| 'f'             | floating point          | 4                |
| 'd'             | floating point          | 8                |

# Python - Access Array Items

Since the array object behaves very much like a sequence, you can perform indexing and slicing operation with it.

## Example

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [1, 2, 3])  
#indexing  
print (a[1])  
#slicing  
print (a[1:])
```

## Changing Array Items

You can assign value to an item in the array just as you assign a value to item in a list.

## Example

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [1, 2, 3])  
a[1] = 20  
print (a[1])
```

Here, you will get "20" as the output. However, Python doesn't allow assigning value of any other type than the typecode used at the time of creating an array. The following assignment raises TypeError.

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [1, 2, 3])  
# assignment  
a[1] = 'A'
```

It will produce the following **output** –

TypeError: 'str' object cannot be interpreted as an integer

## Python - Add Array Items

### The append() Method

The append() method adds a new element at the end of given array.

### Syntax

```
array.append(v)
```

### Parameters

- **v** – new value is added at the end of the array. The new value must be of the same type as datatype argument used while declaring array object.

### Example

```
</>
```

Open Compiler

```
import array as arr
a = arr.array('i', [1, 2, 3])
a.append(10)
print (a)
```

It will produce the following **output** –

array('i', [1, 2, 3, 10])

### The insert() Method

The array class also defines insert() method. It is possible to insert a new element at the specified index.

## Syntax

```
array.insert(i, v)
```

## Parameters

- **i** – The index at which new value is to be inserted.
- **v** – The value to be inserted. Must be of the arraytype.

## Example

```
</>
```

Open Compiler

```
import array as arr
a = arr.array('i', [1, 2, 3])
a.insert(1, 20)
print (a)
```

It will produce the following **output** –

```
array('i', [1, 20, 2, 3])
```

## The extend() Method

The extend() method in array class appends all the elements from another array of same typecode.

## Syntax

```
array.extend(x)
```

## Parameters

- **x** – Object of array.array class

## Example

```
</>

import array as arr
a = arr.array('i', [1, 2, 3, 4, 5])
b = arr.array('i', [6,7,8,9,10])
a.extend(b)
print (a)
```

It will produce the following **output** –

```
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

## Python - Remove Array Items

The array class defines two methods with the help of which we can remove an element from the array. It has `remove()` and `pop()` methods

### array.remove() Method

The `remove()` method removes the first occurrence of a given value from the array

#### Syntax

```
array.remove(v)
```

#### Parameters

- **v** – The value to be removed from the array

#### Example

```
</>

import array as arr
a = arr.array('i', [1, 2, 1, 4, 2])
a.remove(2)
print (a)
```

It will produce the following **output** –

```
array('i', [1, 1, 4, 2])
```

## array.pop() Method

The pop() method removes an element at the specified index from the array, and returns the removed element.

### Syntax

```
array.pop(i)
```

### Parameters

- **i** – The index for the element to be removed. The method returns element at *i*th position after removal.

### Example

```
</>
```

Open Compiler

```
import array as arr
a = arr.array('i', [1, 2, 1, 4, 2])
a.pop(2)
print (a)
```

It will produce the following **output** –

```
array('i', [1, 2, 4, 2])
```

## Python - Loop Arrays

Since the array object behaves like a sequence, you can iterate through its elements with the help of **for** loop or **while** loop.

### "for" Loop with Array

Take a look at the following example –

```
</>
import array as arr
a = arr.array('d', [1, 2, 3])
for x in a:
    print (x)
```

It will produce the following **output** –

```
1.0
2.0
3.0
```

## "while Loop with Array

The following example shows how you can loop through an array using a **while** loop –

```
</>
import array as arr
a = arr.array('d', [1, 2, 3])
l = len(a)
idx = 0
while idx < l:
    print (a[idx])
    idx += 1
```

## "for Loop with Array Index

We can find the length of array with built-in `len()` function. Use the it to create a range object to get the series of indices and then access the array elements in a **for** loop.

```
</>
import array as arr
a = arr.array('d', [1, 2, 3])
l = len(a)
for x in range(l):
    print (a[x])
```

You will get the same output as in the first example.

# Python - Copy Arrays

Python's built-in sequence types i.e. list, tuple and string are indexed collection of items. However, unlike arrays in C/C++, Java etc. , they are not homogenous, in the sense the elements in these types of collection may be of different types. Python's array module helps you to create object similar to Java like arrays. In this chapter, we discuss how to copy an array object to another.

Python arrays can be of string, integer or float type. The array class constructor is used as follows –

```
import array
obj = array.array(typecode[, initializer])
```

The typecode may be a character constant representing the data type.

We can assign an array to another by the assignment operator.

```
a = arr.array('i', [1, 2, 3, 4, 5])
b=a.copy()
```

However, such assignment doesn't create a new array in the memory. In Python, a variable is just a label or reference to the object in the memory. So, a is the reference to an array, and so is b. Check the id() of both a and b. Same value of id confirms that simple assignment doesn't create a copy

</>

[Open Compiler](#)

```
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5])
b=a
print (id(a), id(b))
```

It will produce the following **output** –

2771967068656 2771967068656

Because "a" and "b" refer to the same array object, any change in "a" will reflect in "b" too

```
a[2]=10
print (a,b)
```

It will produce the following **output** –

```
array('i', [1, 2, 10, 4, 5]) array('i', [1, 2, 10, 4, 5])
```

To create another physical copy of an array, we use another module in Python library, named `copy` and use `deepcopy()` function in the module. A deep copy constructs a new compound object and then, recursively inserts copies into it of the objects found in the original.

```
import array, copy
a = arr.array('i', [1, 2, 3, 4, 5])
import copy
b = copy.deepcopy(a)
```

Now check the `id()` of both "a" and "b". You will find the ids are different.

```
print (id(a), id(b))
```

It will produce the following **output** –

```
2771967069936 2771967068976
```

This proves that a new object "b" is created which is an actual copy of "a". If we change an element in "a", it is not reflected in "b".

```
a[2]=10
print (a,b)
```

It will produce the following **output** –

```
array('i', [1, 2, 10, 4, 5]) array('i', [1, 2, 3, 4, 5])
```

## Python - Reverse Arrays

In this chapter, we shall explore the different ways to rearrange the given array in the reverse order of the index. In Python, array is not one of the built-in data types. However, Python's standard library has `array` module. The `array` class helps us to create a homogenous collection of string, integer or float types.

The **syntax** used for creating array is –

```
import array  
obj = array.array(typecode[, initializer])
```

Let us first create an array consisting of a few objects of **int** type –

```
import array as arr  
a = arr.array('i', [10,5,15,4,6,20,9])
```

The array class doesn't have any built-in method to reverse array. Hence, we have to use another array. An empty array "b" is declared as follows –

```
b = arr.array('i')
```

Next, we traverse the numbers in array "a" in reverse order, and append each element to the "b" array –

```
for i in range(len(a)-1, -1, -1):  
    b.append(a[i])
```

The array "b" now holds numbers from original array in reverse order.

## Example 1

Here is the **complete code** to reverse an array in Python –

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [10,5,15,4,6,20,9])  
b = arr.array('i')  
for i in range(len(a)-1, -1, -1):  
    b.append(a[i])  
print (a, b)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9]) array('i', [9, 20, 6, 4, 15, 5, 10])
```

We can also reverse the sequence of numbers in an array using the `reverse()` method in `list` class. `List` is a built-in type in Python.

We have to first transfer the contents of an array to a list with `tolist()` method of array class –

```
a = arr.array('i', [10,5,15,4,6,20,9])
b = a.tolist()
```

We can call the `reverse()` method now –

```
b.reverse()
```

If we now convert the list back to an array, we get the array with reversed order,

```
a = arr.array('i')
a.fromlist(b)
```

## Example 2

Here is the **complete code** –

```
from array import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = a.tolist()
b.reverse()
a = arr.array('i')
a.fromlist(b)
print (a)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9])
```

## Python - Sort Arrays

Python's array module defines the array class. An object of array class is similar to the array as present in Java or C/C++. Unlike the built-in Python sequences, array is a homogenous collection of either strings, or integers, or float objects.

The array class doesn't have any function/method to give a sorted arrangement of its elements. However, we can achieve it with one of the following approaches –

- Using a sorting algorithm
- Using the `sort()` method from List

- Using the built-in sorted() function

Let's discuss each of these methods in detail.

## Using a Sorting Algorithm

We shall implement the classical **bubble sort algorithm** to obtain the sorted array. To do it, we use two nested loops and swap the elements for rearranging in sorted order.

Save the following code using a Python code editor –

&lt;/&gt;

Open Compiler

```
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
for i in range(0, len(a)):
    for j in range(i+1, len(a)):
        if(a[i] > a[j]):
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
print (a)
```

It will produce the following **output** –

```
array('i', [4, 5, 6, 9, 10, 15, 20])
```

## Using the sort() Method from List

Even though array doesn't have a **sort()** method, Python's built-in List class does have a sort method. We shall use it in the next example.

First, declare an array and obtain a list object from it, using **tolist()** method –

```
a = arr.array('i', [10,5,15,4,6,20,9])
b=a.tolist()
```

We can easily obtain the sorted list as follows –

```
b.sort()
```

All we need to do is to convert this list back to an array object –

```
a.fromlist(b)
```

Here is the **complete code** –

```
from array import array as arr
a = arr.array('i', [10, 5, 15, 4, 6, 20, 9])
b=a.tolist()
b.sort()
a = arr.array('i')
a.fromlist(b)
print (a)
```

It will produce the following **output** –

```
array('i', [4, 5, 6, 9, 10, 15, 20])
```

## Using the Builtin sorted() Function

The third technique to sort an array is with the `sorted()` function, which is a built-in function.

The **syntax** of `sorted()` function is as follows –

```
sorted(iterable, reverse=False)
```

The function returns a new list containing all items from the iterable in ascending order. Set `reverse` parameter to `True` to get a descending order of items.

The `sorted()` function can be used along with any iterable. Python array is an iterable as it is an indexed collection. Hence, an array can be used as a parameter to `sorted()` function.

```
from array import array as arr
a = arr.array('i', [4, 5, 6, 9, 10, 15, 20])
sorted(a)
print (a)
```

It will produce the following **output** –

```
array('i', [4, 5, 6, 9, 10, 15, 20])
```

# Python - Join Arrays

In Python, array is a homogenous collection of Python's built in data types such as strings, integer or float objects. However, array itself is not a built-in type, instead we need to use the array class in Python's built-in array module.

## First Method

To join two arrays, we can do it by appending each item from one array to other.

Here are two Python arrays –

```
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
```

Run a **for** loop on the array "b". Fetch each number from "b" and append it to array "a" with the following loop statement –

```
for i in range(len(b)):
    a.append(b[i])
```

The array "a" now contains elements from "a" as well as "b".

Here is the **complete code** –

```
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
for i in range(len(b)):
    a.append(b[i])
print (a, b)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9, 2, 7, 8, 11, 3, 10])
```

## Second Method

Using another method to join two arrays, first convert arrays to list objects –

```
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
x=a.tolist()
y=b.tolist()
```

The list objects can be concatenated with the '+' operator.

```
z=x+y
```

If "z" list is converted back to array, you get an array that represents the joined arrays –

```
a.fromlist(z)
```

Here is the **complete code** –

```
from array import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
x=a.tolist()
y=b.tolist()
z=x+y
a=arr.array('i')
a.fromlist(z)
print (a)
```

## Third Method

We can also use the `extend()` method from the `List` class to append elements from one list to another.

First, convert the array to a list and then call the `extend()` method to merge the two lists –

```
from array import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
a.extend(b)
print (a)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9, 2, 7, 8, 11, 3, 10])
```

# Python - Array Methods

## array.reverse() Method

Like the sequence types, the array class also supports the reverse() method which rearranges the elements in reverse order.

### Syntax

```
array.reverse()
```

### Parameters

This method has no parameters

### Example

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [1, 2, 3, 4, 5])  
a.reverse()  
print (a)
```

It will produce the following **output** –

```
array('i', [5, 4, 3, 2, 1])
```

The array class also defines the following useful methods.

## array.count() Method

The count() method returns the number of times a given element occurs in the array.

### Syntax

```
array.count(v)
```

### Parameters

- **v** – The value whose occurrences are to be counted

## Return value

The count() method returns an integer corresponding the number of times v appears in the array.

## Example

```
</> Open Compiler  
  
import array as arr  
a = arr.array('i', [1, 2, 3, 2, 5, 6, 2, 9])  
c = a.count(2)  
print ("Count of 2:", c)
```

It will produce the following **output** –

Count of 2: 3

## array.index() method

The index() method in array class finds the position of first occurrence of a given element in the array.

## Syntax

```
array.index(v)
```

## Parameters

- **v** – the value for which the index is to be found

## Example

```
a = arr.array('i', [1, 2, 3, 2, 5, 6, 2, 9])  
c = a.index(2)  
print ("index of 2:", c)
```

It will produce the following **output** –

index of 2: 1

## array.fromlist() Method

The fromlist() method appends items from a Python list to the array object.

### Syntax

```
array.fromlist(l)
```

### Parameters

- **i** – The list, items of which are appended to the array. All items in the list must be of same arrtype.

### Example

```
</>
```

Open Compiler

```
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5])
lst = [6, 7, 8, 9, 10]
c = a.fromlist(lst)
print (a)
```

It will produce the following **output** –

array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

## array.tofile() Method

The tofile() method in array class writes all items (as machine values) in the array to the file object f.

### Syntax

```
array.tofile(f)
```

## Parameters

- **f** – the file object obtained with open() function. The file to be opened in wb mode.

## Example

```
</>
```

Open Compiler

```
import array as arr
f = open('list.txt','wb')
arr.array("i", [10, 20, 30, 40, 50]).tofile(f)
f.close()
```

## Output

After running the above code, a file named as "list.txt" will be created in the current directory.

## array.fromfile() Method

The fromfile() method reads a binary file and appends specified number of items to the array object.

## Syntax

```
array.fromfile(f, n)
```

## Parameters

- **f** – The file object referring to a disk file opened in rb mode
- **n** – number of items to be appended

## Example

```
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5])
```

```
f = open("list.txt", "rb")
a.fromfile(f, 5)
print (a)
```

It will produce the following **output** –

```
array('i', [1, 2, 3, 4, 5, 10, 20, 30, 40, 50])
```

## Python - Array Exercises

### Example 1

Python program to find the largest number in an array –

```
</> Open Compiler
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
print (a)
largest = a[0]
for i in range(1, len(a)):
    if a[i]>largest:
        largest=a[i]
print ("Largest number:", largest)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9])
Largest number: 20
```

### Example 2

Python program to store all even numbers from an array in another array –

```
</> Open Compiler
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
```

```
print (a)
b = arr.array('i')
for i in range(len(a)):
    if a[i]%2 == 0:
        b.append(a[i])
print ("Even numbers:", b)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9])
Even numbers: array('i', [10, 4, 6, 20])
```

## Example 3

Python program to find the average of all numbers in a Python array –

</>

Open Compiler

```
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
print (a)
s = 0
for i in range(len(a)):
    s+=a[i]
avg = s/len(a)
print ("Average:", avg)

# Using sum() function
avg = sum(a)/len(a)
print ("Average:", avg)
```

It will produce the following **output** –

```
array('i', [10, 5, 15, 4, 6, 20, 9])
Average: 9.857142857142858
Average: 9.857142857142858
```

## Exercise Programs

- Python program find difference between each number in the array and the average of all numbers
- Python program to convert a string in an array
- Python program to split an array in two and store even numbers in one array and odd numbers in the other.
- Python program to perform insertion sort on an array.
- Python program to store the Unicode value of each character in the given array.

## Python - File Handling

When we use any computer application, some data needs to be provided. Data is stored in computer's main memory (RAM) until the application is running. Thereafter, memory contents from RAM are erased.

We would like to store it in such a way that it can be retrieved whenever required in a persistent medium such as a disk file.

Python uses built-in **input()** and **print()** functions to perform standard input/output operations. Python program interacts with these IO devices through standard stream objects `stdin` and `stdout` defined in `sys` module.

The `input()` function reads bytes from a standard input stream device i.e. keyboard. Hence both the following statements read input from the user.

```
name = input()  
#is equivalent to  
import sys  
name = sys.stdin.readline()
```

The `print()` function on the other hand, sends the data towards standard output stream device, i.e., the display monitor. It is a convenience function emulating `write()` method of `stdout` object.

```
print (name)  
  
#is equivalent to  
import sys  
sys.stdout.write(name)
```

Any object that interacts with input and output steam is called File object. Python's built-in function `open()` returns a file object.

## The open() Function

This function creates a file object, which would be utilized to call other support methods associated with it.

### Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are the parameter details –

- **file\_name** – The file\_name argument is a string value that contains the name of the file that you want to access.
- **access\_mode** – The access\_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
- **buffering** – If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

## File Opening Modes

Following are the file opening modes –

| Sr.No. | Modes & Description                                                                                                                             |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>r</b><br>Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.                   |
| 2      | <b>rb</b><br>Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| 3      | <b>r+</b><br>Opens a file for both reading and writing. The file pointer placed at the beginning of the file.                                   |
| 4      | <b>rb+</b>                                                                                                                                      |

|    |                                                                                                                                                                                                                                                          |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.                                                                                                                                        |
| 5  | <b>w</b><br>Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.                                                                                                           |
| 6  | <b>b</b><br>Opens the file in binary mode                                                                                                                                                                                                                |
| 7  | <b>t</b><br>Opens the file in text mode (default)                                                                                                                                                                                                        |
| 8  | <b>+</b><br>open file for updating (reading and writing)                                                                                                                                                                                                 |
| 9  | <b>wb</b><br>Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.                                                                                         |
| 10 | <b>w+</b><br>Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.                                                                         |
| 11 | <b>wb+</b><br>Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.                                                       |
| 12 | <b>a</b><br>Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.                                           |
| 13 | <b>ab</b><br>Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.                         |
| 14 | <b>a+</b><br>Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.                   |
| 15 | <b>ab+</b><br>Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

16

**x**

open for exclusive creation, failing if the file already exists

Once a file is opened and you have one file object, you can get various information related to that file.

## Example

&lt;/&gt;

Open Compiler

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not: ", fo.closed)
print ("Opening mode: ", fo.mode)
fo.close()
```

It will produce the following **output** –

Name of the file: foo.txt

Closed or not: False

Opening mode: wb

## Python - Write to File

To write data to a file in Python, you need to open a file. Any object that interacts with input and output stream is called File object. Python's built-in function `open()` returns a file object.

```
fileObject = open(file_name [, access_mode][, buffering])
```

After you obtain the file object with the `open()` function, you can use the `write()` method to write any string to the file represented by the file object. It is important to note that Python strings can have binary data and not just text.

The `write()` method does not add a newline character ('\n') to the end of the string.

## Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

## Example

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opened file
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. The program shows no output as such, although if you would open this file with any text editor application such as Notepad, it would have the following content –

```
Python is a great language.
Yeah its great!!
```

## Writing in Binary Mode

By default, read/write operation on a file object are performed on text string data. If we want to handle files of different other types such as media (mp3), executables (exe), pictures (jpg) etc., we need to add 'b' prefix to read/write mode.

Following statement will convert a string to bytes and write in a file.

```
f=open('test.bin', 'wb')
data=b"Hello World"
f.write(data)
f.close()
```

Conversion of text string to bytes is also possible using encode() function.

```
data="Hello World".encode('utf-8')
```

## Appending to a File

When any existing file is opened in 'w' mode to store additional text, its earlier contents are erased. Whenever a file is opened with write permission, it is treated as if it is a new file. To add data to an existing file, use 'a' for append mode.

## Syntax

```
fileobject = open(file_name, "a")
```

## Example

```
# Open a file in append mode
fo = open("foo.txt", "a")
text = "TutorialsPoint has a fabulous Python tutorial"
fo.write(text)

# Close opened file
fo.close()
```

When the above program is executed, no output is shown, but a new line is appended to foo.txt. To verify, open with a text editor.

```
Python is a great language.
Yeah its great!!
TutorialsPoint has a fabulous Python tutorial
```

## Using the w+ Mode

When a file is opened for writing (with 'w' or 'a'), it is not possible to perform write operation at any earlier byte position in the file. Th 'w+' mode enables using write() as well as read() methods without closing a file. The File object supports seek() unction to rewind the stream to any desired byte position.

Following is the syntax for seek() method –

```
fileObject.seek(offset[, whence])
```

## Parameters

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Let us use the seek() method to show how simultaneous read/write operation on a file can be done.

## Example

The following program opens the file in w+ mode (which is a read-write mode), adds some data. Then it seeks a certain position in file and overwrites its earlier contents with new text.

```
# Open a file in read-write mode
fo=open("foo.txt","w+")
fo.write("This is a rat race")
fo.seek(10,0)
data=fo.read(3)
fo.seek(10,0)
fo.write('cat')
fo.close()
```

## Output

If we open the file in Read mode (or seek the starting position while in w+ mode), and read the contents, it shows –

```
This is a cat race
```

# Python - Read Files

To programmatically read data from a file using Python, it must be opened first. Use the built-in open() function –

```
file object = open(file_name [, access_mode][, buffering])
```

Here are the parameter details –

- **file\_name** – The file\_name argument is a string value that contains the name of the file that you want to access.
- **access\_mode** – The access\_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is an optional parameter and the default file access mode is read (r).

These two statements are identical –

```
fo = open("foo.txt", "r")
fo = open("foo.txt")
```

To read data from the opened file, use `read()` method of the File object. It is important to note that Python strings can have binary data apart from the text data.

## Syntax

```
fileObject.read([count])
```

## Parameters

- **count** – Number of bytes to be read.

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

## Example

```
# Open a file
fo = open("foo.txt", "r")
text = fo.read()
print (text)

# Close the opened file
fo.close()
```

It will produce the following **output** –

```
Python is a great language.
Yeah its great!!
```

## Reading in Binary Mode

By default, read/write operation on a file object are performed on text string data. If we want to handle files of different other types such as media (mp3), executables (exe), pictures (jpg) etc., we need to add 'b' prefix to read/write mode.

Assuming that the **test.bin** file has already been written with binary mode.

```
f=open('test.bin', 'wb')
data=b"Hello World"
f.write(data)
f.close()
```

We need to use 'rb' mode to read binary file. Returned value of read() method is first decoded before printing

```
f=open('test.bin', 'rb')
data=f.read()
print (data.decode(encoding='utf-8'))
```

It will produce the following **output** –

```
Hello World
```

## Read Integer Data from File

In order to write integer data in a binary file, the integer object should be converted to bytes by to\_bytes() method.

```
n=25
n.to_bytes(8,'big')
f=open('test.bin', 'wb')
data=n.to_bytes(8,'big')
f.write(data)
```

To read back from a binary file, convert the output of read() function to integer by using the from\_bytes() function.

```
f=open('test.bin', 'rb')
data=f.read()
n=int.from_bytes(data, 'big')
print (n)
```

## Read Float Data from File

For floating point data, we need to use **struct** module from Python's standard library.

```
import struct
x=23.50
```

```
data=struct.pack('f',x)
f=open('test.bin', 'wb')
f.write(data)
```

Unpacking the string from read() function to retrieve the float data from binary file.

```
f=open('test.bin', 'rb')
data=f.read()
x=struct.unpack('f', data)
print (x)
```

## Using the r+ Mode

When a file is opened for reading (with 'r' or 'rb'), it is not possible to write data in it. We need to close the file before doing other operation. In order to perform both operations simultaneously, we have to add '+' character in the mode parameter. Hence 'w+' or 'r+' mode enables using write() as well as read() methods without closing a file.

The File object also supports the seek() function to rewind the stream to read from any desired byte position.

Following is the syntax for seek() method –

```
fileObject.seek(offset[, whence])
```

## Parameters

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Let us use the seek() method to show how to read data from a certain byte position.

## Example

This program opens the file in w+ mode (which is a read-write mode), adds some data. Then it seeks a certain position in file and overwrites its earlier contents with new text.

```
fo=open("foo.txt","r+")
fo.seek(10,0)
data=fo.read(3)
```

```
print (data)
fo.close()
```

It will produce the following **output** –

```
rat
```

## Python Simultaneous Read/Write

When a file is opened for writing (with 'w' or 'a'), it is not possible to read from it and vice versa. Doing so throws UnSupportedOperation error. We need to close the file before doing other operation.

In order to perform both operations simultaneously, we have to add '+' character in the mode parameter. Hence 'w+' or 'r+' mode enables using write() as well as read() methods without closing a file. The File object also supports the seek() function to rewind the stream to any desired byte position.

### The seek() Method

The method seek() sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

There is no return value. Note that if the file is opened for appending using either 'a' or 'a+', any seek() operations will be undone at the next write.

If the file is only opened for writing in append mode using 'a', this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+').

If the file is opened in text mode using 't', only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

## Syntax

Following is the syntax for seek() method –

```
fileObject.seek(offset[, whence])
```

## Parameters

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Let us use the `seek()` method to show how simultaneous read/write operation on a file can be done.

The following program opens the file in `w+` mode (which is a read-write mode), adds some data. Then it seeks a certain position in file and overwrites its earlier contents with new text.

## Example

```
</> Open Compiler  
  
# Open a file in read-write mode  
fo=open("foo.txt","w+")  
fo.write("This is a rat race")  
fo.seek(10,0)  
data=fo.read(3)  
fo.seek(10,0)  
fo.write('cat')  
fo.seek(0,0)  
data=fo.read()  
print (data)  
fo.close()
```

## Output

This is a cat race

# Python - Renaming and Deleting Files

Python `os` module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module, you need to import it first and then you can call any related functions.

## rename() Method

The `rename()` method takes two arguments, the current filename and the new filename.

## Syntax

```
os.rename(current_file_name, new_file_name)
```

## Example

Following is an example to rename an existing file "test1.txt" to "test2.txt" –

```
#!/usr/bin/python3
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

## remove() Method

You can use the `remove()` method to delete files by supplying the name of the file to be deleted as the argument.

## Syntax

```
os.remove(file_name)
```

## Example

Following is an example to delete an existing file "test2.txt" –

```
#!/usr/bin/python3
import os
# Delete file test2.txt
os.remove("text2.txt")
```

# Python - Directories

All files are contained within various directories, and Python has no problem handling these too. The `os` module has several methods that help you create, remove, and change directories.

## The `mkdir()` Method

You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

## Syntax

```
os.mkdir("newdir")
```

## Example

Following is an example to create a directory `test` in the current directory –

```
#!/usr/bin/python3
import os

# Create a directory "test"
os.mkdir("test")
```

## The `chdir()` Method

You can use the `chdir()` method to change the current directory. The `chdir()` method takes an argument, which is the name of the directory that you want to make the current directory.

## Syntax

```
os.chdir("newdir")
```

## Example

Following is an example to go into `"/home/newdir"` directory –

```
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

## The `getcwd()` Method

The `getcwd()` method displays the current working directory.

## Syntax

```
os.getcwd()
```

## Example

Following is an example to give current directory –

```
#!/usr/bin/python3
import os

# This would give location of the current directory
os.getcwd()
```

## The rmdir() Method

The `rmdir()` method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

## Syntax

```
os.rmdir('dirname')
```

## Example

Following is an example to remove the `"/tmp/test"` directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
#!/usr/bin/python3
import os

# This would remove "/tmp/test" directory.
os.rmdir( "/tmp/test" )
```

# Python - File Methods

A file object is created using `open()` function. The `file` class defines the following methods with which different file IO operations can be done. The methods can be used with any file like object such as byte stream or network stream.

| Sr.No. | Methods & Description                                                                                                                                                                                                                                                                                          |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>file.close()</b><br>Close the file. A closed file cannot be read or written any more.                                                                                                                                                                                                                       |
| 2      | <b>file.flush()</b><br>Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.                                                                                                                                                                                          |
| 3      | <b>file.fileno()</b><br>Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.                                                                                                                                                 |
| 4      | <b>file.isatty()</b><br>Returns True if the file is connected to a tty(-like) device, else False.                                                                                                                                                                                                              |
| 5      | <b>next(file)</b><br>Returns the next line from the file each time it is being called.                                                                                                                                                                                                                         |
| 6      | <b>file.read([size])</b><br>Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).                                                                                                                                                                                    |
| 7      | <b>file.readline([size])</b><br>Reads one entire line from the file. A trailing newline character is kept in the string.                                                                                                                                                                                       |
| 8      | <b>file.readlines([sizehint])</b><br>Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. |
| 9      | <b>file.seek(offset[, whence])</b><br>Sets the file's current position                                                                                                                                                                                                                                         |
| 10     | <b>file.tell()</b><br>Returns the file's current position                                                                                                                                                                                                                                                      |
| 11     | <b>file.truncate([size])</b><br>Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.                                                                                                                                                             |
| 12     | <b>file.write(str)</b><br>Writes a string to the file. There is no return value.                                                                                                                                                                                                                               |
| 13     | <b>file.writelines(sequence)</b>                                                                                                                                                                                                                                                                               |

Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

Let us go through the above methods briefly.

## Python - OS File/Directory Methods

The **os** module provides a big range of useful methods to manipulate files. Most of the useful methods are listed here –

| Sr.No. | Methods with Description                                                                                                                                                                                  |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>os.close(fd)</b><br>Close file descriptor fd.                                                                                                                                                          |
| 2      | <b>os.closerange(fd_low, fd_high)</b><br>Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors.                                                                      |
| 3      | <b>os.dup(fd)</b><br>Return a duplicate of file descriptor fd.                                                                                                                                            |
| 4      | <b>os.fdatasync(fd)</b><br>Force write of file with filedescriptor fd to disk.                                                                                                                            |
| 5      | <b>os.fdopen(fd[, mode[, bufsize]])</b><br>Return an open file object connected to the file descriptor fd.                                                                                                |
| 6      | <b>os.fsync(fd)</b><br>Force write of file with filedescriptor fd to disk.                                                                                                                                |
| 7      | <b>os.ftruncate(fd, length)</b><br>Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.                                                                     |
| 8      | <b>os.lseek(fd, pos, how)</b><br>Set the current position of file descriptor fd to position pos, modified by how.                                                                                         |
| 9      | <b>os.open(file, flags[, mode])</b><br>Open the file file and set various flags according to flags and possibly its mode according to mode.                                                               |
| 10     | <b>os.read(fd, n)</b><br>Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned. |

|    |                                                                                                                      |
|----|----------------------------------------------------------------------------------------------------------------------|
| 11 | <b>os.tmpfile()</b><br>Return a new file object opened in update mode (w+b).                                         |
| 12 | <b>os.write(fd, str)</b><br>Write the string str to file descriptor fd. Return the number of bytes actually written. |

## Python - OOP Concepts

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts. However, here is a small introduction of Object-Oriented Programming (OOP) to help you.

### Procedural Oriented Approach

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

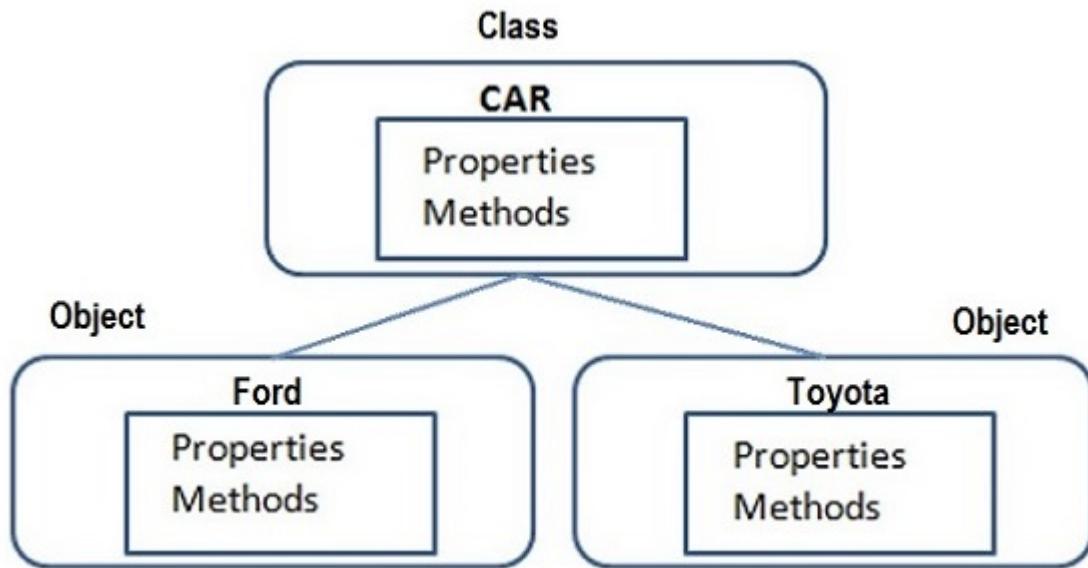
Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

Prominent problems related to procedural approach are as follows –

- Its top-down approach makes the program difficult to maintain.
- It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
- It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
- Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.

## Python - OOP Concepts

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



## Attributes

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

## Behavior

Processing attributes associated with an object.

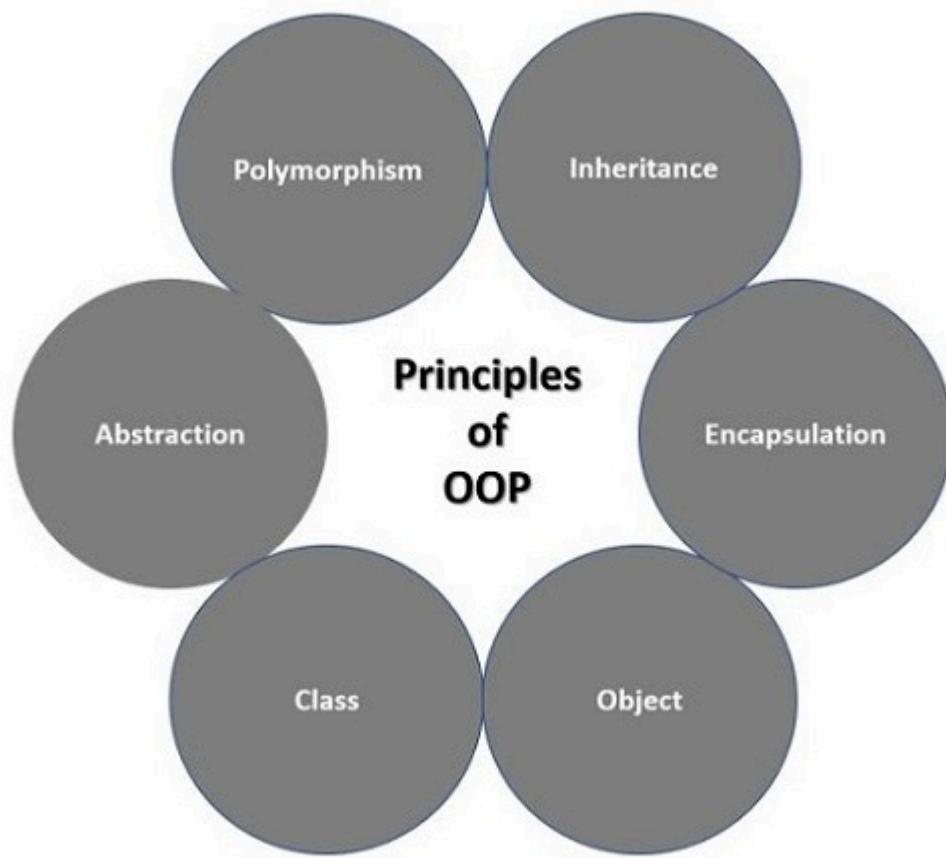
- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class.

Object-oriented programming paradigm is characterized by the following principles –



## Class

A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

## Object

An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

## Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So

object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

## Inheritance

A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called base or parent class, while new class is called child or sub class.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

## Polymorphism

Polymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.

# Python - Object and Classes

Python is a highly object-oriented language. In Python, each and every element in a Python program is an object of one or the other class. A number, string, list, dictionary etc. used in a program they are objects of corresponding built-in classes.

## Example

&lt;/&gt;

Open Compiler

```
num=20
print (type(num))
num1=55.50
print (type(num1))
s="TutorialsPoint"
print (type(s))
dct={'a':1,'b':2,'c':3}
print (type(dct))
def SayHello():
    print ("Hello World")
    return
print (type(SayHello))
```

When you execute this code, it will produce the following **output** –

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'dict'>
<class 'function'>
```

In Python, the Object class is the base or parent class for all the classes, built-in as well as user defined.

The **class** keyword is used to define a new class. The name of the class immediately follows the keyword class followed by a colon as follows –

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

- The class has a documentation string, which can be accessed via `ClassName.__doc__`.
- The `class_suite` consists of all the component statements defining class members, data attributes and functions.

## Example

```
class Employee(object):
    'Common base class for all employees'
    pass
```

Any class in Python is a subclass of object class, hence object is written in parentheses. However, later versions of Python don't require object to be put in parentheses.

```
class Employee:
    'Common base class for all employees'
    pass
```

To define an object of this class, use the following syntax –

```
e1 = Employee()
```

## Python - Class Attributes

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute –

- **`__dict__`** – Dictionary containing the class's namespace.
- **`__doc__`** – Class documentation string or none, if undefined.
- **`__name__`** – Class name.
- **`__module__`** – Module name in which the class is defined. This attribute is "`__main__`" in interactive mode.
- **`__bases__`** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class, let us try to access all these attributes –

&lt;/&gt;

Open Compiler

```
class Employee:  
    def __init__(self, name="Bhavana", age=24):  
        self.name = name  
        self.age = age  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", age: ", self.age)  
  
    print ("Employee.__doc__:", Employee.__doc__)  
    print ("Employee.__name__:", Employee.__name__)  
    print ("Employee.__module__:", Employee.__module__)  
    print ("Employee.__bases__:", Employee.__bases__)  
    print ("Employee.__dict__:", Employee.__dict__ )
```

It will produce the following **output** –

```
Employee.__doc__: None  
Employee.__name__: Employee  
Employee.__module__: __main__  
Employee.__bases__: (<class 'object'>,)  
Employee.__dict__: {'__module__': '__main__', '__init__': <function Employee.__init__>}
```

## Class Variables

In the above Employee class example, name and age are instance variables, as their values may be different for each object. A class attribute or variable whose value is shared among all the instances of a in this class. A class attribute represents common attribute of all objects of a class.

Class attributes are not initialized inside `__init__()` constructor. They are defined in the class, but outside any method. They can be accessed by name of class in addition to object. In other words, a class attribute available to class as well as its object.

## Example

Let us add a class variable called **empCount** in Employee class. For each object declared, the `__init__()` method is automatically called. This method initializes the instance variables as well as increments the **empCount** by 1.

&lt;/&gt;

Open Compiler

```
class Employee:  
    empCount = 0  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
        Employee.empCount += 1  
        print ("Name: ", self.__name, "Age: ", self.__age)  
        print ("Employee Number:", Employee.empCount)  
  
e1 = Employee("Bhavana", 24)  
e2 = Employee("Rajesh", 26)  
e3 = Employee("John", 27)
```

## Output

We have declared three objects. Every time, the **empCount** increments by 1.

```
Name: Bhavana Age: 24  
Employee Number: 1  
Name: Rajesh Age: 26  
Employee Number: 2  
Name: John Age: 27  
Employee Number: 3
```

# Python - Class Methods

An instance method accesses the instance variables of the calling object because it takes the reference to the calling object. But it can also access the class variable as it is common to all the objects.

Python has a built-in function `classmethod()` which transforms an instance method to a class method which can be called with the reference to the class only and not the object.

## Syntax

```
classmethod(instance_method)
```

## Example

In the Employee class, define a `showcount()` instance method with the "`self`" argument (reference to calling object). It prints the value of `empCount`. Next, transform the method to class method `counter()` that can be accessed through the class reference.

```
class Employee:  
    empCount = 0  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
        Employee.empCount += 1  
    def showcount(self):  
        print (self.empCount)  
    counter=classmethod(showcount)  
  
e1 = Employee("Bhavana", 24)  
e2 = Employee("Rajesh", 26)  
e3 = Employee("John", 27)  
  
e1.showcount()  
Employee.counter()
```

## Output

Call `showcount()` with object and call `count()` with class, both show the value of employee count.

3

3

Using `@classmethod()` decorator is the prescribed way to define a class method as it is more convenient than first declaring an instance method and then transforming to a class method.

```
@classmethod  
def showcount(cls):  
    print (cls.empCount)  
  
Employee.showcount()
```

The class method acts as an alternate constructor. Define a `newemployee()` class method with arguments required to construct a new object. It returns the constructed object, something that the `__init__()` method does.

```
@classmethod  
def showcount(cls):  
    print (cls.empCount)  
    return  
  
@classmethod  
def newemployee(cls, name, age):  
    return cls(name, age)  
  
e1 = Employee("Bhavana", 24)  
e2 = Employee("Rajesh", 26)  
e3 = Employee("John", 27)  
e4 = Employee.newemployee("Anil", 21)  
  
Employee.showcount()
```

There are four Employee objects now.

## Python - Static Methods

is that the static method doesn't have a mandatory argument like reference to the object – `self` or reference to the class – `cls`. Python's standard library function `staticmethod()` returns a static method.

In the Employee class below, a method is converted into a static method. This static method can now be called by its object or reference of class itself.

```

class Employee:
    empCount = 0

    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        Employee.empCount += 1

    #@staticmethod
    def showcount():
        print (Employee.empCount)
        return

    counter = staticmethod(showcount)

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.counter()
Employee.counter()

```

Python also has `@staticmethod` decorator that conveniently returns a static method.

```

@staticmethod
def showcount():
    print (Employee.empCount)
e1.showcount()
Employee.showcount()

```

## Python - Constructors

In object-oriented programming, an object of a class is characterized by one or more instance variables or attributes, whose values are unique to each object. For example, if the `Employee` class has an instance attribute as `name`. Each of its objects `e1` and `e2` may have different value for the `name` variable.

A constructor is an instance method in a class, that is automatically called whenever a new object of the class is declared. The constructor' role is to assign value to instance variables as soon as the object is declared.

Python uses a special method called `__init__()` to initialize the instance variables for the object, as soon as it is declared.

The `__init__()` method acts as a constructor. It needs a mandatory argument `self`, which is the reference to the object.

```
def __init__(self):  
    #initialize instance variables
```

The `__init__()` method as well as any instance method in a class has a mandatory parameter, `self`. However, you can give any name to the first parameter, not necessarily `self`.

Let us define the constructor in `Employee` class to initialize name and age as instance variables. We can then access these attributes of its object.

## Example

</>

Open Compiler

```
class Employee:  
    'Common base class for all employees'  
    def __init__(self):  
        self.name = "Bhavana"  
        self.age = 24  
  
    e1 = Employee()  
    print ("Name: {}".format(e1.name))  
    print ("age: {}".format(e1.age))
```

It will produce the following **output** –

Name: Bhavana

age: 24

## Parameterized Constructor

For the above `Employee` class, each object we declare will have same value for its instance variables name and age. To declare objects with varying attributes instead of the default, define arguments for the `__init__()` method. (A method is nothing but a function defined inside a class.)

## Example

In this example, the `__init__()` constructor has two formal arguments. We declare Employee objects with different values –

```
</> Open Compiler

class Employee:
    'Common base class for all employees'
    def __init__(self, name, age):
        self.name = name
        self.age = age

e1 = Employee("Bhavana", 24)
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))
```

It will produce the following **output** –

```
Name: Bhavana
age: 24
Name: Bharat
age: 25
```

You can assign defaults to the formal arguments in the constructor so that the object can be instantiated with or without passing parameters.

```
</> Open Compiler

class Employee:
    'Common base class for all employees'
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age

e1 = Employee()
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
```

```
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))
```

It will produce the following **output** –

```
Name: Bhavana
age: 24
Name: Bharat
age: 25
```

## Python - Instance Methods

In addition to the `__init__()` constructor, there may be one or more instance methods defined in a class. A method with `self` as one of the formal arguments is called instance method, as it is called by a specific object.

### Example

In the following example a `displayEmployee()` method has been defined. It returns the name and age attributes of the `Employee` object that calls the method.

</>

Open Compiler

```
class Employee:
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age
    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.age)

e1 = Employee()
e2 = Employee("Bharat", 25)

e1.displayEmployee()
e2.displayEmployee()
```

It will produce the following **output** –

```
Name : Bhavana , age: 24
```

Name : Bharat , age: 25

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.salary = 7000 # Add a 'salary' attribute.
emp1.name = 'xyz' # Modify 'name' attribute.
del emp1.salary # Delete 'salary' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj, name)** – to check if an attribute exists or not.
- The **setattr(obj, name, value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
print (hasattr(e1, 'salary')) # Returns true if 'salary' attribute exists
print (getattr(e1, 'name')) # Returns value of 'name' attribute
setattr(e1, 'salary', 7000) # Set attribute 'salary' at 8
delattr(e1, 'age') # Delete attribute 'age'
```

It will produce the following **output** –

False  
Bhavana

## Python - Access Modifiers

The languages such as C++ and Java, use access modifiers to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only.

- Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures implementation of principle of encapsulation.

- **Protected members** are accessible from within the class as well as by classes derived from that class.

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a class are public.

## Example

Here, we have Employee class with instance variables name and age. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

&lt;/&gt;

Open Compiler

```
class Employee:
    'Common base class for all employees'
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age

    e1 = Employee()
    e2 = Employee("Bharat", 25)

    print ("Name: {}".format(e1.name))
    print ("age: {}".format(e1.age))
    print ("Name: {}".format(e2.name))
    print ("age: {}".format(e2.age))
```

It will produce the following **output** –

```
Name: Bhavana
age: 24
Name: Bharat
age: 25
```

Python doesn't enforce restrictions on accessing any instance variable or method. However, Python prescribes a convention of prefixing name of variable/method with single or double underscore to emulate behavior of protected and private access modifiers.

To indicate that an instance variable is private, prefix it with double underscore (such as "age"). To imply that a certain instance variable is protected, prefix it with single underscore (such as "\_salary")

## Example

Let us modify the Employee class. Add another instance variable salary. Make **age** private and **salary** as protected by prefixing double and single underscores respectively.

&lt;/&gt;

Open Compiler

```
class Employee:  
    def __init__(self, name, age, salary):  
        self.name = name # public variable  
        self.__age = age # private variable  
        self._salary = salary # protected variable  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", age: ", self.__age, ", salary: ", self._sal  
  
e1=Employee("Bhavana", 24, 10000)  
  
print (e1.name)  
print (e1._salary)  
print (e1.__age)
```

When you run this code, it will produce the following **output** –

```
Bhavana  
10000  
Traceback (most recent call last):  
File "C:\Users\user\example.py", line 14, in <module>  
    print (e1.__age)  
    ^^^^^^^^^^  
AttributeError: 'Employee' object has no attribute '__age'
```

Python displays AttributeError because `__age` is private, and not available for use outside the class.

## Name Mangling

Python doesn't block access to private data, it just leaves for the wisdom of the programmer, not to write any code that access it from outside the class. You can still access the private members by Python's name mangling technique.

Name mangling is the process of changing name of a member with double underscore to the form **object.\_class\_\_variable**. If so required, it can still be accessed from outside the class, but the practice should be refrained.

In our example, the private instance variable "`__name`" is mangled by changing it to the format

```
obj._class__privatevar
```

So, to access the value of "`__age`" instance variable of "e1" object, change it to "`e1._Employee__age`".

Change the `print()` statement in the above program to –

```
print (e1._Employee__age)
```

It now prints 24, the age of **e1**.

## Python Property Object

Python's standard library has a built-in `property()` function. It returns a property object. It acts as an interface to the instance variables of a Python class.

The encapsulation principle of object-oriented programming requires that the instance variables should have a restricted private access. Python doesn't have efficient mechanism for the purpose. The `property()` function provides an alternative.

The `property()` function uses the getter, setter and delete methods defined in a class to define a property object for the class.

## Syntax

```
property(fget=None, fset=None, fdel=None, doc=None)
```

## Parameters

- **fget** – an instance method that retrieves value of an instance variable.
- **fset** – an instance method that assigns value to an instance variable.
- **fdel** – an instance method that removes an instance variable
- **fdoc** – Documentation string for the property.

The function uses getter and setter methods to return the property object.

## Getters and Setter Methods

A getter method retrieves the value of an instance variable, usually named as `get_varname`, whereas the setter method assigns value to an instance variable – named as `set_varname`.

Let us define getter methods `get_name()` and `get_age()`, and setters `set_name()` and `set_age()` in the `Employee` class.

### Example

&lt;/&gt;

Open Compiler

```
class Employee:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def get_name(self):  
        return self.__name  
    def get_age(self):  
        return self.__age  
    def set_name(self, name):  
        self.__name = name  
        return  
    def set_age(self, age):  
        self.__age=age  
  
e1=Employee("Bhavana", 24)  
print ("Name:", e1.get_name(), "age:",  
  
e1.get_age())  
e1.set_name("Archana")  
e1.set_age(21)  
print ("Name:", e1.get_name(), "age:", e1.get_age())
```

It will produce the following **output** –

Name: Bhavana age: 24

Name: Archana age: 21

The getter and setter methods can retrieve or assign value to instance variables. The `property()` function uses them to add property objects as class attributes.

The name property is defined as –

```
name = property(get_name, set_name, "name")
```

Similarly, you can add the `age` property –

```
age = property(get_age, set_age, "age")
```

The advantage of the property object is that you can use to retrieve the value of its associated instance variable, as well as assign value.

For example,

```
print (e1.name) displays value of e1.__name  
e1.name = "Archana" assigns value to e1.__age
```

## Example

The complete program with property objects and their use is given below –

```
</> Open Compiler  
  
class Employee:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def get_name(self):  
        return self.__name  
    def get_age(self):  
        return self.__age  
    def set_name(self, name):  
        self.__name = name  
        return  
    def set_age(self, age):  
        self.__age=age  
        return  
    name = property(get_name, set_name, "name")  
    age = property(get_age, set_age, "age")
```

```
e1=Employee("Bhavana", 24)
print ("Name:", e1.name, "age:", e1.age)

e1.name = "Archana"
e1.age = 23
print ("Name:", e1.name, "age:", e1.age)
```

It will produce the following **output** –

```
Name: Bhavana age: 24
Name: Archana age: 23
```

## Python - Inheritance

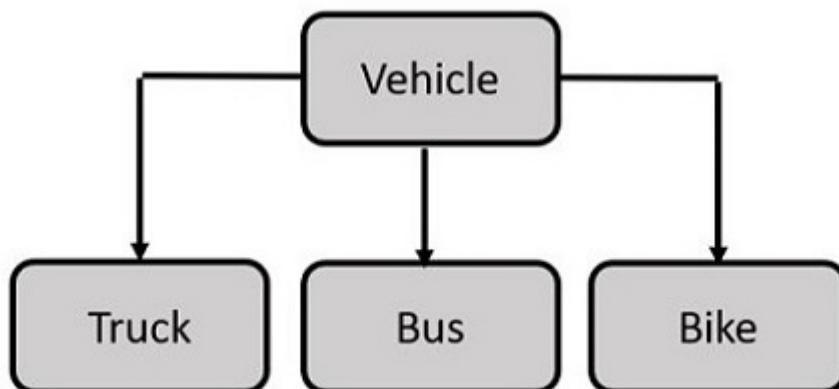
Inheritance is one of the most important features of Object-oriented programming methodology. It is most often used in software development process using many languages such as Java, PHP, Python, etc.

Instead of starting from scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

Instead of starting from scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

If you have to design a new class whose most of the attributes are already well defined in an existing class, then why redefine them? Inheritance allows capabilities of existing class to be reused and if required extended to design new class.

Inheritance comes into picture when a new class possesses 'IS A' relationship with an existing class. Car IS a vehicle. Bus IS a vehicle; Bike IS also a vehicle. Vehicle here is the parent class, whereas car, bus and bike are the child classes.



## Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```

## Example

</>

Open Compiler

```
class Parent: # define parent class  
    def __init__(self):  
        self.attr = 100  
        print ("Calling parent constructor")  
    def parentMethod(self):  
        print ('Calling parent method')  
    def set_attr(self, attr):  
        self.attr = attr  
  
    def get_attr(self):  
        print ("Parent attribute :", self.attr)  
  
class Child(Parent): # define child class  
    def __init__(self):  
        print ("Calling child constructor")  
  
    def childMethod(self):  
        print ('Calling child method')  
  
c = Child()      # instance of child  
c.childMethod() # child calls its method  
c.parentMethod() # calls parent's method  
c.set_attr(200)  # again call parent's method  
c.get_attr()    # again call parent's method
```

## Output

When you execute this code, it will produce the following output –

```
Calling child constructor  
Calling child method  
Calling parent method  
Parent attribute : 200
```

## Python - Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The Child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

### Syntax

```
class parent1:  
    #statements  
  
class parent2:  
    #statements  
  
class child(parent1, parent2):  
    #statements
```

Python's standard library has a built-in `divmod()` function that returns a two-item tuple. First number is the division of two arguments, the second is the mod value of the two operands.

### Example

This example tries to emulate the `divmod()` function. We define two classes `division` and `modulus`, and then have a `div_mod` class that inherits them.

```
class division:  
    def __init__(self, a,b):  
        self.n=a  
        self.d=b  
    def divide(self):  
        return self.n/self.d  
class modulus:  
    def __init__(self, a,b):  
        self.n=a  
        self.d=b  
    def mod_divide(self):
```

```

        return self.n%self.d

class div_mod(dvision,modulus):
    def __init__(self, a,b):
        self.n=a
        self.d=b
    def div_and_mod(self):
        divval=division.divide(self)
        modval=modulus.mod_divide(self)
        return (divval, modval)

```

The child class has a new method `div_and_mod()` which internally calls the `divide()` and `mod_divide()` methods from its inherited classes to return the division and mod values.

```

x=div_mod(10,3)
print ("division:",x.divide())
print ("mod_division:",x.mod_divide())
print ("divmod:",x.div_and_mod())

```

## Output

```

division: 3.333333333333335
mod_division: 1
divmod: (3.333333333333335, 1)

```

## Method Resolution Order (MRO)

The term "method resolution order" is related to multiple inheritance in Python. In Python, inheritance may be spread over more than one levels. Let us say A is the parent of B, and B the parent for C. The class C can override the inherited method or its object may invoke it as defined in its parent. So, how does Python find the appropriate method to call.

Each Python has a `mro()` method that returns the hierarchical order that Python uses to resolve the method to be called. The resolution order is from bottom of inheritance order to top.

In our previous example, the `div_mod` class inherits `division` and `modulus` classes. So, the `mro` method returns the order as follows –

```
[<class '__main__.div_mod'>, <class '__main__.division'>, <class '__main__.modulus'>]
```

# Python - Polymorphism

The term "polymorphism" refers to a function or method taking different form in different contexts. Since Python is a dynamically typed language, Polymorphism in Python is very easily implemented.

If a method in a parent class is overridden with different business logic in its different child classes, the base class method is a polymorphic method.

## Example

As an example of polymorphism given below, we have **shape** which is an abstract class. It is used as parent by two classes circle and rectangle. Both classes override parent's draw() method in different ways.

&lt;/&gt;

Open Compiler

```
from abc import ABC, abstractmethod
class shape(ABC):
    @abstractmethod
    def draw(self):
        "Abstract method"
        return

class circle(shape):
    def draw(self):
        super().draw()
        print ("Draw a circle")
        return

class rectangle(shape):
    def draw(self):
        super().draw()
        print ("Draw a rectangle")
        return

shapes = [circle(), rectangle()]
for shp in shapes:
    shp.draw()
```

## Output

When you execute this code, it will produce the following output –

```
Draw a circle
Draw a rectangle
```

The variable **shp** first refers to circle object and calls draw() method from circle class. In next iteration, it refers to rectangle object and calls draw() method from rectangle class. Hence draw() method in shape class is polymorphic.

## Python - Method Overriding

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

### Example

&lt;/&gt;

Open Compiler

```
class Parent: # define parent class
    def myMethod(self):
        print ('Calling parent method')

class Child(Parent): # define child class
    def myMethod(self):
        print ('Calling child method')

c = Child() # instance of child
c.myMethod() # child calls overridden method
```

When the above code is executed, it produces the following **output** –

```
Calling child method
```

To understand inheritance in Python, let us take another example. We use following Employee class as parent class –

```
class Employee:
    def __init__(self,nm, sal):
        self.name=nm
        self.salary=sal
    def getName(self):
```

```
    return self.name
def getSalary(self):
    return self.salary
```

Next, we define a SalesOfficer class that uses Employee as parent class. It inherits the instance variables name and salary from the parent. Additionally, the child class has one more instance variable incentive.

We shall use built-in function super() that returns reference of the parent class and call the parent constructor within the child constructor `__init__()` method.

```
class SalesOfficer(Employee):
    def __init__(self,nm, sal, inc):
        super().__init__(nm,sal)
        self.incnt=inc
    def getSalary(self):
        return self.salary+self.incnt
```

The `getSalary()` method is overridden to add the incentive to salary.

## Example

Declare the object of parent and child classes and see the effect of overriding. Complete code is below –

</>

Open Compiler

```
class Employee:
    def __init__(self,nm, sal):
        self.name=nm
        self.salary=sal
    def getName(self):
        return self.name
    def getSalary(self):
        return self.salary

class SalesOfficer(Employee):
    def __init__(self,nm, sal, inc):
        super().__init__(nm,sal)
        self.incnt=inc
    def getSalary(self):
        return self.salary+self.incnt
```

```
e1=Employee("Rajesh", 9000)
print ("Total salary for {} is Rs {}".format(e1.getName(),e1.getSalary()))
s1=SalesOfficer('Kiran', 10000, 1000)
print ("Total salary for {} is Rs {}".format(s1.getName(),s1.getSalary()))
```

When you execute this code, it will produce the following **output** –

```
Total salary for Rajesh is Rs 9000
Total salary for Kiran is Rs 11000
```

## Base Overridable Methods

The following table lists some generic functionality of the object class, which is the parent class for all Python classes. You can override these methods in your own class –

| Sr.No | Method, Description & Sample Call                                                                                      |
|-------|------------------------------------------------------------------------------------------------------------------------|
| 1     | <b>__init__( self [,args...] )</b><br>Constructor (with any optional arguments)<br>Sample Call : obj = className(args) |
| 2     | <b>__del__( self )</b><br>Destructor, deletes an object<br>Sample Call : del obj                                       |
| 3     | <b>__repr__( self )</b><br>Evaluable string representation<br>Sample Call : repr(obj)                                  |
| 4     | <b>__str__( self )</b><br>Printable string representation<br>Sample Call : str(obj)                                    |

## Python - Method Overloading

Method overloading is an important feature of object-oriented programming. Java, C++, C# languages support method overloading, but in Python it is not possible to perform method overloading.

When you have a class with method of one name defined more than one but with different argument types and/or return type, it is a case of method overloading. Python doesn't support this mechanism as the following code shows –

## Example

```
</> Open Compiler

class example:
    def add(self, a, b):
        x = a+b
        return x
    def add(self, a, b, c):
        x = a+b+c
        return x

obj = example()

print (obj.add(10,20,30))
print (obj.add(10,20))
```

## Output

The first call to add() method with three arguments is successful. However, calling add() method with two arguments as defined in the class fails.

60

Traceback (most recent call last):

File "C:\Users\user\example.py", line 12, in <module>  
print (obj.add(10,20))  
^^^^^^^^^^^^^^^^^

TypeError: example.add() missing 1 required positional argument: 'c'

The output tells you that Python considers only the latest definition of add() method, discarding the earlier definitions.

To simulate method overloading, we can use a workaround by defining default value to method arguments as None, so that it can be used with one, two or three arguments.

## Example

```
</> Open Compiler

class example:
    def add(self, a = None, b = None, c = None):
```

```

x=0

if a !=None and b != None and c != None:
    x = a+b+c
elif a !=None and b != None and c == None:
    x = a+b
return x

obj = example()

print (obj.add(10,20,30))
print (obj.add(10,20))

```

It will produce the following **output** –

```

60
30

```

With this workaround, we are able to incorporate method overloading in Python class.

Python's standard library doesn't have any other provision for implementing method overloading. However, we can use dispatch function from a third party module named MultipleDispatch for this purpose.

First, you need to install the Multipledispatch module.

```
pip install multipledispatch
```

This module has a @dispatch decorator. It takes the number of arguments to be passed to the method to be overloaded. Define multiple copies of add() method with @dispatch decorator as below –

## Example

```

from multipledispatch import dispatch
class example:
    @dispatch(int, int)
    def add(self, a, b):
        x = a+b
        return x
    @dispatch(int, int, int)
    def add(self, a, b, c):
        x = a+b+c
        return x

```

```
obj = example()

print (obj.add(10,20,30))
print (obj.add(10,20))
```

## Output

When you execute this code, it will produce the following output –

```
60
30
```

# Python - Dynamic Binding

In object-oriented programming, the concept of dynamic binding is closely related to polymorphism. In Python, dynamic binding is the process of resolving a method or attribute at runtime, instead of at compile time.

According to the polymorphism feature, different objects respond differently to the same method call based on their individual implementations. This behavior is achieved through method overriding, where a subclass provides its own implementation of a method defined in its superclass.

The Python interpreter determines which is the appropriate method or attribute to invoke by based on the object's type or class hierarchy at runtime. This means that the specific method or attribute to be called is determined dynamically, based on the actual type of the object.

## Example

The following example illustrates dynamic binding in Python –

```
</>
```

Open Compiler

```
class shape:
    def draw(self):
        print ("draw method")
        return

class circle(shape):
    def draw(self):
        print ("Draw a circle")
```

```
return\n\nclass rectangle(shape):\n    def draw(self):\n        print ("Draw a rectangle")\n        return\n\nshapes = [circle(), rectangle()]\nfor shp in shapes:\n    shp.draw()
```

It will produce the following **output** –

```
Draw a circle\nDraw a rectangle
```

As you can see, the `draw()` method is bound dynamically to the corresponding implementation based on the object's type. This is how dynamic binding is implemented in Python.

## Duck Typing

Another concept closely related to dynamic binding is **duck typing**. Whether an object is suitable for a particular use is determined by the presence of certain methods or attributes, rather than its type. This allows for greater flexibility and code reuse in Python.

Duck typing is an important feature of dynamic typing languages like Python (Perl, Ruby, PHP, Javascript, etc.) that focuses on an object's behavior rather than its specific type. According to the "duck typing" concept, "If it walks like a duck and quacks like a duck, then it must be a duck."

Duck typing allows objects of different types to be used interchangeably as long as they have the required methods or attributes. The goal is to promote flexibility and code reuse. It is a broader concept that emphasizes on object behavior and interface rather than formal types.

Here is an example of duck typing –

```
</>\n\nclass circle:\n    def draw(self):\n        print ("Draw a circle")
```

Open Compiler

```
return

class rectangle:
    def draw(self):
        print ("Draw a rectangle")
        return

class area:
    def area(self):
        print ("calculate area")
        return

def duck_function(obj):
    obj.draw()

objects = [circle(), rectangle(), area()]
for obj in objects:
    duck_function(obj)
```

It will produce the following **output** –

```
Draw a circle
Draw a rectangle
Traceback (most recent call last):
  File "C:\Python311\hello.py", line 21, in <module>
    duck_function(obj)
  File "C:\Python311\hello.py", line 17, in duck_function
    obj.draw()
AttributeError: 'area' object has no attribute 'draw'
```

The most important idea behind duck typing is that the `duck_function()` doesn't care about the specific types of objects it receives. It only requires the objects to have a `draw()` method. If an object "quacks like a duck" by having the necessary behavior, it is treated as a "duck" for the purpose of invoking the `draw()` method.

Thus, in duck typing, the focus is on the object's behavior rather than its explicit type, allowing different types of objects to be used interchangeably as long as they exhibit the required behavior.

## Python - Dynamic Typing

One of the standout features of Python language is that it is a dynamically typed language. The compiler-based languages C/C++, Java, etc. are statically typed. Let us try to understand the difference between static typing and dynamic typing.

In a statically typed language, each variable and its data type must be declared before assigning it a value. Any other type of value is not acceptable to the compiler, and it raises a compile-time error.

Let us take the following snippet of a Java program –

```
public class MyClass {
    public static void main(String args[]) {
        int var;
        var="Hello";

        System.out.println("Value of var = " + var);
    }
}
```

Here, **var** is declared as an integer variable. When we try to assign it a string value, the compiler gives the following error message –

```
/MyClass.java:4: error: incompatible types: String cannot be converted to int
    x="Hello";
    ^
1 error
```

A variable in Python is only a label, or reference to the object stored in the memory, and not a named memory location. Hence, the prior declaration of type is not needed. Because it's just a label, it can be put on another object, which may be of any type.

In Java, the type of the variable decides what it can store and what not. In Python it is the other way round. Here, the type of data (i.e. object) decides the type of the variable. To begin with, let us store a string in the variable and check its type.

```
>>> var="Hello"
>>> print ("id of var is ", id(var))
id of var is 2822590451184
>>> print ("type of var is ", type(var))
type of var is <class 'str'>
```

So, **var** is of string type. However, it is not permanently bound. It's just a label; and can be assigned to any other type of object, say a float, which will be stored with a different `id()` –

```
>>> var=25.50
>>> print ("id of var is ", id(var))
id of var is 2822589562256
>>> print ("type of var is ", type(var))
type of var is <class 'float'>
```

or a tuple. The var label now sits on a different object.

```
>>> var=(10,20,30)
>>> print ("id of var is ", id(var))
id of var is 2822592028992
>>> print ("type of var is ", type(var))
type of var is <class 'tuple'>
```

We can see that the type of **var** changes every time it refers to a new object. That's why Python is a dynamically typed language.

Dynamic typing feature of Python makes it flexible compared to C/C++ and Java. However, it is prone to runtime errors, so the programmer has to be careful.

## Python - Abstraction

Abstraction is one of the important principles of object-oriented programming. It refers to a programming approach by which only the relevant data about an object is exposed, hiding all the other details. This approach helps in reducing the complexity and increasing the efficiency in application development.

There are two types of abstraction. One is data abstraction, wherein the original data entity is hidden via a data structure that can internally work through the hidden data entities. Other type is called process abstraction. It refers to hiding the underlying implementation details of a process.

In object-oriented programming terminology, a class is said to be an abstract class if it cannot be instantiated, that is you can have an object of an abstract class. You can however use it as a base or parent class for constructing other classes.

To form an abstract class in Python, it must inherit ABC class that is defined in the abc module. This module is available in Python's standard library. Moreover, the class must have at least one abstract method. Again, an abstract method is the one which cannot be called, but can be overridden. You need to decorate it with @abstractmethod decorator.

### Example

```
from abc import ABC, abstractmethod
class demo(ABC):
    @abstractmethod
    def method1(self):
        print ("abstract method")
        return
    def method2(self):
        print ("concrete method")
```

The **demo** class inherits ABC class. There is a method1() which is an abstract method. Note that the class may have other non-abstract (concrete) methods.

If you try to declare an object of demo class, Python raises TypeError –

```
obj = demo()
^^^^^
```

TypeError: Can't instantiate abstract class demo with abstract method method1

The **demo** class here may be used as parent for another class. However, the child class must override the abstract method in parent class. If not, Python throws this error –

TypeError: Can't instantiate abstract class concreteclass with abstract method method1

Hence, the child class with the abstract method overridden is given in the following **example** –

</>

Open Compiler

```
from abc import ABC, abstractmethod
class democlass(ABC):
    @abstractmethod
    def method1(self):
        print ("abstract method")
        return
    def method2(self):
        print ("concrete method")

class concreteclass(democlass):
    def method1(self):
        super().method1()
        return
```

```
obj = concreteclass()
obj.method1()
obj.method2()
```

## Output

When you execute this code, it will produce the following output –

```
abstract method
concrete method
```

# Python - Encapsulation

The principle of Encapsulation is one of the main pillars on which the object-oriented programming paradigm is based. Python takes a different approach towards the implementation of encapsulation.

We know that a class is a user-defined prototype for an object. It defines a set of data members and methods, capable of processing the data. According to principle of data encapsulation, the data members that describe an object are hidden from environment that is external to class. They are available for processing to methods defined within the class only. Methods themselves on the other hand are accessible from outside class context. Hence object data is said to be encapsulated by the methods. The result of such encapsulation is that any unwarranted access to the object data is prevented.

Languages such as C++ and Java use access modifiers to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only. Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures the implementation of encapsulation.

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a Python class are public, as is demonstrated by the following example.

## Example 1

Here, we have an Employee class with instance variables, **name** and **age**. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

&lt;/&gt;

[Open Compiler](#)

```
class Student:
    def __init__(self, name="Rajaram", marks=50):
        self.name = name
        self.marks = marks

    s1 = Student()
    s2 = Student("Bharat", 25)

    print ("Name: {} marks: {}".format(s1.name, s2.marks))
    print ("Name: {} marks: {}".format(s2.name, s2.marks))
```

It will produce the following **output** –

```
Name: Rajaram marks: 50
Name: Bharat marks: 25
```

In the above example, the instance variables are initialized inside the class. However, there is no restriction on accessing the value of instance variable from outside the class, which is against the principle of encapsulation.

Although there are no keywords to enforce visibility, Python has a convention of naming the instance variables in a peculiar way. In Python, prefixing name of variable/method with single or double underscore to emulate behavior of protected and private access modifiers.

If a variable is prefixed by a single double underscore (such as "age"), the instance variable is private, similarly if a variable name is prefixed it with single underscore (such as "\_salary")

## Example 2

Let us modify the Student class. Add another instance variable salary. Make name private and **marks** as private by prefixing double underscores to them.

&lt;/&gt;

[Open Compiler](#)

```
class Student:
    def __init__(self, name="Rajaram", marks=50):
        self.__name = name
        self.__marks = marks
```

```
def studentdata(self):
    print ("Name: {} marks: {}".format(self.__name, self.__marks))

s1 = Student()
s2 = Student("Bharat", 25)

s1.studentdata()
s2.studentdata()
print ("Name: {} marks: {}".format(s1.__name, s2.__marks))
print ("Name: {} marks: {}".format(s2.__name, s2.marks))
```

When you run this code, it will produce the following **output** –

```
Name: Rajaram marks: 50
Name: Bharat marks: 25
Traceback (most recent call last):
  File "C:\Python311\hello.py", line 14, in <module>
    print ("Name: {} marks: {}".format(s1.__name, s2.__marks))
AttributeError: 'Student' object has no attribute '__name'
```

The above output makes it clear that the instance variables name and age, although they can be accessed by a method declared inside the class (the studentdata() method), but since the double underscores prefix makes the variables private, and hence accessing them outside the class is disallowed, raising Attribute error.

Python doesn't block access to private data entirely. It just leaves it for the wisdom of the programmer, not to write any code that access it from outside the class. You can still access the private members by Python's name mangling technique.

Name mangling is the process of changing name of a member with double underscore to the form **object.\_class\_\_variable**. If so required, it can still be accessed from outside the class, but the practice should be refrained.

In our example, the private instance variable "\_\_name" is mangled by changing it to the format

```
obj._class__privatevar
```

So, to access the value of "\_\_marks" instance variable of "s1" object, change it to "s1.\_Student\_\_marks".

Change the print() statement in the above program to –

```
print (s1._Student__marks)
```

It now prints 50, the marks of s1.

Hence, we can conclude that Python doesn't implement encapsulation exactly as per the theory of object oriented programming. It adapts a more mature approach towards it by prescribing a name convention, and letting the programmer to use name mangling if it is really required to have access to private data in the public scope.

## Python - Interfaces

In software engineering, an interface is a software architectural pattern. An interface is like a class but its methods just have prototype signature definition without any body to implement. The recommended functionality needs to be implemented by a concrete class.

In languages like Java, there is interface keyword which makes it easy to define an interface. Python doesn't have it or any similar keyword. Hence the same ABC class and @abstractmethod decorator is used as done in an abstract class.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

### Example

```
from abc import ABC, abstractmethod
class demoInterface(ABC):
    @abstractmethod
    def method1(self):
        print ("Abstract method1")
        return

    @abstractmethod
    def method2(self):
        print ("Abstract method1")
        return
```

The above interface has two abstract methods. As in abstract class, we cannot instantiate an interface.

```
obj = demoInterface()
^^^^^^^^^^^^^^^^^
```

`TypeError: Can't instantiate abstract class demoInterface with abstract methods method1`

Let us provide a class that implements both the abstract methods. If doesn't contain implementations of all abstract methods, Python shows following error –

```
obj = concreteclass()
^^^^^^^^^^^^^^^^^
```

`TypeError: Can't instantiate abstract class concreteclass with abstract method method2`

The following class implements both methods –

```
class concreteclass(demoInterface):
    def method1(self):
        print ("This is method1")
        return

    def method2(self):
        print ("This is method2")
        return

obj = concreteclass()
obj.method1()
obj.method2()
```

## Output

When you execute this code, it will produce the following output –

```
This is method1
This is method2
```

## Python - Packages

In Python, module is a Python script with .py extension and contains objects such as classes, functions etc. Packages in Python extend the concept of modular approach further. Package is a folder containing one or more module files; additionally a special file "`__init__.py`" file which may be empty but may contain the package list.

Let us create a Python package with the name mypackage. Follow the steps given below –

- Create an outer folder to hold the contents of mypackage. Let its name be packagedemo.
- Inside it, create another folder mypackage. This will be the Python package we are going to construct. Two Python modules areafunctions.py and mathfunctions.py will be created inside mypackage.
- Create an empty "\_\_init\_\_.py" file inside mypackage folder.
- Inside the outer folder, we shall later on store a Python script example.py to test our package.

The **file/folder structure** should be as shown below –

## PackageDemo

- example.py
- testpackage.py
- setup.py
- mypackage
  - \_\_init\_\_.py
  - areafunctions.py
  - mathfunctions.py

Using your favorite code editor, save the following two Python modules in mypackage folder.

```
# mathfunctions.py
def sum(x,y):
    val = x+y
    return val

def average(x,y):
    val = (x+y)/2
    return val

def power(x,y):
    val = x**y
    return val
```

Create another Python script –

```
# areafunctions.py
def rectangle(w,h):
    area = w*h
    return area

def circle(r):
    import math
    area = math.pi*math.pow(r,2)
    return area
```

Let us now test the **myexample** package with the help of a Python script above this package folder. Refer to the folder structure above.

```
#example.py
from mypackage.areafunctions import rectangle
print ("Area :", rectangle(10,20))

from mypackage.mathsfunctions import average
print ("average:", average(10,20))
```

This program imports functions from **mypackage**. If the above script is executed, you should get following **output** –

```
Area : 200
average: 15.0
```

## Define Package List

You can put selected functions or any other resources from the package in the "`__init__.py`" file. Let us put the following code in it.

```
from .areafunctions import circle
from .mathsfunctions import sum, power
```

To import the available functions from this package, save the following script as `testpackage.py`, above the package folder as before.

```
#testpackage.py
from mypackage import power, circle
```

```
print ("Area of circle:", circle(5))
print ("10 raised to 2:", power(10,2))
```

It will produce the following **output** –

```
Area of circle: 78.53981633974483
10 raised to 2: 100
```

## Package Installation

Right now, we are able to access the package resources from a script just above the package folder. To be able to use the package anywhere in the file system, you need to install it using the PIP utility.

First of all, save the following script in the parent folder, at the level of package folder.

```
#setup.py
from setuptools import setup
setup(name='mypackage',
version='0.1',
description='Package setup script',
url='#',
author='anonymous',
author_email='test@gmail.com',
license='MIT',
packages=['mypackage'],
zip_safe=False)
```

Run the PIP utility from command prompt, while remaining in the parent folder.

```
C:\Users\user\packagedemo>pip3 install .
Processing c:\users\user\packagedemo
Preparing metadata (setup.py) ... done
Installing collected packages: mypackage
  Running setup.py install for mypackage ... done
Successfully installed mypackage-0.1
```

You should now be able to import the contents of the package in any environment.

```
C:\Users>python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)]
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import mypackage  
>>> mypackage.circle(5)  
78.53981633974483
```

# Python - Inner Classes

A class defined inside another class is known as an inner class in Python. Sometimes inner class is also called nested class. If the inner class is instantiated, the object of inner class can also be used by the parent class. Object of inner class becomes one of the attributes of the outer class. Inner class automatically inherits the attributes of the outer class without formally establishing inheritance.

## Syntax

```
class outer:  
    def __init__(self):  
        pass  
    class inner:  
        def __init__(self):  
            pass
```

An inner class lets you group classes. One of the advantages of nesting classes is that it becomes easy to understand which classes are related. The inner class has a local scope. It acts as one of the attributes of the outer class.

## Example

In the following code, we have student as the outer class and subjects as the inner class. The `__init__()` constructor of student initializes name attribute and an instance of subjects class. On the other hand, the constructor of inner subjects class initializes two instance variables sub1, sub2.

A `show()` method of outer class calls the method of inner class with the object that has been instantiated.

```
class student:  
    def __init__(self):  
        self.name = "Ashish"  
        self.subs = self.subjects()  
    def show(self):  
        print ("Name:", self.name)
```

```

        self.subs.display()

class subjects:
    def __init__(self):
        self.sub1 = "Phy"
        self.sub2 = "Che"
    return
    def display(self):
        print ("Subjects:",self.sub1, self.sub2)

s1 = student()
s1.show()

```

When you execute this code, it will produce the following **output** –

```

Name: Ashish
Subjects: Phy Che

```

It is quite possible to declare an object of outer class independently, and make it call its own display() method.

```
sub = student().subjects().display()
```

It will list out the subjects.

## Python - Anonymous Class and Objects

Python's built-in type() function returns the class that an object belongs to. In Python, a class, both a built-in class or a user-defined class are objects of type class.

### Example

</>

[Open Compiler](#)

```

class myclass:
    def __init__(self):
        self.myvar=10
    return

obj = myclass()

print ('class of int', type(int))

```

```
print ('class of list', type(list))
print ('class of dict', type(dict))
print ('class of myclass', type(myclass))
print ('class of obj', type(obj))
```

It will produce the following **output** –

```
class of int <class 'type'>
class of list <class 'type'>
class of dict <class 'type'>
class of myclass <class 'type'>
```

The `type()` has a three argument version as follows –

## Syntax

```
newclass=type(name, bases, dict)
```

Using above syntax, a class can be dynamically created. Three arguments of `type` function are –

- **name** – name of the class which becomes `__name__` attribute of new class
- **bases** – tuple consisting of parent classes. Can be blank if not a derived class
- **dict** – dictionary forming namespace of the new class containing attributes and methods and their values.

We can create an anonymous class with the above version of `type()` function. The name argument is a null string, second argument is a tuple of one class the object class (note that each class in Python is inherited from object class). We add certain instance variables as the third argument dictionary. We keep it empty for now.

```
anon=type(' ', (object, ), {})
```

To create an object of this anonymous class –

```
obj = anon()
print ("type of obj:", type(obj))
```

The result shows that the object is of anonymous class

```
type of obj: <class '__main__.'>
```

## Example

We can also add instance variables and instance methods dynamically. Take a look at this example –

```
</>
```

[Open Compiler](#)

```
def getA(self):
    return self.a
obj = type('',(object,),{'a':5,'b':6,'c':7,'getA':getA,'getB':lambda self : self.b})
print (obj.getA(), obj.getB())
```

It will produce the following **output** –

```
5 6
```

## Python - Singleton Class

A Singleton class is a class of which only one object can be created. This helps in optimizing memory usage when you perform some heavy operation, like creating a database connection.

## Example

```
</>
```

[Open Compiler](#)

```
class SingletonClass:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print('Creating the object')
            cls._instance = super(SingletonClass, cls).__new__(cls)
        return cls._instance

obj1 = SingletonClass()
print(obj1)
```

```
obj2 = SingletonClass()  
print(obj2)
```

This is how the above code works –

When an instance of a Python class declared, it internally calls the `__new__()` method. We override the `__new__()` method that is called internally by Python when you create an object of a class. It checks whether our instance variable is `None`. If the instance variable is `None`, it creates a new object and call the `super()` method and returns the instance variable that contains the object of this class.

If multiple objects are created, it becomes clear that the object is only created the first time; after that, the same object instance is returned.

Creating the object

```
<__main__.SingletonClass object at 0x000002A5293A6B50>  
<__main__.SingletonClass object at 0x000002A5293A6B50>
```

## Python - Wrapper Classes

A function in Python is a first-order object. A function can have another function as its argument and wrap another function definition inside it. This helps in modifying a function without actually changing it. Such functions are called decorators.

This feature is also available for wrapping a class. This technique is used to manage the class after it is instantiated by wrapping its logic inside a decorator.

### Example

</>

Open Compiler

```
def decorator_function(Wrapped):  
    class Wrapper:  
        def __init__(self,x):  
            self.wrap = Wrapped(x)  
        def print_name(self):  
            return self.wrap.name  
    return Wrapper  
  
@decorator_function  
class Wrapped:
```

```
def __init__(self,x):
    self.name = x

obj = Wrapped('TutorialsPoint')
print(obj.print_name())
```

Here, **Wrapped** is the name of the class to be wrapped. It is passed as argument to a function. Inside the function, we have a Wrapper class, modify its behavior with the attributes of the passed class, and return the modified class. The returned class is instantiated and its method can now be called.

When you execute this code, it will produce the following **output** –

```
TutorialsPoint
```

## Python - Enums

The term 'enumeration' refers to the process of assigning fixed constant values to a set of strings, so that each string can be identified by the value bound to it. Python's standard library offers the **enum** module. The **Enum** class included in **enum** module is used as the parent class to define enumeration of a set of identifiers – conventionally written in upper case.

### Example 1

```
from enum import Enum

class subjects(Enum):
    ENGLISH = 1
    MATHS = 2
    SCIENCE = 3
    SANSKRIT = 4
```

In the above code, "subjects" is the enumeration. It has different enumeration members, e.g., `subjects.MATHS`. Each member is assigned a value.

Each member is an object of the enumeration class `subjects`, and has name and value attributes.

```
obj = subjects.MATHS
print(type(obj), obj.value)
```

It results in following **output** –

```
<enum 'subjects'> 2
```

## Example 2

Value bound to the enum member needn't always be an integer, it can be a string as well. See the following example –

</>

Open Compiler

```
from enum import Enum

class subjects(Enum):
    ENGLISH = "E"
    MATHS = "M"
    GEOGRAPHY = "G"
    SANSKRIT = "S"

obj = subjects.SANSKRIT
print (type(obj), obj.name, obj.value)
```

It will produce the following **output** –

```
<enum 'subjects'> SANSKRIT S
```

## Example 3

You can iterate through the enum members in the order of their appearance in the definition, with the help of a **for** loop –

```
for sub in subjects:
    print (sub.name, sub.value)
```

It will produce the following **output** –

```
ENGLISH E
MATHS M
GEOGRAPHY G
SANSKRIT S
```

The enum member can be accessed with the unique value assigned to it, or by its name attribute. Hence, `subjects("E")` as well as `subjects["ENGLISH"]` returns `subjects.ENGLISH` member.

## Example 4

An enum class cannot have same member appearing twice, however, more than one members may be assigned same value. To ensure that each member has a unique value bound to it, use the `@unique` decorator.

&lt;/&gt;

Open Compiler

```
from enum import Enum, unique

@unique
class subjects(Enum):
    ENGLISH = 1
    MATHS = 2
    GEOGRAPHY = 3
    SANSKRIT = 2
```

This will raise an exception as follows –

```
@unique
^^^^^^^
raise ValueError('duplicate values found in %r: %s' %
ValueError: duplicate values found in <enum 'subjects'>: SANSKRIT -> MATHS
```

The `Enum` class is a callable class, hence you can use the following alternative method of defining enumeration –

```
from enum import Enum
subjects = Enum("subjects", "ENGLISH MATHS SCIENCE SANSKRIT")
```

The `Enum` constructor uses two arguments here. First one is the name of enumeration. Second argument is a string consisting of enumeration member symbolic names, separated by a whitespace.

## Python - Reflection

In object-oriented programming, reflection refers to the ability to extract information about any object in use. You can get to know the type of object, is it a subclass of any

other class, what are its attributes and much more. Python's standard library has a number of functions that reflect on different properties of an object. Reflection is also sometimes called introspect.

Let us take a review of reflection functions.

## The type() Function

We have used this function many times. It tells you which class does an object belong to.

### Example

Following statements print the respective class of different built-in data type objects

```
</> Open Compiler  
print (type(10))  
print (type(2.56))  
print (type(2+3j))  
print (type("Hello World"))  
print (type([1,2,3]))  
print (type({1:'one', 2:'two'}))
```

Here, you will get the following **output** –

```
<class 'int'>  
<class 'float'>  
<class 'complex'>  
<class 'str'>  
<class 'list'>  
<class 'dict'>
```

Let us verify the type of an object of a user-defined class –

```
</> Open Compiler  
class test:  
    pass  
  
obj = test()  
print (type(obj))
```

It will produce the following **output** –

```
<class '__main__.test'>
```

## The `isinstance()` Function

This is another built-in function in Python which ascertains if an object is an instance of the given class

### Syntax

```
isinstance(obj, class)
```

This function always returns a Boolean value, true if the object is indeed belongs to the given class and false if not.

### Example

Following statements return True –

```
</> Open Compiler  
print (isinstance(10, int))  
print (isinstance(2.56, float))  
print (isinstance(2+3j, complex))  
print (isinstance("Hello World", str))
```

In contrast, these statements print False.

```
</> Open Compiler  
print (isinstance([1,2,3], tuple))  
print (isinstance({1:'one', 2:'two'}, set))
```

It will produce the following **output** –

```
True  
True  
True  
True
```

```
False
```

```
False
```

You can also perform check with a user defined class

```
</> Open Compiler  
  
class test:  
    pass  
  
obj = test()  
print (isinstance(obj, test))
```

It will produce the following **output** –

```
True
```

In Python, even the classes are objects. All classes are objects of object class. It can be verified by following code –

```
</> Open Compiler  
  
class test:  
    pass  
  
print (isinstance(int, object))  
print (isinstance(str, object))  
print (isinstance(test, object))
```

All the above print statements print True.

## The issubclass() Function

This function checks whether a class is a subclass of another class. Pertains to classes, not their instances.

As mentioned earlier, all Python classes are subclassed from object class. Hence, output of following print statements is True for all.

```
</>
```

Open Compiler

```
class test:  
    pass  
  
print (issubclass(int, object))  
print (issubclass(str, object))  
print (issubclass(test, object))
```

It will produce the following **output** –

True  
True  
True

## The callable() Function

An object is callable if it invokes a certain process. A Python function, which performs a certain process, is a callable object. Hence `callable(function)` returns True. Any function, built-in, user defined or a method is callable. Objects of built-in data types such as int, str, etc., are not callable.

### Example

```
</> Open Compiler  
  
def test():  
    pass  
  
print (callable("Hello"))  
print (callable(abs))  
print (callable(list.clear([1,2])))  
print (callable(test))
```

A string object is not callable. But `abs` is a function which is callable. The `pop` method of `list` is callable, but `clear()` is actually call to the function and not a function object, hence not a callable

It will produce the following **output** –

False  
True

True

False

True

A class instance is callable if it has a `__call__()` method. In the example below, the test class includes `__call__()` method. Hence, its object can be used as if we are calling function. Hence, object of a class with `__call__()` function is a callable.

</>

Open Compiler

```
class test:  
    def __init__(self):  
        pass  
    def __call__(self):  
        print ("Hello")  
  
obj = test()  
obj()  
print ("obj is callable?", callable(obj))
```

It will produce the following **output** –

Hello

obj is callable? True

## The getattr() Function

The `getattr()` built-in function retrieves the value of the named attribute of object.

### Example

</>

Open Compiler

```
class test:  
    def __init__(self):  
        self.name = "Manav"  
  
obj = test()  
print (getattr(obj, "name"))
```

It will produce the following **output** –

Manav

## The setattr() Function

The setattr() built-in function adds a new attribute to the object and assigns it a value. It can also change the value of an existing attribute.

In the example below, the object of test class has a single attribute – name. We use setattr to add age attribute and to modify the value of name attribute.

</>

Open Compiler

```
class test:  
    def __init__(self):  
        self.name = "Manav"  
  
obj = test()  
setattr(obj, "age", 20)  
setattr(obj, "name", "Madhav")  
print (obj.name, obj.age)
```

It will produce the following **output** –

Madhav 20

## The hasattr() Function

This built-in function returns True if the given attribute is available to the object argument, and false if not. We use the same test class and check if it has a certain attribute or not.

</>

Open Compiler

```
class test:  
    def __init__(self):  
        self.name = "Manav"  
  
obj = test()
```

```
print (hasattr(obj, "age"))
print (hasattr(obj, "name"))
```

It will produce the following **output** –

False

True

## The dir() Function

If this built-in function called without an argument, return the names in the current scope. For any object as argument, it returns a list of the attributes of the given object, and of attributes reachable from it.

- **For a module object** – the function returns the module's attributes.
- **For a class object** – the function returns its attributes, and recursively the attributes of its bases.
- **For any other object** – its attributes, its class's attributes, and recursively the attributes of its class's base classes.

## Example

</>

Open Compiler

```
print ("dir(int):", dir(int))
```

It will produce the following **output** –

```
dir(int): ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
           '__dir__', '__eq__', '__ge__', '__getattribute__', '__hash__', '__le__', '__lt__', '__ne__',
           '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__radd__', '__rsub__', '__setattr__',
           '__str__', '__truediv__', '__floordiv__', '__mod__', '__mul__', '__pow__', '__rmod__',
           '__rpow__', '__rfloordiv__', '__rmod__', '__rpow__']
```

## Example

</>

Open Compiler

```
print ("dir(dict):", dir(dict))
```

It will produce the following **output** –

```
dir(dict):['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

## Example

&lt;/&gt;

Open Compiler

```
class test:  
    def __init__(self):  
        self.name = "Manav"  
  
obj = test()  
print ("dir(obj):", dir(obj))
```

It will produce the following **output** –

```
dir(obj):['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

## Python - Syntax Errors

Generally, three types of errors appear in a computer program: Syntax errors, logical errors and runtime errors. Syntax errors are the most common type of errors one faces while writing a program, whether you are new to programming or an experienced programmer. Syntax errors are basically related to the rules of grammar of a certain language.

Syntax errors occur whenever the rules laid down by the language are not followed. In Python, there are well defined rules for giving name to an identifier, that is, a variable, a function, a class, a module or any Python object. Similarly, Python keywords should be used as per the syntax defined. Whenever these rules are not followed, Python interpreter displays a syntax error message.

A simple example of declaring a variable in Python interactive shell is given below.

```
>>> name="Python  
File "<stdin>", line 1  
      name="Python  
      ^  
SyntaxError: unterminated string literal (detected at line 1)
```

Python interpreter displays syntax error along with a certain explanatory message. In the above example, because the quotation symbol is not closed, the Syntax error occurs.

Similarly, Python requires each function name should be followed by parentheses inside which the function arguments should be given.

In the following example, we get a syntax error –

```
>>> print "Hello"
```

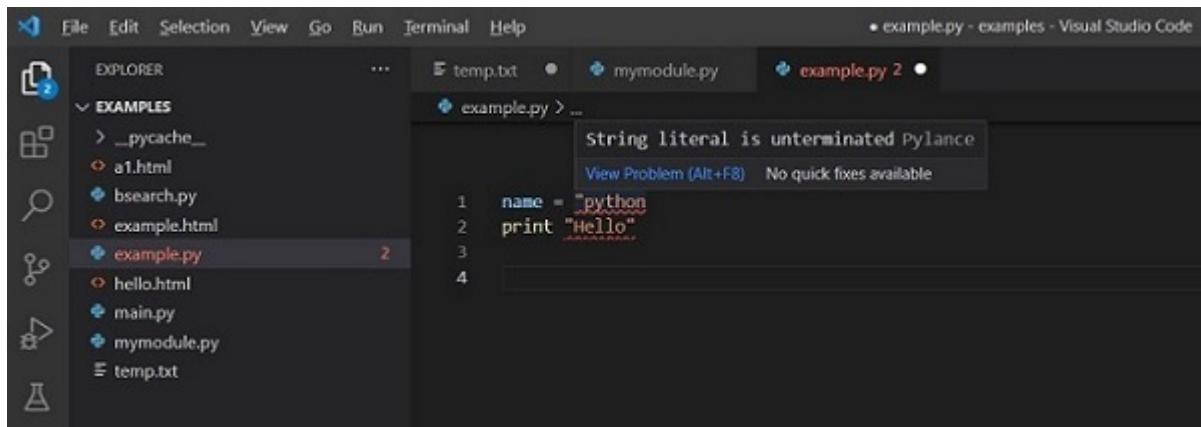
```
  File "<stdin>", line 1
    print "Hello"
    ^^^^^^^^^^^^^^
```

```
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?
```

The reason can be understood from the error message, that the `print()` function is missing parentheses.

There are many popular IDEs for Python programming. Most of them use colorized syntax highlighting, which makes it easy to visually identify the error.

One such IDE is **VS Code**. While entering an instruction, the syntax errors are suitably highlighted.



The error is highlighted. If you put the cursor there, VS Code tells more about the error. If you still go ahead and execute the code, error messages appear in the command terminal.

Syntax errors are easy to identify and rectify. The IDE such as VS Code makes it easy. However, sometimes, your code doesn't show any syntax errors, but still the output of the program is not what you anticipate. Such errors are logical errors. They are hard to detect, as the error lies in the logic used in the code. You learn by experience how to correct logical errors. VS Code and other IDEs have features such as watches and breakpoints to trap these errors.

Third type of error is a runtime error also called exception. There is no syntax error nor there is any logical error in your program. Most of the times, the program gives desired

output, but in some specific situations you get abnormal behaviour of the program, such as the program abnormally terminates or gives some absurd result.

The factors causing exceptions are generally external to the program. For example incorrect input, type conversion or malfunction IO device etc.

## What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Python's standard library defines standard exception classes. As with other Python classes, Exceptions are also subclasses of Object class. Following is the object hierarchy of Python's Exceptions.

```
object
  BaseException
    Exception
      ArithmeticError
        FloatingPointError
        OverflowError
        ZeroDivisionError
      AssertionError
      AttributeError
      BufferError
      EOFError
      ImportError
        ModuleNotFoundError
      LookupError
        IndexError
        KeyError
      MemoryError
      NameError
      OSError
      ReferenceError
      RuntimeError
      StopAsyncIteration
```

StopIteration  
SyntaxError

# Python - Exceptions Handling

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the **try:** block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

- The **try:** block contains statements which are susceptible for exception
- If exception occurs, the program jumps to the **except:** block.
- If no exception in the **try:** block, the **except:** block is skipped.

## Syntax

Here is the simple syntax of **try...except...else** blocks –

```
try:  
    You do your operations here  
    .....  
except ExceptionI:  
    If there is ExceptionI, then execute this block.  
except ExceptionII:  
    If there is ExceptionII, then execute this block.  
    .....  
else:  
    If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax –

- A single **try** statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic **except** clause, which handles any exception.
- After the except clause(s), you can include an **else** clause. The code in the **else** block executes if the code in the try: block does not raise an exception.
- The **else** block is a good place for code that does not need the try: block's protection.

## Example

This example opens a file, writes content in the file and comes out gracefully because there is no problem at all.

&lt;/&gt;

Open Compiler

```
try:  
    fh = open("testfile", "w")  
    fh.write("This is my test file for exception handling!!")  
except IOError:  
    print ("Error: can't find file or read data")  
else:  
    print ("Written content in the file successfully")  
    fh.close()
```

It will produce the following **output** –

Written content in the file successfully

However, change the mode parameter in open() function to "w". If the testfile is not already present, the program encounters IOError in except block, and prints following error message –

Error: can't find file or read data

## Python - The try-except Block

You can also use the **except** statement with no exceptions defined as follows –

```
try:  
    You do your operations here  
    .....  
except:  
    If there is any exception, then execute this block.  
    .....  
else:  
    If there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

You can also use the same except statement to handle multiple exceptions as follows –

```
try:
    You do your operations here
    .....
except(Exception1[, Exception2[,...ExceptionN]]):
    If there is any exception from the given exception list,
    then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

## Python - The try-finally Block

You can use a **finally:** block along with a **try:** block. The **finally:** block is a place to put any code that must execute, whether the try-block raised an exception or not.

The syntax of the **try-finally** statement is this –

```
try:
    You do your operations here;
    .....
    Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

**Note** – You can provide except clause(s), or a finally clause, but not both. You cannot use else clause as well along with a finally clause.

### Example

</>

Open Compiler

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
finally:
```

```
print ("Error: can't find file or read data")
fh.close()
```

If you do not have permission to open the file in writing mode, then it will produce the following **output** –

```
Error: can't find file or read data
```

The same example can be written more cleanly as follows –

```
</> Open Compiler

try:
    fh = open("testfile", "w")
    try:
        fh.write("This is my test file for exception handling!!")
    finally:
        print ("Going to close the file")
        fh.close()
except IOError:
    print ("Error: can't find file or read data")
```

When an exception is thrown in the try block, the execution immediately passes to the **finally** block. After all the statements in the **finally** block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

## Exception with Arguments

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
    You do your operations here
    .....
except ExceptionType as Argument:
    You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions,

you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

## Example

Following is an example for a single exception –

```
</> Open Compiler  
  
# Define a function here.  
def temp_convert(var):  
    try:  
        return int(var)  
    except ValueError as Argument:  
        print("The argument does not contain numbers\n",Argument)  
# Call above function here.  
temp_convert("xyz")
```

It will produce the following **output** –

```
The argument does not contain numbers  
invalid literal for int() with base 10: 'xyz'
```

# Python - Raising Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows –

## Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

## Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
    if level <1:
        raise Exception(level)
        # The code below to this would not be executed
        # if we raise the exception
    return level
```

**Note** – In order to catch an exception, an "except" clause must refer to the same exception thrown either as a class object or a simple string. For example, to capture the above exception, we must write the except clause as follows –

```
try:
    Business Logic here...
except Exception as e:
    Exception handling here using e.args...
else:
    Rest of the code here...
```

The following example illustrates the use of raising an exception –

</>
Open Compiler

```
def functionName( level ):
    if level <1:
        raise Exception(level)
        # The code below to this would not be executed
        # if we raise the exception
    return level

try:
    l=functionName(-10)
    print ("level=",l)
except Exception as e:
    print ("error in level argument",e.args[0])
```

This will produce the following **output** –

error in level argument -10

## Python - Exception Chaining

Exception chaining is a technique of handling exceptions by re-throwing a caught exception after wrapping it inside a new exception. The original exception is saved as a property (such as cause) of the new exception.

During the handling of one exception 'A', it is possible that another exception 'B' may occur. It is useful to know about both exceptions in order to debug the problem. Sometimes it is useful for an exception handler to deliberately re-raise an exception, either to provide extra information or to translate an exception to another type.

In Python 3.x, it is possible to implement exception chaining. If there is any unhandled exception inside an except section, it will have the exception being handled attached to it and included in the error message.

### Example

In the following code snippet, trying to open a non-existent file raises FileNotFoundError. It is detected by the except block. While handling another exception is raised.

```
</> Open Compiler  
  
try:  
    open("nofile.txt")  
except OSError:  
    raise RuntimeError("unable to handle error")
```

It will produce the following **output** –

```
Traceback (most recent call last):  
  File "/home/cg/root/64afc39c651/main.py", line 2, in <module>  
    open("nofile.txt")  
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):  
  File "/home/cg/root/64afc39c651/main.py", line 4, in <module>
```

```
raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

## raise .. from

If you use an optional `from` clause in the `raise` statement, it indicates that an exception is a direct consequence of another. This can be useful when you are transforming exceptions. The token after `from` keyword should be the exception object.

```
</> Open Compiler

try:
    open("nofile.txt")
except OSError as exc:
    raise RuntimeError from exc
```

It will produce the following **output** –

```
Traceback (most recent call last):
  File "/home/cg/root/64afcadc39c651/main.py", line 2, in <module>
    open("nofile.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "/home/cg/root/64afcadc39c651/main.py", line 4, in <module>
    raise RuntimeError from exc
RuntimeError
```

## raise .. from None

If we use `None` in `from` clause instead of exception object, the automatic exception chaining that was found in the earlier example is disabled.

```
</> Open Compiler

try:
    open("nofile.txt")
```

```
except OSError as exc:
    raise RuntimeError from None
```

It will produce the following **output** –

```
Traceback (most recent call last):
File "C:\Python311\hello.py", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

## \_\_context\_\_ and \_\_cause\_\_

Raising an exception in the except block will automatically add the captured exception to the `__context__` attribute of the new exception. Similarly, you can also add `__cause__` to any exception using the expression `raise ... from` syntax.

&lt;/&gt;

Open Compiler

```
try:
    try:
        raise ValueError("ValueError")
    except ValueError as e1:
        raise TypeError("TypeError") from e1
except TypeError as e2:
    print("The exception was", repr(e2))
    print("Its __context__ was", repr(e2.__context__))
    print("Its __cause__ was", repr(e2.__cause__))
```

It will produce the following **output** –

```
The exception was TypeError('TypeError')
Its __context__ was ValueError('ValueError')
Its __cause__ was ValueError('ValueError')
```

## Python - Nested try Block

In a Python program, if there is another **try-except** construct either inside either a **try** block or inside its **except** block, it is known as a nested-try block. This is needed when

different blocks like outer and inner may cause different errors. To handle them, we need nested try blocks.

We start with an example having a single "try – except – finally" construct. If the statements inside try encounter exception, it is handled by except block. With or without exception occurred, the finally block is always executed.

## Example 1

Here, the **try** block has "division by 0" situation, hence the **except** block comes into play. It is equipped to handle the generic exception with Exception class.

```
</> Open Compiler  
  
a=10  
b=0  
try:  
    print (a/b)  
except Exception:  
    print ("General Exception")  
finally:  
    print ("inside outer finally block")
```

It will produce the following **output** –

```
General Exception  
inside outer finally block
```

## Example 2

Let us now see how to nest the **try** constructs. We put another "try – except – finally" blocks inside the existing try block. The except keyword for inner try now handles generic Exception, while we ask the except block of outer try to handle ZeroDivisionError.

Since exception doesn't occur in the inner **try** block, its corresponding generic Except isn't called. The division by 0 situation is handled by outer except clause.

```
</>
```

Open Compiler

```
a=10  
b=0  
try:
```

```
print (a/b)
try:
    print ("This is inner try block")
except Exception:
    print ("General exception")
finally:
    print ("inside inner finally block")

except ZeroDivisionError:
    print ("Division by 0")
finally:
    print ("inside outer finally block")
```

It will produce the following **output** –

```
Division by 0
inside outer finally block
```

### Example 3

Now we reverse the situation. Out of the nested **try** blocks, the outer one doesn't have any exception raised, but the statement causing division by 0 is inside inner try, and hence the exception handled by inner **except** block. Obviously, the **except** part corresponding to outer **try**: will not be called upon.

```
</> Open Compiler

a=10
b=0
try:
    print ("This is outer try block")
    try:
        print (a/b)
    except ZeroDivisionError:
        print ("Division by 0")
    finally:
        print ("inside inner finally block")

except Exception:
    print ("General Exception")
```

```
finally:  
    print ("inside outer finally block")
```

It will produce the following **output** –

```
This is outer try block  
Division by 0  
inside inner finally block  
inside outer finally block
```

In the end, let us discuss another situation which may occur in case of nested blocks. While there isn't any exception in the outer **try:**, there isn't a suitable except block to handle the one inside the inner **try:** block.

## Example 4

In the following example, the inner **try:** faces "division by 0", but its corresponding **except:** is looking for **KeyError** instead of **ZeroDivisionError**. Hence, the exception object is passed on to the **except:** block of the subsequent except statement matching with outer **try:** statement. There, the **zeroDivisionError** exception is trapped and handled.

```
</> Open Compiler  
  
a=10  
b=0  
try:  
    print ("This is outer try block")  
    try:  
        print (a/b)  
    except KeyError:  
        print ("Key Error")  
    finally:  
        print ("inside inner finally block")  
  
except ZeroDivisionError:  
    print ("Division by 0")  
finally:  
    print ("inside outer finally block")
```

It will produce the following **output** –

This is outer try block  
inside inner finally block  
Division by 0  
inside outer finally block

## Python - User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example that has a user-defined MyException class. Here, a class is created that is subclassed from base Exception class. This is useful when you need to display more specific information when an exception is caught.

In the **try** block, the user-defined exception is raised whenever value of num variable is less than 0 or more than 100 and caught in the **except** block. The variable e is used to create an instance of the class MyException.

### Example

</>

Open Compiler

```
class MyException(Exception):
    "Invalid marks"
    pass

num = 10
try:
    if num <0 or num>100:
        raise MyException
except MyException as e:
    print ("Invalid marks:", num)
else:
    print ("Marks obtained:", num)
```

### Output

For different values of **num**, the program shows the following **output** –

Marks obtained: 10  
Invalid marks: 104

Invalid marks: -10

# Python - Logging

The term "logging" refers to the mechanism of recording different intermediate events in a certain process. Recording logs in a software application proves helpful for the developer in debugging and tracing any errors in the application logic. Python's standard library includes logging module with which application logs can be generated and recorded.

It is a normal practice to use `print()` statements intermittently in a program to check intermediate values of different variables and objects. It helps the developer to verify if the program is behaving as per expectation or not. However, logging is more beneficial than the intermittent print statements as it gives more insight into the events.

## Logging Levels

One of the important features of logging is that you can generate log message of different severity levels. The logging module defines following levels with their values.

| Level    | When it's used                                                                                                                                                         | Value |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| DEBUG    | Detailed information, typically of interest only when diagnosing problems.                                                                                             | 10    |
| INFO     | Confirmation that things are working as expected.                                                                                                                      | 20    |
| WARNING  | An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected. | 30    |
| ERROR    | Due to a more serious problem, the software has not been able to perform some function.                                                                                | 40    |
| CRITICAL | A serious error, indicating that the program itself may be unable to continue running.                                                                                 | 50    |

## Example

The following code illustrates how to generate logging messages.

&lt;/&gt;

Open Compiler

```
import logging
```

```
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

It will produce the following **output** –

WARNING:root:This is a warning message

ERROR:root:This is an error message

CRITICAL:root:This is a critical message

Note that only the log messages after the WARNING level are displayed here. That is because root - the default logger ignores all severity levels above WARNING severity level. Notice severity level is logged before the first colons (:) of each line. Similarly, root is the name of the logger is also displayed the LogRecord.

## Logging Configuration

The logs generated by the program can be customized with `BasicConfig()` method. You can define one or more of the following parameters for configuration –

- **filename** – Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.
- **filemode** – If filename is specified, open the file in this mode. Defaults to 'a'.
- **datefmt** – Use the specified date/time format, as accepted by `time.strftime()`.
- **style** – If format is specified, use this style for the format string. One of '%', '{}' or '\$' for printf-style, `str.format()` or `string.Template` respectively. Defaults to '%'.
- **level** – Set the root logger level to the specified level.
- **errors** – If this keyword argument is specified along with filename, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value 'backslashreplace' is used. Note that if None is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing 'errors'.

## Example

To log all the messages above DEBUG level, set the level parameter to `logging.DEBUG`

</>

Open Compiler

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This message will get logged')
```

It will produce the following **output** –

```
DEBUG:root:This message will get logged
```

To record the logging messages in a file instead of echoing them on the console, use filename parameter.

```
import logging

logging.basicConfig(filename='logs.txt', filemode='w', level=logging.DEBUG)
logging.warning('This message will be saved to a file')
```

No output will be displayed on the console. However, a logs.txt file is created in current directory with the text **WARNING:root:This message will be saved to a file** in it.

## Variable Data in Logging Message

More often than not, you would like to include values of one or more variables in the logging messages to gain more insight into the cause especially of errors generated while the application is running. To do that, any of the dynamic string formatting techniques such as format() method of **str** class, or f-strings can be used.

### Example

```
</>
```

Open Compiler

```
import logging

logging.basicConfig(level=logging.DEBUG)
marks = 120
logging.error("Invalid marks:{} Marks must be between 0 to 100".format(marks))
subjects = ["Phy", "Maths"]
logging.warning("Number of subjects: {}. Should be at least three".format(len(subj
```

It will produce the following **output** –

```
ERROR:root:Invalid marks:120 Marks must be between 0 to 100
WARNING:root:Number of subjects: 2. Should be at least three
```

## Python - Assertions

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

- The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.
- Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

### The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.

The **syntax** for assert is –

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception, using the try-except statement. If they are not handled, they will terminate the program and produce a traceback.

### Example

```
</>
```

Open Compiler

```
print ('enter marks out of 100')
num=75
assert num>=0 and num<=100
```

```

print ('marks obtained: ', num)

num=125
assert num>=0 and num<=100
print ('marks obtained: ', num)

```

It will produce the following **output** –

```

enter marks out of 100
marks obtained: 75
Traceback (most recent call last):
File "C:\Users\user\example.py", line 7, in <module>
assert num>=0 and num<=100
          ^^^^^^^^^^
AssertionError

```

To display custom error message, put a string after the expression in the assert statement

```
assert num>=0 and num<=100, "only numbers in 0-100 accepted"
```

The AssertionError is also a built-in exception. So it can be used as argument in except block. When input causes AssertionError exception, it will be handled by except block. The except block treats string in assert statement goes as exception object.

```

try:
    num=int(input('enter a number'))
    assert (num >=0), "only non negative numbers accepted"
    print (num)
except AssertionError as msg:
    print (msg)

```

## Python - Built-in Exceptions

Here is a list of Standard Exceptions available in Python –

| Sr.No. | Exception Name & Description                      |
|--------|---------------------------------------------------|
| 1      | <b>Exception</b><br>Base class for all exceptions |

|    |                                                                                                                                  |
|----|----------------------------------------------------------------------------------------------------------------------------------|
| 2  | <b>StopIteration</b><br>Raised when the next() method of an iterator does not point to any object.                               |
| 3  | <b>SystemExit</b><br>Raised by the sys.exit() function.                                                                          |
| 4  | <b>StandardError</b><br>Base class for all built-in exceptions except StopIteration and SystemExit.                              |
| 5  | <b>ArithmeticError</b><br>Base class for all errors that occur for numeric calculation.                                          |
| 6  | <b>OverflowError</b><br>Raised when a calculation exceeds maximum limit for a numeric type.                                      |
| 7  | <b>FloatingPointError</b><br>Raised when a floating point calculation fails.                                                     |
| 8  | <b>ZeroDivisionError</b><br>Raised when division or modulo by zero takes place for all numeric types.                            |
| 9  | <b>AssertionError</b><br>Raised in case of failure of the Assert statement.                                                      |
| 10 | <b>AttributeError</b><br>Raised in case of failure of attribute reference or assignment.                                         |
| 11 | <b>EOFError</b><br>Raised when there is no input from either the raw_input() or input() function and the end of file is reached. |
| 12 | <b>ImportError</b><br>Raised when an import statement fails.                                                                     |
| 13 | <b>KeyboardInterrupt</b><br>Raised when the user interrupts program execution, usually by pressing Ctrl+C.                       |
| 14 | <b>LookupError</b><br>Base class for all lookup errors.                                                                          |
| 15 | <b>IndexError</b><br>Raised when an index is not found in a sequence.                                                            |
| 16 | <b>KeyError</b><br>Raised when the specified key is not found in the dictionary.                                                 |

|    |                                                                                                                                                                    |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 17 | <b>NameError</b><br>Raised when an identifier is not found in the local or global namespace.                                                                       |
| 18 | <b>UnboundLocalError</b><br>Raised when trying to access a local variable in a function or method but no value has been assigned to it.                            |
| 19 | <b>EnvironmentError</b><br>Base class for all exceptions that occur outside the Python environment.                                                                |
| 20 | <b>IOError</b><br>Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| 21 | <b>OSError</b><br>Raised for operating system-related errors.                                                                                                      |
| 22 | <b>SyntaxError</b><br>Raised when there is an error in Python syntax.                                                                                              |
| 23 | <b>IndentationError</b><br>Raised when indentation is not specified properly.                                                                                      |
| 24 | <b>SystemError</b><br>Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.              |
| 25 | <b>SystemExit</b><br>Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.          |
| 26 | <b>TypeError</b><br>Raised when an operation or function is attempted that is invalid for the specified data type.                                                 |
| 27 | <b>ValueError</b><br>Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.           |
| 28 | <b>RuntimeError</b><br>Raised when a generated error does not fall into any category.                                                                              |
| 29 | <b>NotImplementedError</b><br>Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.                       |

Here are some examples of standard exceptions –

## IndexError

It is shown when trying to access item at invalid index.

&lt;/&gt;

[Open Compiler](#)

```
numbers=[10,20,30,40]
for n in range(5):
    print (numbers[n])
```

It will produce the following **output** –

```
10
20
30
40
Traceback (most recent call last):
```

```
    print (numbers[n])
IndexError: list index out of range
```

## ModuleNotFoundError

This is displayed when module could not be found.

```
import notamodule
Traceback (most recent call last):

    import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

## KeyError

It occurs as dictionary key is not found.

```
D1={'1':"aa", '2':"bb", '3':"cc"}
print ( D1['4'])
Traceback (most recent call last):

    D1['4']
KeyError: '4'
```

## ImportError

It is shown when specified function is not available for import.

```
from math import cube
Traceback (most recent call last):

    from math import cube
ImportError: cannot import name 'cube'
```

## StopIteration

This error appears when next() function is called after iterator stream exhausts.

```
.it=iter([1,2,3])
next(it)
next(it)
next(it)
next(it)
Traceback (most recent call last):

    next(it)
StopIteration
```

## TypeError

This is shown when operator or function is applied to an object of inappropriate type.

```
print ('2'+2)
Traceback (most recent call last):

'2'+2
TypeError: must be str, not int
```

## ValueError

It is displayed when function's argument is of inappropriate type.

```
print (int('xyz'))
Traceback (most recent call last):
```

```
int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

## NameError

This is encountered when object could not be found.

```
print (age)
Traceback (most recent call last):

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

## ZeroDivisionError

It is shown when second operator in division is zero.

```
x=100/0
Traceback (most recent call last):

    x=100/0
ZeroDivisionError: division by zero
```

## KeyboardInterrupt

When user hits the interrupt key normally Control-C during execution of program.

```
name=input('enter your name')
enter your name^c
Traceback (most recent call last):

    name=input('enter your name')
KeyboardInterrupt
```

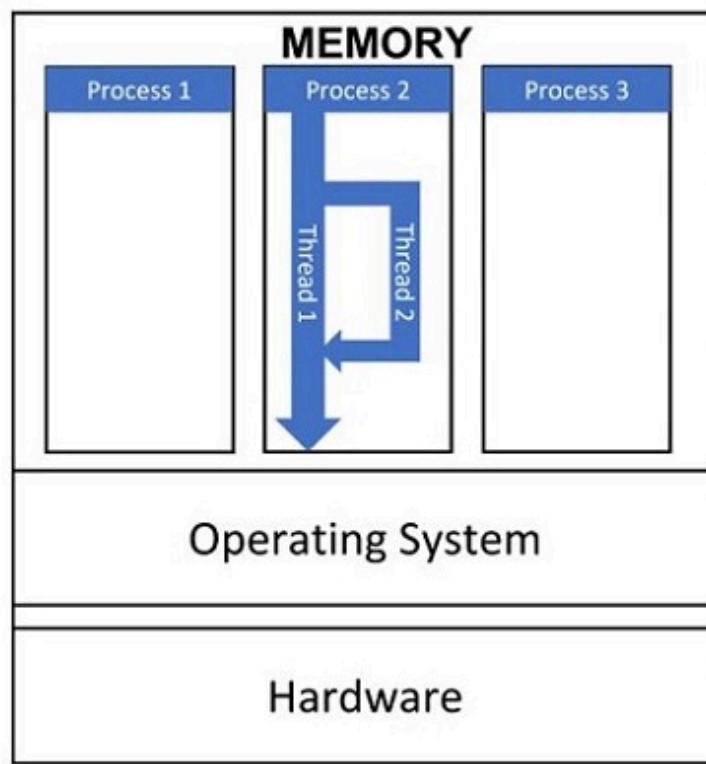
# Python - Multithreading

By default, a computer program executes the instructions in a sequential manner, from start to the end. Multithreading refers to the mechanism of dividing the main task in more than one sub-tasks and executing them in an overlapping manner. This makes the execution faster as compared to single thread.

The operating system is capable of handling multiple processes concurrently. It allocates a separate memory space to each process, so that one process cannot access or write anything other's space. A thread on the other hand can be thought of as a light-weight sub-process in a single program. Threads of a single program share the memory space allocated to it.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

As they are light-weight, do not require much memory overhead; they are cheaper than processes.

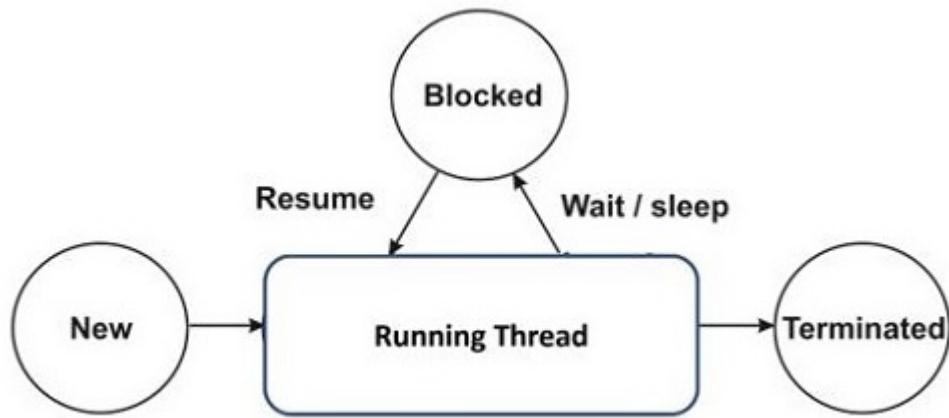


A process always starts with a single thread (main thread). As and when required, a new thread can be started and sub task is delegated to it. Now the two threads are working in an overlapping manner. When the task assigned to the secondary thread is over, it merges with the main thread.

## Python - Thread Life cycle

A thread object goes through different stages. When a new thread object is created, it must be started. This calls the `run()` method of `thread` class. This method contains the logic of the process to be performed by the new thread. The thread completes its task as the `run()` method is over, and the newly created thread merges with the main thread.

While a thread is running, it may be paused either for a predefined duration or it may be asked to pause till a certain event occurs. The thread resumes after the specified interval or the process is over.



Python's standard library has two modules, "`_thread`" and "`threading`", that include the functionality to handle threads. The "`_thread`" module is a low-level API. In Python 3, the **threading module** has been included, which provides more comprehensive functionality for thread management.

## Python The `_thread` Module

The **`_thread`** module (earlier **thread** module) has been a part of Python's standard library since version 2. It is a low-level API for thread management, and works as a support for many of the other modules with advanced concurrent execution features such as threading and multiprocessing.

## Python - The `threading` Module

The newer `threading` module provides much more powerful, high-level support for thread management.

The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass.

```
threading.Thread(target, name, args, kwargs, daemon)
```

### Parameters

- **target** – function to be invoked when a new thread starts. Defaults to None, meaning nothing is called.
- **name** – is the thread name. By default, a unique name is constructed such as "Thread-N".
- **daemon** – If set to True, the new thread runs in the background.
- **args and kwargs** – optional arguments to be passed to target function.

## Python - Creating a Thread

The **start\_new\_thread()** function included in the **\_thread module** is used to create a new thread in the running program.

### Syntax

```
_thread.start_new_thread ( function, args[, kwargs] )
```

This function starts a new thread and returns its identifier.

### Parameters

- **function** – Newly created thread starts running and calls the specified function. If any arguments are required for the function, that may be passed as parameters args and kwargs.

### Example

```
import _thread
import time

# Define a function for the thread
def thread_task( threadName, delay):
    for count in range(1, 6):
        time.sleep(delay)
        print ("Thread name: {} Count: {}".format ( threadName, count ))

# Create two threads as follows
try:
    _thread.start_new_thread( thread_task, ("Thread-1", 2, ) )
    _thread.start_new_thread( thread_task, ("Thread-2", 4, ) )
except:
```

```
print ("Error: unable to start thread")

while True:
    pass
```

It will produce the following **output** –

```
Thread name: Thread-1 Count: 1
Thread name: Thread-2 Count: 1
Thread name: Thread-1 Count: 2
Thread name: Thread-1 Count: 3
Thread name: Thread-2 Count: 2
Thread name: Thread-1 Count: 4
Thread name: Thread-1 Count: 5
Thread name: Thread-2 Count: 3
Thread name: Thread-2 Count: 4
Thread name: Thread-2 Count: 5
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 17, in <module>
    while True:
KeyboardInterrupt
```

The program goes in an infinite loop. You will have to press "ctrl-c" to stop.

## Python - Starting a Thread

This `start()` method starts the thread's activity. It must be called once a thread object is created.

The `start()` method automatically invokes the object's `run()` method in a separate thread. However, if it is called more than once, then a `RuntimeError` will be raised.

### Syntax

Here is the syntax to use the `start()` method in order to start a thread –

```
threading.Thread.start()
```

### Example

Take a look at the following example –

```
thread1 = myThread("Thread-1")  
  
# Start new Thread  
thread1.start()
```

This automatically calls the run() method.

## The run() Method

The run() method represents the thread's activity. It may be overridden in a subclass. Instead of the standard run() method, the object invokes the function passed to its constructor as the target argument.

# Python - Joining the Threads

The join() method in thread class blocks the calling thread until the thread whose join() method is called terminates. The termination may be either normal, because of an unhandled exception – or until the optional timeout occurs. It can be called many times. The join() raises a RuntimeError if an attempt is made to join the current thread. Attempt to join() a thread before it has been started also raises the same exception.

## Syntax

```
thread.join(timeout)
```

## Parameters

- **timeout** – it should be a floating point number specifying a timeout for which the thread is to be blocked.

The join() method always returns None. you must call is\_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out. When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be joined many times.

## Example

```
thread1.start()  
thread2.start()
```

```
thread1.join()  
thread2.join()
```

## is\_alive() method

This method returns whether the thread is alive. It returns True just before calling run() method and until just after the run() method terminates.

# Python - Naming the Threads

The name of a thread is for identification purpose only, and has no role as far as the semantics is concerned. More than one threads may have same name. Thread name can be specified as one of the parameters in thread() constructor.

```
thread(name)
```

Here **name** is the thread name. By default, a unique name is constructed such as "Thread-N".

Thread object also has a property object for getter and setter methods of thread's name attribute.

```
thread.name = "Thread-1"
```

## The daemon Property

A Boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start() is called.

## Example

To implement a new thread using the threading module, you have to do the following –

- Define a new subclass of the Thread class.
- Override the `__init__(self [,args])` method to add additional arguments.
- Then, override the `run(self [,args])` method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the `start()`, which in turn calls the `run()`method.

```
</>
```

Open Compiler

```
import threading
import time
class myThread (threading.Thread):
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):
        print ("Starting " + self.name)
        for count in range(1,6):
            time.sleep(5)
            print ("Thread name: {} Count: {}".format ( self.name, count ))
        print ("Exiting " + self.name)

# Create new threads
thread1 = myThread("Thread-1")
thread2 = myThread("Thread-2")

# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("Exiting Main Thread")
```

It will produce the following **output** –

```
Starting Thread-1
Starting Thread-2
Thread name: Thread-1 Count: 1
Thread name: Thread-2 Count: 1
Thread name: Thread-1 Count: 2
Thread name: Thread-2 Count: 2
Thread name: Thread-1 Count: 3
Thread name: Thread-2 Count: 3
Thread name: Thread-1 Count: 4
Thread name: Thread-2 Count: 4
Thread name: Thread-1 Count: 5
Exiting Thread-1
Thread name: Thread-2 Count: 5
```

Exiting Thread-2

Exiting Main Thread

## Python - Thread Scheduling

Python supports multiple threads in a program. A multi-threaded program can execute multiple sub-tasks independently, which allows the parallel execution of tasks.

Python interpreter maps Python thread requests to either POSIX/pthreads, or Windows threads. Hence, similar to ordinary threads, Python threads are handled by the host operating system.

However, there is no support for thread scheduling in the Python interpreter. Hence, thread priority, scheduling schemes, and thread pre-emption is not possible with the Python interpreter. The scheduling and context switching of Python threads is at the disposal of the host scheduler.

Python does have some support for task scheduling in the form of `sched` module as the standard library. It can be used in the creation of bots and other monitoring and automation applications. The **sched** module implements a generic event scheduler for running tasks at specific times. It provides similar tools like task scheduler in windows or Linux.

The scheduler class is defined in the **sched** built-in module.

```
scheduler(timefunc=time.monotonic, delayfunc=time.sleep)
```

The methods defined in scheduler class include –

- **scheduler.enter()** – Events can be scheduled to run after a delay, or at a specific time. To schedule them with a delay, enter() method is used.
- **scheduler.cancel()** – Remove the event from the queue. If the event is not an event currently in the queue, this method will raise a ValueError.
- **scheduler.run(blocking=True)** – Run all scheduled events.

Events can be scheduled to run after a delay, or at a specific time. To schedule them with a delay, use the enter() method, which takes four arguments.

- A number representing the delay
- A priority value
- The function to call
- A tuple of arguments for the function

## Example 1

This example schedules two different events –

```
</> Open Compiler  
  
import sched  
import time  
  
scheduler = sched.scheduler(time.time, time.sleep)  
  
def schedule_event(name, start):  
    now = time.time()  
    elapsed = int(now - start)  
    print('elapsed=', elapsed, 'name=', name)  
  
    start = time.time()  
    print('START:', time.ctime(start))  
    scheduler.enter(2, 1, schedule_event, ('EVENT_1', start))  
    scheduler.enter(5, 1, schedule_event, ('EVENT_2', start))  
  
scheduler.run()
```

It will produce the following **output** –

```
START: Mon Jun 5 15:37:29 2023  
elapsed= 2 name= EVENT_1  
elapsed= 5 name= EVENT_2
```

## Example 2

Let's take another example to understand the concept better –

```
</>
```

Open Compiler

```
import sched
from datetime import datetime
import time

def addition(a,b):
    print("Performing Addition : ", datetime.now())
    print("Time : ", time.monotonic())
    print("Result : ", a+b)

s = sched.scheduler()

print("Start Time : ", datetime.now())

event1 = s.enter(10, 1, addition, argument = (5,6))
print("Event Created : ", event1)
s.run()
print("End Time : ", datetime.now())
```

It will produce the following **output** –

```
Start Time : 2023-06-05 15:49:49.508400
Event Created : Event(time=774087.453, priority=1, sequence=0, action=<function add
Performing Addition : 2023-06-05 15:49:59.512213
Time : 774087.484
Result : 11
End Time : 2023-06-05 15:49:59.559659
```

## Python - Thread Pools

### What is a Thread Pool?

A thread pool is a mechanism that automatically manages a pool of worker threads. Each thread in the pool is called a worker or a worker thread. Worker threads can be re-used once the task is completed. A single thread is able to execute a single task once.

A thread pool controls when the threads are created, and what threads should do when they are not being used.

The pool is significantly efficient to use a thread pool instead of manually starting, managing, and closing threads, especially with a large number of tasks.

Multiple Threads in Python concurrently execute a certain function. Asynchronous execution of a function by multiple threads can be achieved by ThreadPoolExecutor class defined in concurrent.futures module.

The concurrent.futures module includes Future class and two Executor classes – ThreadPoolExecutor and ProcessPoolExecutor.

## The Future Class

The concurrent.futures.Future class is responsible for handling asynchronous execution of any callable such as a function. To obtain an object of Future class, you should call the submit() method on any Executor object. It should not be created directly by its constructor.

Important methods in the Future class are –

### result(timeout=None)

This method returns the value returned by the call. If the call hasn't yet completed, then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. If timeout is not specified, there is no limit to the wait time.

### cancel()

This method makes attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will return True.

### cancelled()

This method returns True if the call was successfully cancelled.

### running()

This method returns True if the call is currently being executed and cannot be cancelled.

### done()

This method returns True if the call was successfully cancelled or finished running.

## The ThreadPoolExecutor Class

This class represents a pool of specified number maximum worker threads to execute calls asynchronously.

```
concurrent.futures.ThreadPoolExecutor(max_threads)
```

## Example

&lt;/&gt;

Open Compiler

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep
def square(numbers):
    for val in numbers:
        ret = val*val
        sleep(1)
        print("Number:{} Square:{}".format(val, ret))
def cube(numbers):
    for val in numbers:
        ret = val*val*val
        sleep(1)
        print("Number:{} Cube:{}".format(val, ret))
if __name__ == '__main__':
    numbers = [1,2,3,4,5]
    executor = ThreadPoolExecutor(4)
    thread1 = executor.submit(square, (numbers))
    thread2 = executor.submit(cube, (numbers))
    print("Thread 1 executed ? :",thread1.done())
    print("Thread 2 executed ? :",thread2.done())
    sleep(2)
    print("Thread 1 executed ? :",thread1.done())
    print("Thread 2 executed ? :",thread2.done())
```

It will produce the following **output** –

```
Thread 1 executed ? : False
Thread 2 executed ? : False
Number:1 Square:1
Number:1 Cube:1
Number:2 Square:4
Number:2 Cube:8
Thread 1 executed ? : False
Thread 2 executed ? : False
Number:3 Square:9
Number:3 Cube:27
```

```
Number:4 Square:16
Number:4 Cube:64
Number:5 Square:25
Number:5 Cube:125
Thread 1 executed ? : True
Thread 2 executed ? : True
```

## Python - Main Thread

Every Python program has at least one thread of execution called the **main thread**. The **main thread** by default is a non-daemon thread.

Sometimes we may need to create additional threads in our program in order to execute the code concurrently.

Here is the **syntax** for creating a new thread –

```
object = threading.Thread(target, daemon)
```

The Thread() constructor creates a new object. By calling the start() method, the new thread starts running, and it calls automatically a function given as argument to target parameter which defaults to **run**. The second parameter is "daemon" which is by default None.

### Example

```
</> Open Compiler

from time import sleep
from threading import current_thread
from threading import Thread

# function to be executed by a new thread
def run():
    # get the current thread
    thread = current_thread()
    # is it a daemon thread?
    print(f'Daemon thread: {thread.daemon}')

# create a new thread
thread = Thread(target=run)
```

```
# start the new thread  
thread.start()  
  
# block for a 0.5 sec  
sleep(0.5)
```

It will produce the following **output** –

```
Daemon thread: False
```

So, creating a thread by the following statement –

```
t1=threading.Thread(target=run)
```

This statement creates a non-daemon thread. When started, it calls the run() method.

## Python - Thread Priority

The **queue** module in Python's standard library is useful in threaded programming when information must be exchanged safely between multiple threads. The Priority Queue class in this module implements all the required locking semantics.

With a priority queue, the entries are kept sorted (using the **heapq** module) and the lowest valued entry is retrieved first.

The Queue objects have following methods to control the Queue –

- **get()** – The get() removes and returns an item from the queue.
- **put()** – The put adds item to a queue.
- **qsize()** – The qsize() returns the number of items that are currently in the queue.
- **empty()** – The empty( ) returns True if queue is empty; otherwise, False.
- **full()** – the full() returns True if queue is full; otherwise, False.

```
queue.PriorityQueue(maxsize=0)
```

This is the Constructor for a priority queue. maxsize is an integer that sets the upper limit on the number of items that can be placed in the queue. If maxsize is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by min(entries)). A typical pattern for entries is a tuple in the form –

```
(priority_number, data)
```

## Example

</>

Open Compiler

```
from time import sleep
from random import random, randint
from threading import Thread
from queue import PriorityQueue

queue = PriorityQueue()

def producer(queue):
    print('Producer: Running')
    for i in range(5):

        # create item with priority
        value = random()
        priority = randint(0, 5)
        item = (priority, value)
        queue.put(item)

    # wait for all items to be processed
    queue.join()

    queue.put(None)
    print('Producer: Done')

def consumer(queue):
    print('Consumer: Running')

    while True:

        # get a unit of work
        item = queue.get()
        if item is None:
            break

        sleep(item[1])
        print(item)
        queue.task_done()
```

```
print('Consumer: Done')

producer = Thread(target=producer, args=(queue,))
producer.start()

consumer = Thread(target=consumer, args=(queue,))
consumer.start()

producer.join()
consumer.join()
```

It will produce the following **output** –

```
Producer: Running
Consumer: Running
(0, 0.15332707626852804)
(2, 0.4730737391435892)
(2, 0.8679231358257962)
(3, 0.051924220435665025)
(4, 0.23945882716108446)
Producer: Done
Consumer: Done
```

## Python - Daemon Threads

Sometimes, it is necessary to execute a task in the background. A special type of thread is used for background tasks, called a **daemon thread**. In other words, daemon threads execute tasks in the background.

It may be noted that daemon threads execute such non-critical tasks that although may be useful to the application but do not hamper it if they fail or are canceled mid-operation.

Also, a daemon thread will not have control over when it is terminated. The program will terminate once all non-daemon threads finish, even if there are daemon threads still running at that point of time.

This is a major difference between daemon threads and non-daemon threads. The process will exit if only daemon threads are running, whereas it cannot exit if at least one non-daemon thread is running.

**Daemon**

**Non-daemon**

A process will exit if only daemon threads are running (or if no threads are running).

A process will not exit if at least one non-daemon thread is running.

## Creating a Daemon Thread

To create a daemon thread, you need to set the **daemon** property to True.

```
t1=threading.Thread(daemon=True)
```

If a thread object is created without any parameter, its daemon property can also be set to True, before calling the start() method.

```
t1=threading.Thread()  
t1.daemon=True
```

## Example

Take a look at the following example –

</>

Open Compiler

```
from time import sleep  
from threading import current_thread  
from threading import Thread  
  
# function to be executed in a new thread  
def run():  
  
    # get the current thread  
    thread = current_thread()  
    # is it a daemon thread?  
    print(f'Daemon thread: {thread.daemon}')  
  
    # create a new thread  
    thread = Thread(target=run, daemon=True)  
  
    # start the new thread  
    thread.start()  
  
    # block for a 0.5 sec for daemon thread to run  
    sleep(0.5)
```

It will produce the following **output** –

Daemon thread: True

Daemon threads can perform executing tasks that support non-daemon threads in the program. For example –

- Create a file that stores Log information in the background.
- Perform web scraping in the background.
- Save the data automatically into a database in the background.

## Example

If a running thread is configured to be daemon, then a RuntimeError is raised. Take a look at the following example –

```
</> Open Compiler

from time import sleep
from threading import current_thread
from threading import Thread

# function to be executed in a new thread
def run():
    # get the current thread
    thread = current_thread()
    # is it a daemon thread?
    print(f'Daemon thread: {thread.daemon}')
    thread.daemon = True

# create a new thread
thread = Thread(target=run)

# start the new thread
thread.start()

# block for a 0.5 sec for daemon thread to run
sleep(0.5)
```

It will produce the following **output** –

Exception in thread Thread-1 (run):

Traceback (most recent call last):

....

....

```
thread.daemon = True  
^^^^^^^^^^^^^^^^^
```

File "C:\Python311\Lib\threading.py", line 1219, in daemon

```
raise RuntimeError("cannot set daemon status of active thread")
```

RuntimeError: cannot set daemon status of active thread

## Python - Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

The acquire(blocking) method of the new lock object is used to force the threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.

If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The release() method of the new lock object is used to release the lock when it is no longer required.

### Example

</>

Open Compiler

```
import threading  
  
import time  
  
class myThread (threading.Thread):  
    def __init__(self, threadID, name, counter):  
        threading.Thread.__init__(self)  
        self.threadID = threadID  
        self.name = name  
        self.counter = counter  
    def run(self):  
        print ("Starting " + self.name)
```

```
# Get lock to synchronize threads
threadLock.acquire()
print_time(self.name, self.counter, 3)
# Free lock to release next thread
threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

## Output

When the above code is executed, it produces the following output –

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Jul 13 21:10:11 2023
Thread-1: Thu Jul 13 21:10:12 2023
Thread-1: Thu Jul 13 21:10:13 2023
Thread-2: Thu Jul 13 21:10:15 2023
```

Thread-2: Thu Jul 13 21:10:17 2023

Thread-2: Thu Jul 13 21:10:19 2023

Exiting Main Thread

## Python - Inter-Thread Communication

Threads share the memory allocated to a process. As a result, threads in the same process can communicate with each other. To facilitate inter-thread communication, the `threading` module provides `Event` object and `Condition` object.

### The Event Object

An `Event` object manages the state of an internal flag. The flag is initially false and becomes true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

Methods of `Event` object –

#### `is_set()` method

Return True if and only if the internal flag is true.

#### `set()` method

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

#### `clear()` method

Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

#### `wait(timeout=None)` method

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds.

### Example

The following code attempts to simulate the traffic flow being controlled by the state of traffic signal either GREEN or RED.

There are two threads in the program, targeting two different functions. The `signal_state()` function periodically sets and resets the event indicating change of signal from GREEN to RED.

The `traffic_flow()` function waits for the event to be set, and runs a loop till it remains set.

&lt;/&gt;

Open Compiler

```
from threading import *
import time

def signal_state():
    while True:
        time.sleep(5)
        print("Traffic Police Giving GREEN Signal")
        event.set()
        time.sleep(10)
        print("Traffic Police Giving RED Signal")
        event.clear()

def traffic_flow():
    num=0
    while num<10:
        print("Waiting for GREEN Signal")
        event.wait()
        print("GREEN Signal ... Traffic can move")
        while event.is_set():
            num=num+1
            print("Vehicle No:", num, " Crossing the Signal")
            time.sleep(2)
        print("RED Signal ... Traffic has to wait")

    event=Event()
    t1=Thread(target=signal_state)
    t2=Thread(target=traffic_flow)
    t1.start()
    t2.start()
```

## Output

```
Waiting for GREEN Signal
Traffic Police Giving GREEN Signal
GREEN Signal ... Traffic can move
Vehicle No: 1 Crossing the Signal
Vehicle No: 2 Crossing the Signal
Vehicle No: 3 Crossing the Signal
Vehicle No: 4 Crossing the Signal
Vehicle No: 5 Crossing the Signal
Signal is RED
RED Signal ... Traffic has to wait
Waiting for GREEN Signal
Traffic Police Giving GREEN Signal
GREEN Signal ... Traffic can move
Vehicle No: 6 Crossing the Signal
Vehicle No: 7 Crossing the Signal
Vehicle No: 8 Crossing the Signal
Vehicle No: 9 Crossing the Signal
Vehicle No: 10 Crossing the Signal
```

## The Condition Object

Condition class in threading module class implements condition variable objects. Condition object forces one or more threads to wait until notified by another thread. Condition is associated with a Reentrant Lock. A condition object has acquire() and release() methods that call the corresponding methods of the associated lock.

```
threading.Condition(lock=None)
```

Following are the methods of the Condition object –

### acquire(\*args)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

### release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

### wait(timeout=None)

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

## `wait_for(predicate, timeout=None)`

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

## `notify(n=1)`

This method wakes up at most `n` of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

## `notify_all()`

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

## Example

In the following code, the thread `t2` runs `taskB()` function and `t1` runs `taskA()` function. The `t1` thread acquires the condition and notifies it. By that time the `t2` thread is in waiting state. After the condition is released, the waiting thread proceeds to consume the random number generated by the notifying function.

&lt;/&gt;

Open Compiler

```
from threading import *
import time
import random

numbers=[]
def taskA(c):
    while True:
        c.acquire()
        num=random.randint(1,10)
        print("Generated random number:", num)
        numbers.append(num)
        print("Notification issued")
        c.notify()
        c.release()
```

```
time.sleep(5)

def taskB(c):
    while True:
        c.acquire()
        print("waiting for update")
        c.wait()
        print("Obtained random number", numbers.pop())
        c.release()
        time.sleep(5)

c=Condition()
t1=Thread(target=taskB, args=(c,))
t2=Thread(target=taskA, args=(c,))
t1.start()
t2.start()
```

When you execute this code, it will produce the following **output** –

```
waiting for update
Generated random number: 4
Notification issued
Obtained random number 4
waiting for update
Generated random number: 6
Notification issued
Obtained random number 6
waiting for update
Generated random number: 10
Notification issued
Obtained random number 10
waiting for update
```

## Python - Thread Deadlock

A deadlock may be described as a concurrency failure mode. It is a situation in a program where one or more threads wait for a condition that never occurs. As a result, the threads are unable to progress and the program is stuck or frozen and must be terminated manually.

Deadlock situation may arise in many ways in your concurrent program. Deadlocks are never developed intentionally, instead, they are in fact a side effect or bug in the code.

Common causes of thread deadlocks are listed below –

- A thread that attempts to acquire the same mutex lock twice.
- Threads that wait on each other (e.g. A waits on B, B waits on A).
- When a thread that fails to release a resource such as lock, semaphore, condition, event, etc.
- Threads that acquire mutex locks in different orders (e.g. fail to perform lock ordering).

If more than one threads in a multi-threaded application try to gain access to same resource, such as performing read/write operation on same file, it may cause data inconsistency. Hence it is important that the concurrent handling is synchronized so that it is locked for other threads when one thread is using the resource.

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

## The Lock Object

An object of Lock class has two possible states – locked or unlocked, initially in unlocked state when first created. A lock doesn't belong to any particular thread.

The Lock class defines acquire() and release() methods.

### The acquire() Method

When the state is unlocked, this method changes the state to locked and returns immediately. The method takes an optional blocking argument.

### Syntax

```
Lock.acquire(blocking, timeout)
```

### Parameters

- **blocking** – If set to False, it means do not block. If a call with blocking set to True would block, return False immediately; otherwise, set the lock to locked and return True.

The return value of this method is True if the lock is acquired successfully; False if not.

## The release() Method

When the state is locked, this method in another thread changes it to unlocked. This can be called from any thread, not only the thread which has acquired the lock

### Syntax

```
Lock.release()
```

The release() method should only be called in the locked state. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. There is no return value of this method.

### Example

In the following program, two threads try to call the synchronized() method. One of them acquires the lock and gains the access while the other waits. When the run() method is completed for the first thread, the lock is released and the synchronized method is available for second thread.

When both the threads join, the program comes to an end.

```
</>
```

Open Compiler

```
from threading import Thread, Lock
import time

lock=Lock()
threads=[]

class myThread(Thread):
    def __init__(self,name):
        Thread.__init__(self)
        self.name=name
    def run(self):
        lock.acquire()
        synchronized(self.name)
        lock.release()

    def synchronized(threadName):
        print ("{} has acquired lock and is running synchronized method".format(threadN
```

```
counter=5
while counter:
    print ('***', end=' ')
    time.sleep(2)
    counter=counter-1
print('\nlock released for', threadName)

t1=myThread('Thread1')
t2=myThread('Thread2')

t1.start()
threads.append(t1)

t2.start()
threads.append(t2)

for t in threads:
    t.join()
print ("end of main thread")
```

It will produce the following **output** –

```
Thread1 has acquired lock and is running synchronized method
*****
lock released for Thread1
Thread2 has acquired lock and is running synchronized method
*****
lock released for Thread2
end of main thread
```

## The Semaphore Object

Python supports thread synchronization with another mechanism called **semaphore**. It is one of the oldest synchronization techniques invented by a well-known computer scientist, Edsger W. Dijkstra.

The basic concept of semaphore is to use an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

The Semaphore class in threading module defines acquire() and release() methods.

## The acquire() Method

If the internal counter is larger than zero on entry, decrement it by one and return True immediately.

If the internal counter is zero on entry, block until awoken by a call to release(). Once awoken (and the counter is greater than 0), decrement the counter by 1 and return True. Exactly one thread will be awoken by each call to release(). The order in which threads awake is arbitrary.

If blocking parameter is set to False, do not block. If a call without an argument would block, return False immediately; otherwise, do the same thing as when called without arguments, and return True.

## The release() Method

Release a semaphore, incrementing the internal counter by 1. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up n of those threads.

## Example

```
</> Open Compiler

from threading import *
import time

# creating thread instance where count = 3
lock = Semaphore(4)

# creating instance
def synchronized(name):

    # calling acquire method
    lock.acquire()

    for n in range(3):
        print('Hello! ', end = ' ')
        time.sleep(1)
        print( name)

    # calling release method
    lock.release()
```

```
# creating multiple thread
thread_1 = Thread(target = synchronized , args = ('Thread 1',))
thread_2 = Thread(target = synchronized , args = ('Thread 2',))
thread_3 = Thread(target = synchronized , args = ('Thread 3',))

# calling the threads
thread_1.start()
thread_2.start()
thread_3.start()
```

It will produce the following **output** –

```
Hello! Hello! Hello! Thread 1
```

```
Hello! Thread 2
```

```
Thread 3
```

```
Hello! Hello! Thread 1
```

```
Hello! Thread 3
```

```
Thread 2
```

```
Hello! Hello! Thread 1
```

```
Thread 3
```

```
Thread 2
```

## Python - Interrupting a Thread

In a multi-threaded program, a task in a new thread, may be required to be stopped. This may be for many reasons, such as: (a) The result from the task is no longer required or (b) outcome from the task has gone astray or (c) The application is shutting down.

A thread can be stopped using a `threading.Event` object. An Event object manages the state of an internal flag that can be either set or not set.

When a new Event object is created, its flag is not set (false) to start. If its `set()` method is called by one thread, its flag value can be checked in another thread. If found to be true, you can terminate its activity.

### Example

In this example, we have a `MyThread` class. Its object starts executing the `run()` method. The main thread sleeps for a certain period and then sets an event. Till the event is detected, loop in the `run()` method continues. As soon as the event is detected, the loop terminates.

</>

Open Compiler

```
from time import sleep
from threading import Thread
from threading import Event

class MyThread(Thread):
    def __init__(self, event):
        super(MyThread, self).__init__()
        self.event = event

    def run(self):
        i=0
        while True:
            i+=1
            print ('Child thread running...',i)
            sleep(0.5)
            if self.event.is_set():
                break
            print()
        print('Child Thread Interrupted')

event = Event()
thread1 = MyThread(event)
thread1.start()

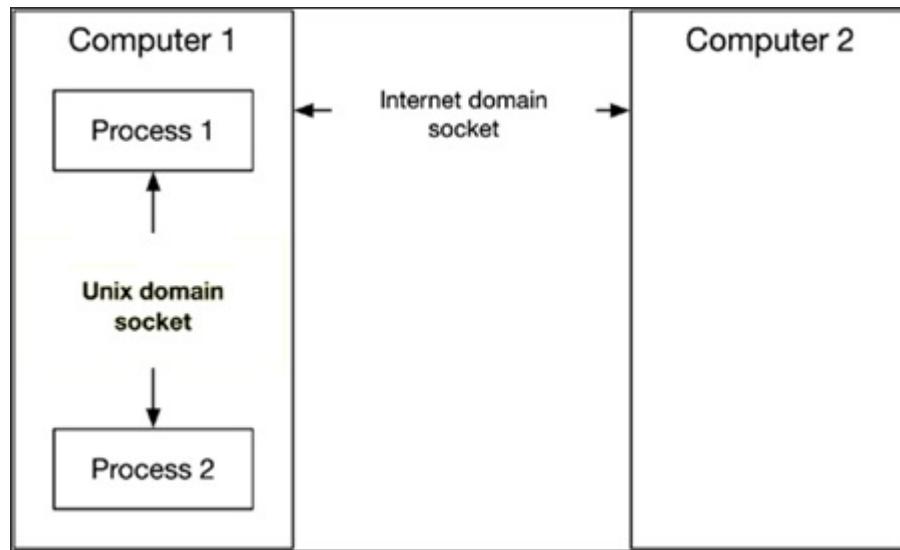
sleep(3)
print('Main thread stopping child thread')
event.set()
thread1.join()
```

When you execute this code, it will produce the following **output** –

```
Child thread running... 1
Child thread running... 2
Child thread running... 3
Child thread running... 4
Child thread running... 5
Child thread running... 6
Main thread stopping child thread
Child Thread Interrupted
```

# Python - Network Programming

The threading module in Python's standard library is capable of handling multiple threads and their interaction within a single process. Communication between two processes running on the same machine is handled by Unix domain sockets, whereas for the processes running on different machines connected with TCP (Transmission control protocol), Internet domain sockets are used.



Python's standard library consists of various built-in modules that support interprocess communication and networking. Python provides two levels of access to the network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

| Protocol | Common function    | Port No | Python module              |
|----------|--------------------|---------|----------------------------|
| HTTP     | Web pages          | 80      | httplib, urllib, xmlrpclib |
| NNTP     | Usenet news        | 119     | nntplib                    |
| FTP      | File transfers     | 20      | ftplib, urllib             |
| SMTP     | Sending email      | 25      | smtplib                    |
| POP3     | Fetching email     | 110     | poplib                     |
| IMAP4    | Fetching email     | 143     | imaplib                    |
| Telnet   | Command lines      | 23      | telnetlib                  |
| Gopher   | Document transfers | 70      | gopherlib, urllib          |

# Python - Socket Programming

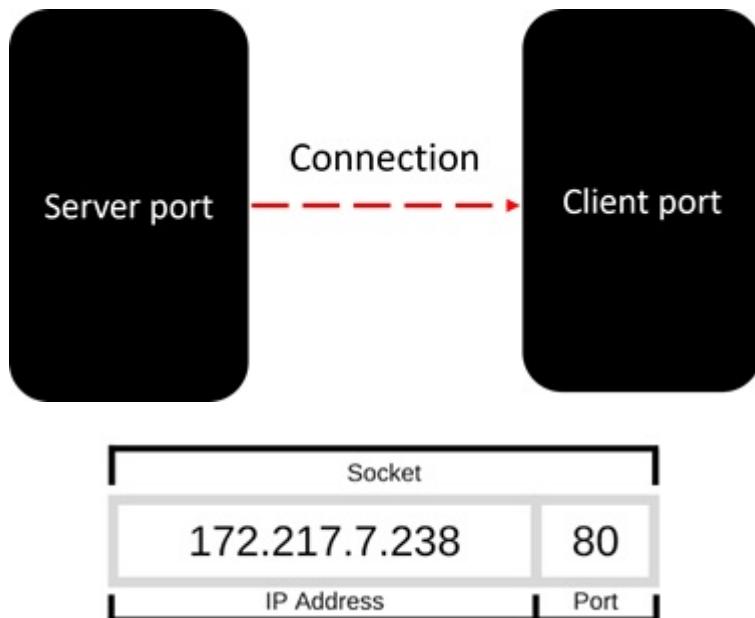
The socket module in the standard library included functionality required for communication between server and client at hardware level.

This module provides access to the BSD socket interface. It is available on all operating systems such as Linux, Windows, MacOS.

## What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

A socket is identified by the combination of IP address and the port number. It should be properly configured at both ends to begin communication.



Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

The term socket programming implies programmatically setting up sockets to be able to send and receive data.

There are two types of communication protocols –

- connection-oriented protocol
- connection-less protocol

TCP or Transmission Control Protocol is a connection-oriented protocol. The data is transmitted in packets by the server, and assembled in the same order of transmission by the receiver. Since the sockets at either end of the communication need to be set before starting, this method is more reliable.

UDP or User Datagram Protocol is connectionless. The method is not reliable because the sockets does not require establishing any connection and termination process for transferring the data.

## Python The socket Module

This module includes Socket class. A socket object represents the pair of hostname and port number. The constructor method has the following signature –

### Syntax

```
socket.socket (socket_family, socket_type, protocol=0)
```

### Parameters

- **family** – AF\_INET by default. Other values - AF\_INET6 (eight groups of four hexadecimal digits), AF\_UNIX, AF\_CAN (Controller Area Network) or AF\_RDS (Reliable Datagram Sockets).
- **socket\_type** – should be SOCK\_STREAM (the default), SOCK\_DGRAM, SOCK\_RAW or perhaps one of the other SOCK\_ constants.
- **protocol** – number is usually zero and may be omitted.

### Return Type

This method returns a socket object.

Once you have the socket object, then you can use the required methods to create your client or server program.

## Server Socket Methods

The socket instantiated on server is called server socket. Following methods are available to the socket object on the server –

- **bind() method** – This method binds the socket to specified IP address and port number.

- **listen() method** – This method starts server and runs into a listen loop looking for connection request from client.
- **accept() method** – When connection request is intercepted by server, this method accepts it and identifies the client socket with its address.

To create a socket on a server, use the following snippet –

```
import socket
server = socket.socket()
server.bind(('localhost', 12345))
server.listen()
client, addr = server.accept()
print ("connection request from: " + str(addr))
```

By default, the server is bound to local machine's IP address 'localhost' listening at arbitrary empty port number.

## Client Socket Methods

Similar socket is set up on the client end. It mainly sends connection request to server socket listening at its IP address and port number

### connect() method

This method takes a two-item tuple object as argument. The two items are IP address and port number of the server.

```
obj=socket.socket()
obj.connect((host,port))
```

Once the connection is accepted by the server, both the socket objects can send and/or receive data.

### send() method

The server sends data to client by using the address it has intercepted.

```
client.send(bytes)
```

Client socket sends data to socket it has established connection with.

### sendall() method

similar to send(). However, unlike send(), this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success.

## sendto() method

This method is to be used in case of UDP protocol only.

## recv() method

This method is used to retrieve data sent to the client. In case of server, it uses the remote socket whose request has been accepted.

```
client.recv(bytes)
```

## recvfrom() method

This method is used in case of UDP protocol.

# Python - Socket Server

To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call the bind(hostname, port) function to specify a port for your service on the given host.

Next, call the accept method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

```
import socket
host = "127.0.0.1"
port = 5001
server = socket.socket()
server.bind((host, port))
server.listen()
conn, addr = server.accept()
print ("Connection from: " + str(addr))
while True:
    data = conn.recv(1024).decode()
    if not data:
        break
    data = str(data).upper()
```

```
print (" from client: " + str(data))
data = input("type message: ")
conn.send(data.encode())
conn.close()
```

## Python - Socket Client

Let us write a very simple client program, which opens a connection to a given port 5001 and a given localhost. It is very simple to create a socket client using the Python's socket module function.

The **socket.connect(hostname, port)** opens a TCP connection to hostname on the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits when 'q' is entered.

```
import socket
host = '127.0.0.1'
port = 5001
obj = socket.socket()
obj.connect((host,port))
message = input("type message: ")
while message != 'q':
    obj.send(message.encode())
    data = obj.recv(1024).decode()
    print ('Received from server: ' + data)
    message = input("type message: ")
obj.close()
```

- Run Server code first. It starts listening.
- Then start client code. It sends request.
- Request accepted. Client address identified
- Type some text and press Enter.
- Data received is printed. Send data to client.
- Data from server is received.
- Loop terminates when 'q' is input.

Server-client interaction is shown below –

The image shows two separate Command Prompt windows. The top window, titled 'Command Prompt', displays the output of a Python server script named 'socketsrvr.py'. It shows a connection from a client at '127.0.0.1' port 52324, followed by four messages: 'HELLO PYTHON', 'Thank you', 'GOOD BYE', and 'See you later'. The bottom window, also titled 'Command Prompt', shows the output of a Python client script named 'socketclient.py'. It sends a message 'Hello Python' to the server, receives 'Thank you' in return, sends 'Good Bye', receives 'See you later', and finally sends a 'q' to quit.

```
D:\socketsrvr>python socketsrvr.py
Connection from: ('127.0.0.1', 52324)
  from client: HELLO PYTHON
  type message: Thank you
  from client: GOOD BYE
  type message: See you later

D:\socketsrvr>

D:\socketsrvr>python socketclient.py
type message: Hello Python
Received from server: Thank you
type message: Good Bye
Received from server: See you later
type message: q

D:\socketsrvr>
```

We have implemented client-server communication with socket module on the local machine. To put server and client codes on two different machines on a network, we need to find the IP address of the server machine.

On Windows, you can find the IP address by running ipconfig command. The ifconfig command is the equivalent command on Ubuntu.

A screenshot of a Windows Command Prompt window showing the output of the ipconfig command. The output lists network adapters and their configuration. The 'IPv4 Address' for the 'Wireless LAN adapter Wi-Fi' is highlighted with a red box. The address is 192.168.1.218.

```
Command Prompt

Wireless LAN adapter Local Area Connection* 68:
  Media State . . . . . : Media disconnected
  Connection-specific DNS Suffix . . . .

Wireless LAN adapter Wi-Fi:
  Connection-specific DNS Suffix . . . . . : ib-wrb304n.setup.in
  Link-local IPv6 Address . . . . . : fe80::a1a2:ef95:61fa:9b7a%16
  IPv4 Address . . . . . : 192.168.1.218
  Subnet Mask . . . . . : 255.255.255.0
  Default Gateway . . . . . : 192.168.1.1
```

Change host string in both the server and client codes with IPv4 Address value and run them as before.

## Python File Transfer with Socket Module

The following program demonstrates how socket communication can be used to transfer a file from server to the client

## Server Code

The code for establishing connection is same as before. After the connection request is accepted, a file on server is opened in binary mode for reading, and bytes are successively read and sent to the client stream till end of file is reached.

```
import socket
host = "127.0.0.1"
port = 5001
server = socket.socket()
server.bind((host, port))
server.listen()
conn, addr = server.accept()
data = conn.recv(1024).decode()
filename='test.txt'
f = open(filename, 'rb')
while True:
    l = f.read(1024)
    if not l:
        break
    conn.send(l)
    print('Sent ',repr(l))
f.close()
print('File transferred')
conn.close()
```

## Client Code

On the client side, a new file is opened in **wb** mode. The stream of data received from server is written to the file. As the stream ends, the output file is closed. A new file will be created on the client machine.

```
import socket

s = socket.socket()
host = "127.0.0.1"
port = 5001

s.connect((host, port))
s.send("Hello server!".encode())

with open('recv.txt', 'wb') as f:
    while True:
```

```

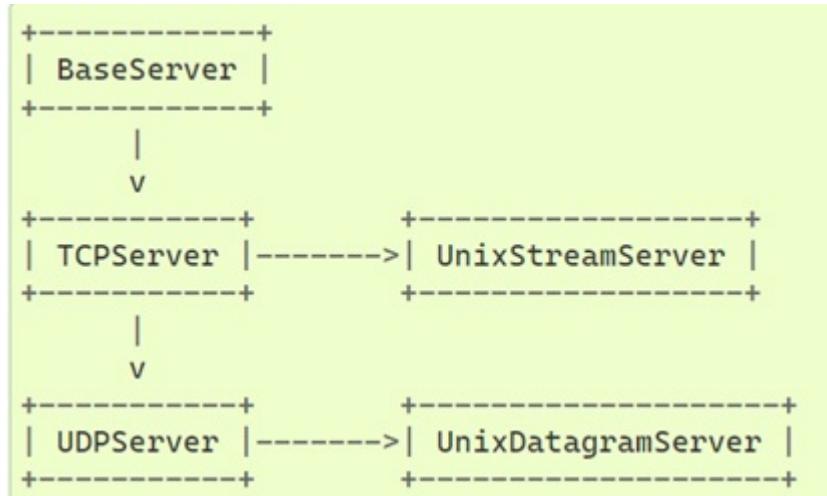
print('receiving data...')
data = s.recv(1024)
if not data:
    break
f.write(data)

f.close()
print('Successfully received')
s.close()
print('connection closed')

```

## Python The socketserver Module

The socketserver module in Python's standard library is a framework for simplifying task of writing network servers. There are following classes in module, which represent synchronous servers –



These classes work with corresponding RequestHandler classes for implementing the service. **BaseServer** is the superclass of all Server objects in the module.

**TCPServer** class uses the internet TCP protocol, to provide continuous streams of data between the client and server. The constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the `BaseServer` base class.

You must also create a subclass of **StreamRequestHandler** class. IT provides `self.rfile` and `self.wfile` attributes to read or write to get the request data or return data to the client.

- **UDPServer** and **DatagramRequestHandler** – These classes are meant to be used for UDP protocol.

- **DatagramRequestHandler** and **UnixDatagramServer** – These classes use Unix domain sockets; they're not available on non-Unix platforms.

## Server Code

You must write a RequestHandler class. It is instantiated once per connection to the server, and must override the handle() method to implement communication to the client.

```
import socketserver
class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        self.data = self.request.recv(1024).strip()
        host, port = self.client_address
        print("{}:{} wrote:".format(host, port))
        print(self.data.decode())
        msg = input("enter text .. ")
        self.request.sendall(msg.encode())
```

On the server's assigned port number, an object of TCPServer class calls the forever() method to put the server in the listening mode and accepts incoming requests from clients.

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        server.serve_forever()
```

## Client Code

When working with socketserver, the client code is more or less similar with the socket client application.

```
import socket
import sys

HOST, PORT = "localhost", 9999

while True:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        # Connect to server and send data
        sock.connect((HOST, PORT))
        data = input("enter text .. .")
        sock.sendall(bytes(data + "\n", "utf-8"))
```

```
# Receive data from the server and shut down
received = str(sock.recv(1024), "utf-8")
print("Sent: {}".format(data))
print("Received: {}".format(received))
```

Run the server code in one command prompt terminal. Open multiple terminals for client instances. You can simulate a concurrent communication between the server and more than one clients.

| Server                                                                                                                                                                                                                                                                                                                                                                                       | Client-1                                                                                                                                                                                                                                                                              | Client-2                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| D:\socketsrvr>python<br>myserver.py<br>127.0.0.1:54518 wrote:<br>from client-1<br>enter text ..<br>hello<br>127.0.0.1:54522 wrote:<br>how are you<br>enter text ..<br>fine<br>127.0.0.1:54523 wrote:<br>from client-2<br>enter text ..<br>hi client-2<br>127.0.0.1:54526 wrote:<br>good bye<br>enter text ..<br>bye bye<br>127.0.0.1:54530 wrote:<br>thanks<br>enter text ..<br>bye client-2 | D:\socketsrvr>python<br>myclient.py<br>enter text ...<br>from client-1<br>Sent:<br>from client-1<br>Received: hello<br>enter text ...<br>how are you<br>Sent:<br>how are you<br>Received: fine<br>enter text ...<br>good bye<br>Sent: good bye<br>Received: bye bye<br>enter text ... | D:\socketsrvr>python<br>myclient.py<br>enter text ...<br>from client-2<br>Sent:<br>from client-2<br>Received: hi client-2<br>enter text ...<br>thanks<br>Sent: thanks<br>Received:<br>bye client-2<br>enter text ... |

## Python - URL Processing

In the world of Internet, different resources are identified by URLs (Uniform Resource Locators). The `urllib` package which is bundled with Python's standard library provides several utilities to handle URLs. It has the following modules –

- **urllib.parse** module is used for parsing a URL into its parts.
- **urllib.request** module contains functions for opening and reading URLs

- **urllib.error** module carries definitions of the exceptions raised by urllib.request
- **urllib.robotparser** module parses the robots.txt files

## The urllib.parse Module

This module serves as a standard interface to obtain various parts from a URL string. The module contains following functions –

### urlparse(urlstring)

Parse a URL into six components, returning a 6-item named tuple. Each tuple item is a string corresponding to following attributes –

| Attribute | Index | Value                              |
|-----------|-------|------------------------------------|
| scheme    | 0     | URL scheme specifier               |
| netloc    | 1     | Network location part              |
| path      | 2     | Hierarchical path                  |
| params    | 3     | Parameters for last path element   |
| query     | 4     | Query component                    |
| fragment  | 5     | Fragment identifier                |
| username  |       | User name                          |
| password  |       | Password                           |
| hostname  |       | Host name (lower case)             |
| Port      |       | Port number as integer, if present |

## Example

&lt;/&gt;

Open Compiler

```
from urllib.parse import urlparse
url = "https://example.com/employees/name/?salary>=25000"
parsed_url = urlparse(url)
print (type(parsed_url))
print ("Scheme:", parsed_url.scheme)
```

```

print ("netloc:", parsed_url.netloc)
print ("path:", parsed_url.path)
print ("params:", parsed_url.params)
print ("Query string:", parsed_url.query)
print ("Frgment:", parsed_url.fragment)

```

It will produce the following **output** –

```

<class 'urllib.parse.ParseResult'>
Scheme: https
netloc: example.com
path: /employees/name/
params:
Query string: salary>=25000
Frgment:

```

## parse\_qs(qs))

This function Parse a query string given as a string argument. Data is returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

To further fetch the query parameters from the query string into a dictionary, use `parse_qs()` function of the `query` attribute of `ParseResult` object as follows –

</>
Open Compiler

```

from urllib.parse import urlparse, parse_qs
url = "https://example.com/employees?name=Anand&salary=25000"
parsed_url = urlparse(url)
dct = parse_qs(parsed_url.query)
print ("Query parameters:", dct)

```

It will produce the following **output** –

```
Query parameters: {'name': ['Anand'], 'salary': ['25000']}
```

## urlsplit(urlstring)

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters

to be applied to each segment of the path portion of the URL is wanted.

## urlunparse(parts)

This function is the opposite of urlparse() function. It constructs a URL from a tuple as returned by urlparse(). The parts argument can be any six-item iterable. This returns an equivalent URL.

### Example

&lt;/&gt;

Open Compiler

```
from urllib.parse import urlunparse

lst = ['https', 'example.com', '/employees/name/', '', 'salary>=25000', '']
new_url = urlunparse(lst)
print ("URL:", new_url)
```

It will produce the following **output** –

URL: https://example.com/employees/name/?salary>=25000

## urlunsplit(parts)

Combine the elements of a tuple as returned by urlsplit() into a complete URL as a string. The parts argument can be any five-item iterable.

## The urllib.request Module

This module defines functions and classes which help in opening URLs

### urlopen() function

This function opens the given URL, which can be either a string or a Request object. The optional timeout parameter specifies a timeout in seconds for blocking operations. This actually only works for HTTP, HTTPS and FTP connections.

This function always returns an object which can work as a context manager and has the properties url, headers, and status.

For HTTP and HTTPS URLs, this function returns a http.client.HTTPResponse object slightly modified.

## Example

The following code uses `urlopen()` function to read the binary data from an image file, and writes it to local file. You can open the image file on your computer using any image viewer.

```
from urllib.request import urlopen
obj = urlopen("https://www.tutorialspoint.com/static/images/simply-easy-learning.jpg")
data = obj.read()
img = open("img.jpg", "wb")
img.write(data)
img.close()
```

It will produce the following **output** –



## The Request Object

The `urllib.request` module includes `Request` class. This class is an abstraction of a URL request. The constructor requires a mandatory string argument a valid URL.

## Syntax

```
urllib.request.Request(url, data, headers, origin_req_host, method=None)
```

## Parameters

- **url** – A string that is a valid URL
- **data** – An object specifying additional data to send to the server. This parameter can only be used with HTTP requests. Data may be bytes, file-like objects, and

iterables of bytes-like objects.

- **headers** – Should be a dictionary of headers and their associated values.
- **origin\_req\_host** – Should be the request-host of the origin transaction
- **method** – should be a string that indicates the HTTP request method. One of GET, POST, PUT, DELETE and other HTTP verbs. Default is GET.

## Example

```
from urllib.request import Request  
obj = Request("https://www.tutorialspoint.com/")
```

This Request object can now be used as an argument to urlopen() method.

```
from urllib.request import Request, urlopen  
obj = Request("https://www.tutorialspoint.com/")  
resp = urlopen(obj)
```

The urlopen() function returns a `HttpResponse` object. Calling its `read()` method fetches the resource at the given URL.

```
from urllib.request import Request, urlopen  
obj = Request("https://www.tutorialspoint.com/")  
resp = urlopen(obj)  
data = resp.read()  
print (data)
```

## Sending Data

If you define `data` argument to the `Request` constructor, a POST request will be sent to the server. The data should be any object represented in bytes.

## Example

```
from urllib.request import Request, urlopen  
from urllib.parse import urlencode  
  
values = {'name': 'Madhu',  
          'location': 'India',  
          'language': 'Hindi' }  
data = urlencode(values).encode('utf-8')  
obj = Request("https://example.com", data)
```

## Sending Headers

The Request constructor also accepts header argument to push header information into the request. It should be in a dictionary object.

```
headers = {'User-Agent': user_agent}
obj = Request("https://example.com", data, headers)
```

## The urllib.error Module

Following exceptions are defined in urllib.error module –

### URLLError

URLLError is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute.

```
from urllib.request import Request, urlopen
import urllib.error as err

obj = Request("http://www.nosuchserver.com")
try:
    urlopen(obj)
except err.URLError as e:
    print(e)
```

It will produce the following **output** –

```
HTTP Error 403: Forbidden
```

### HTTPError

Every time the server sends a HTTP response it is associated with a numeric "status code". It code indicates why the server is unable to fulfil the request. The default handlers will handle some of these responses for you. For those it can't handle, urlopen() function raises an HTTPError. Typical examples of HTTPErrors are '404' (page not found), '403' (request forbidden), and '401' (authentication required).

```
from urllib.request import Request, urlopen
import urllib.error as err
```

```
obj = Request("http://www.python.org/fish.html")
try:
    urlopen(obj)
except err.HTTPError as e:
    print(e.code)
```

It will produce the following **output** –

404

## Python - Generics

In Python, generics is a mechanism with which you can define functions, classes, or methods that can operate on multiple types while maintaining type safety. With the implementation of Generics enable it is possible to write reusable code that can be used with different data types. It ensures promoting code flexibility and type correctness.

Generics in Python are implemented using type hints. This feature was introduced in Python with version 3.5 onwards.

Normally, you don't need to declare a variable type. The type is determined dynamically by the value assigned to it. Python's interpreter doesn't perform type checks and hence it may raise runtime exceptions.

Python's new type hinting feature helps in prompting the user with the expected type of the parameters to be passed.

Type hints allow you to specify the expected types of variables, function arguments, and return values. Generics extend this capability by introducing type variables, which represent generic types that can be replaced with specific types when using the generic function or class.

### Example 1

Let us have a look at the following example that defines a generic function –

```
from typing import List, TypeVar, Generic
T = TypeVar('T')
def reverse(items: List[T]) -> List[T]:
    return items[::-1]
```

Here, we define a generic function called 'reverse'. The function takes a list ('List[T]') as an argument and returns a list of the same type. The type variable 'T' represents the generic type, which will be replaced with a specific type when the function is used.

## Example 2

The function `reverse()` function is called with different data types –

```
numbers = [1, 2, 3, 4, 5]
reversed_numbers = reverse(numbers)
print(reversed_numbers)

fruits = ['apple', 'banana', 'cherry']
reversed_fruits = reverse(fruits)
print(reversed_fruits)
```

It will produce the following **output** –

```
[5, 4, 3, 2, 1]
['cherry', 'banana', 'apple']
```

## Example 3

The following example uses generics with a generic class –

```
from typing import List, TypeVar, Generic
T = TypeVar('T')
class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item
    def get_item(self) -> T:
        return self.item

Let us create objects of the above generic class with int and str type
box1 = Box(42)
print(box1.get_item())

box2 = Box('Hello')
print(box2.get_item())
```

It will produce the following **output** –

```
42
Hello
```

# Python - Date and Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Following modules in Python's standard library handle date and time related processing –

- `DateTime` module
- `Time` module
- `Calendar` module

## What are Tick Intervals

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular **time** module available in Python, which provides functions for working with times, and for converting between representations. The function **time.time()** returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

## Example

```
</>
```

Open Compiler

```
import time # This is required to include time module.  
ticks = time.time()  
print ("Number of ticks since 12:00am, January 1, 1970:", ticks)
```

This would produce a result something as follows –

```
Number of ticks since 12:00am, January 1, 1970: 1681928297.5316436
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

## What is TimeTuple?

Many of the Python's time functions handle time as a tuple of 9 numbers, as shown below

–

| Index | Field            | Values                                    |
|-------|------------------|-------------------------------------------|
| 0     | 4-digit year     | 2016                                      |
| 1     | Month            | 1 to 12                                   |
| 2     | Day              | 1 to 31                                   |
| 3     | Hour             | 0 to 23                                   |
| 4     | Minute           | 0 to 59                                   |
| 5     | Second           | 0 to 61 (60 or 61 are leap-seconds)       |
| 6     | Day of Week      | 0 to 6 (0 is Monday)                      |
| 7     | Day of year      | 1 to 366 (Julian day)                     |
| 8     | Daylight savings | -1, 0, 1, -1 means library determines DST |

**For example,**

```
>>>import time
>>> print (time.localtime())
```

This would produce an **output** as follows –

```
time.struct_time(tm_year=2023, tm_mon=4, tm_mday=19, tm_hour=23, tm_min=49, tm_sec=49, tm_wday=5, tm_yday=118, tm_isdst=0)
```

The above tuple is equivalent to struct\_time structure. This structure has the following attributes –

| Index | Attributes | Values                              |
|-------|------------|-------------------------------------|
| 0     | tm_year    | 2016                                |
| 1     | tm_mon     | 1 to 12                             |
| 2     | tm_mday    | 1 to 31                             |
| 3     | tm_hour    | 0 to 23                             |
| 4     | tm_min     | 0 to 59                             |
| 5     | tm_sec     | 0 to 61 (60 or 61 are leap-seconds) |

|   |          |                                           |
|---|----------|-------------------------------------------|
| 6 | tm_wday  | 0 to 6 (0 is Monday)                      |
| 7 | tm_yday  | 1 to 366 (Julian day)                     |
| 8 | tm_isdst | -1, 0, 1, -1 means library determines DST |

## Getting the Current Time

To translate a time instant from **seconds** since the epoch floating-point value into a time-tuple, pass the floating-point value to a function (e.g., `localtime`) that returns a time-tuple with all valid nine items.

```
import time
localtime = time.localtime(time.time())
print ("Local current time : ", localtime)
```

This would produce the following result, which could be formatted in any other presentable form –

Local current time : time.struct\_time(tm\_year=2023, tm\_mon=4, tm\_mday=19, tm\_hour=23, tm\_min=45, tm\_sec=27, tm\_wday=6, tm\_yday=111, tm\_isdst=0)

## Getting the Formatted Time

You can format any time as per your requirement, but a simple method to get time in a readable format is **asctime()** –

</>

Open Compiler

```
import time

localtime = time.asctime( time.localtime(time.time()) )
print ("Local current time : ", localtime)
```

This would produce the following **output** –

Local current time : Wed Apr 19 23:45:27 2023

## Getting the Calendar for a Month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008).

```
</> Open Compiler

import calendar
cal = calendar.month(2023, 4)
print ("Here is the calendar:")
print (cal)
```

This would produce the following **output** –

Here is the calendar:

April 2023

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

## The time Module

There is a popular **time** module available in Python, which provides functions for working with times and for converting between representations. Here is the list of all available methods.

| Sr.No. | Function with Description                                                                                                                                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>time.altzone</b><br>The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero. |
| 2      | <b>time.asctime([tupletime])</b><br>Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.                                                                                                           |
| 3      | <b>time.clock( )</b><br>Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().                              |

|    |                                                                                                                                                                                                                                  |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4  | <b>time.ctime([secs])</b><br>Like asctime(localtime(secs)) and without arguments is like asctime( )                                                                                                                              |
| 5  | <b>time.gmtime([secs])</b><br>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0                                                                |
| 6  | <b>time.localtime([secs])</b><br>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules). |
| 7  | <b>time.mktime(tupletime)</b><br>Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.                                            |
| 8  | <b>time.sleep(secs)</b><br>Suspends the calling thread for secs seconds.                                                                                                                                                         |
| 9  | <b>time.strftime(fmt[,tupletime])</b><br>Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.                                                    |
| 10 | <b>time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')</b><br>Parses str according to format string fmt and returns the instant in time-tuple format.                                                                                  |
| 11 | <b>time.time()</b><br>Returns the current time instant, a floating-point number of seconds since the epoch.                                                                                                                      |
| 12 | <b>time.tzset()</b><br>Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.                                                                                    |

Let us go through the functions briefly.

There are two important attributes available with time module. They are –

| Sr.No. | Attribute with Description                                                                                                                                                        |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>time.timezone</b><br>Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa). |
| 2      | <b>time.tzname</b>                                                                                                                                                                |

Attribute `time.tzname` is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

## The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call the **`calendar.setfirstweekday()`** function.

Here is a list of functions available with the **`calendar`** module –

| Sr.No. | Function with Description                                                                                                                                                                                                                                                                          |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b><code>calendar.calendar(year,w=2,l=1,c=6)</code></b><br>Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length $21*w+18+2*c$ . l is the number of lines for each week.    |
| 2      | <b><code>calendar.firstweekday( )</code></b><br>Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.                                                                                                         |
| 3      | <b><code>calendar.isleap(year)</code></b><br>Returns True if year is a leap year; otherwise, False.                                                                                                                                                                                                |
| 4      | <b><code>calendar.leapdays(y1,y2)</code></b><br>Returns the total number of leap days in the years within range(y1,y2).                                                                                                                                                                            |
| 5      | <b><code>calendar.month(year,month,w=2,l=1)</code></b><br>Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length $7*w+6$ . l is the number of lines for each week.      |
| 6      | <b><code>calendar.monthcalendar(year,month)</code></b><br>Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.                                                         |
| 7      | <b><code>calendar.monthrange(year,month)</code></b><br>Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12. |

|    |                                                                                                                                                                                                |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8  | <b>calendar.prcal(year,w=2,l=1,c=6)</b><br>Like print calendar.calendar(year,w,l,c).                                                                                                           |
| 9  | <b>calendar.prmonth(year,month,w=2,l=1)</b><br>Like print calendar.month(year,month,w,l).                                                                                                      |
| 10 | <b>calendar.setfirstweekday(weekday)</b><br>Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).                                               |
| 11 | <b>calendar.timegm(tupletime)</b><br>The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch. |
| 12 | <b>calendar.weekday(year,month,day)</b><br>Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).            |

## datetime module

Python's datetime module is included in the standard library. It consists of classes that help manipulate data and time data and perform date time arithmetic.

Objects of datetime classes are either aware or naïve. If the object includes timezone information it is aware, and if not it is classified as naïve. An object of date class is naïve, whereas time and datetime objects are aware.

### date

A date object represents a date with year, month, and day. The current Gregorian calendar is indefinitely extended in both directions.

### Syntax

```
datetime.date(year, month, day)
```

Arguments must be integers, in the following ranges –

- **year** – MINYEAR <= year <= MAXYEAR
- **month** – 1 <= month <= 12
- **day** – 1 <= day <= number of days in the given month and year

If the value of any argument outside those ranges is given, `ValueError` is raised.

## Example

&lt;/&gt;

Open Compiler

```
from datetime import date
date1 = date(2023, 4, 19)
print("Date:", date1)
date2 = date(2023, 4, 31)
```

It will produce the following **output** –

```
Date: 2023-04-19
Traceback (most recent call last):
File "C:\Python311\hello.py", line 8, in <module>
    date2 = date(2023, 4, 31)
ValueError: day is out of range for month
```

## date class attributes

- **date.min** – The earliest representable date, `date(MINYEAR, 1, 1)`.
- **date.max** – The latest representable date, `date(MAXYEAR, 12, 31)`.
- **date.resolution** – The smallest possible difference between non-equal date objects.
- **date.year** – Between MINYEAR and MAXYEAR inclusive.
- **date.month** – Between 1 and 12 inclusive.
- **date.day** – Between 1 and the number of days in the given month of the given year.

## Example

&lt;/&gt;

Open Compiler

```
from datetime import date

# Getting min date
mindate = date.min
```

```
print("Minimum Date:", mindate)

# Getting max date
maxdate = date.max
print("Maximum Date:", maxdate)

Date1 = date(2023, 4, 20)
print("Year:", Date1.year)
print("Month:", Date1.month)
print("Day:", Date1.day)
```

It will produce the following **output** –

```
Minimum Date: 0001-01-01
Maximum Date: 9999-12-31
Year: 2023
Month: 4
Day: 20
```

## Classmethods in date class

- **today()** – Return the current local date.
- **fromtimestamp(timestamp)** – Return the local date corresponding to the POSIX timestamp, such as is returned by time.time().
- **fromordinal(ordinal)** – Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.
- **fromisoformat(date\_string)** – Return a date corresponding to a date\_string given in any valid ISO 8601 format, except ordinal dates

## Example

```
from datetime import date

print (date.today())
d1=date.fromisoformat('2023-04-20')
print (d1)
d2=date.fromisoformat('20230420')
print (d2)
```

```
d3=date.fromisoformat('2023-W16-4')
print (d3)
```

It will produce the following **output** –

```
2023-04-20
2023-04-20
2023-04-20
2023-04-20
```

## Instance methods in date class

- **replace()** – Return a date by replacing specified attributes with new values by keyword arguments are specified.
- **timetuple()** – Return a time.struct\_time such as returned by time.localtime().
- **toordinal()** – Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, date.fromordinal(d.toordinal()) == d.
- **weekday()** – Return the day of the week as an integer, where Monday is 0 and Sunday is 6.
- **isoweekday()** – Return the day of the week as an integer, where Monday is 1 and Sunday is 7.
- **isocalendar()** – Return a named tuple object with three components: year, week and weekday.
- **isoformat()** – Return a string representing the date in ISO 8601 format, YYYY-MM-DD:
- **\_\_str\_\_()** – For a date d, str(d) is equivalent to d.isoformat()
- **ctime()** – Return a string representing the date:
- **strftime(format)** – Return a string representing the date, controlled by an explicit format string.
- **\_\_format\_\_(format)** – Same as date.strftime().

## Example

```
</>   Open Compiler

from datetime import date
d = date.fromordinal(738630) # 738630th day after 1. 1. 0001
```

```
print (d)
print (d.timetuple())
# Methods related to formatting string output
print (d.isoformat())
print (d.strftime("%d/%m/%y"))
print (d.strftime("%A %d. %B %Y"))
print (dctime())

print ('The {1} is {0:%d}, the {2} is {0:%B}'.format(d, "day", "month"))

# Methods for extracting 'components' under different calendars
t = d.timetuple()
for i in t:
    print(i)

ic = d.isocalendar()
for i in ic:
    print(i)

# A date object is immutable; all operations produce a new object
print (d.replace(month=5))
```

It will produce the following **output** –

```
2023-04-20
time.struct_time(tm_year=2023, tm_mon=4, tm_mday=20, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=4, tm_yday=104, tm_isdst=0)
2023-04-20
20/04/23
Thursday 20. April 2023
Thu Apr 20 00:00:00 2023
The day is 20, the month is April.
2023
4
20
0
0
0
3
110
-1
2023
16
```

4

2023-05-20

## time

An object time class represents the local time of the day. It is independent of any particular day. If the object contains the tzinfo details, it is the aware object. If it is None then the time object is the naive object.

### Syntax

```
datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments must be integers in the following ranges –

- **hour** –  $0 \leq \text{hour} < 24$ ,
- **minute** –  $0 \leq \text{minute} < 60$ ,
- **second** –  $0 \leq \text{second} < 60$ ,
- **microsecond** –  $0 \leq \text{microsecond} < 1000000$

If any of the arguments are outside those ranges is given, ValueError is raised.

### Example

```
</> Open Compiler  
  
from datetime import time  
  
time1 = time(8, 14, 36)  
print("Time:", time1)  
  
time2 = time(minute = 12)  
print("time", time2)  
  
time3 = time()  
print("time", time3)  
  
time4 = time(hour = 26)
```

It will produce the following **output** –

```
Time: 08:14:36
time 00:12:00
time 00:00:00
Traceback (most recent call last):
  File "/home/cg/root/64b912f27faef/main.py", line 12, in
    time4 = time(hour = 26)
ValueError: hour must be in 0..23
```

## Class attributes

- **time.min** – The earliest representable time, time(0, 0, 0, 0).
- **time.max** – The latest representable time, time(23, 59, 59, 999999).
- **time.resolution** – The smallest possible difference between non-equal time objects.

## Example

```
</>
```

Open Compiler

```
from datetime import time
print(time.min)
print(time.max)
print (time.resolution)
```

It will produce the following **output** –

```
00:00:00
23:59:59.999999
0:00:00.000001
```

## Instance attributes

- **time.hour** – In range(24)
- **time.minute** – In range(60)
- **time.second** – In range(60)

- **time.microsecond** – In range(1000000)
- **time.tzinfo** – the tzinfo argument to the time constructor, or None.

## Example

&lt;/&gt;

Open Compiler

```
from datetime import time
t = time(8,23,45,5000)
print(t.hour)
print(t.minute)
print (t.second)
print (t.microsecond)
```

It will produce the following **output** –

```
8
23
455000
```

## Instance methods

- **replace()** – Return a time with the same value, except for those attributes given new values by whichever keyword arguments are specified.
- **isoformat()** – Return a string representing the time in ISO 8601 format
- **\_\_str\_\_()** – For a time t, str(t) is equivalent to t.isoformat().
- **strftime(format)** – Return a string representing the time, controlled by an explicit format string.
- **\_\_format\_\_(format)** – Same as time.strftime().
- **utcoffset()** – If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None),
- **dst()** – If tzinfo is None, returns None, else returns self.tzinfo.dst(None),
- **tzname()** – If tzinfo is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception

## datetime

An object of datetime class contains the information of date and time together. It assumes the current Gregorian calendar extended in both directions; like a time object, and there are exactly  $3600 \times 24$  seconds in every day.

## Syntax

```
datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

The year, month and day arguments are required.

- **year** – `MINYEAR <= year <= MAXYEAR`,
- **month** – `1 <= month <= 12`,
- **day** – `1 <= day <= number of days in the given month and year`,
- **hour** – `0 <= hour < 24`,
- **minute** – `0 <= minute < 60`,
- **second** – `0 <= second < 60`,
- **microsecond** – `0 <= microsecond < 1000000`,
- **tzinfo** – in `[0, 1]`.

If any argument in outside ranges is given, `ValueError` is raised.

## Example

```
</>
```

Open Compiler

```
from datetime import datetime
dt = datetime(2023, 4, 20)
print(dt)

dt = datetime(2023, 4, 20, 11, 6, 32, 5000)
print(dt)
```

It will produce the following **output** –

```
2023-04-20 00:00:00
2023-04-20 11:06:32.005000
```

## Class attributes

- **datetime.min** – The earliest representable datetime, datetime(MINYEAR, 1, 1, tzinfo=None).
- **datetime.max** – The latest representable datetime, datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None).
- **datetime.resolution** – The smallest possible difference between non-equal datetime objects, timedelta(microseconds=1).

## Example

&lt;/&gt;

Open Compiler

```
from datetime import datetime
min = datetime.min
print("Min DateTime ", min)

max = datetime.max
print("Max DateTime ", max)
```

It will produce the following **output** –

```
Min DateTime 0001-01-01 00:00:00
Max DateTime 9999-12-31 23:59:59.999999
```

## Instance Attributes

- **datetime.year** – Between MINYEAR and MAXYEAR inclusive.
- **datetime.month** – Between 1 and 12 inclusive.
- **datetime.day** – Between 1 and the number of days in the given month of the given year.
- **datetime.hour** – In range(24)
- **datetime.minute** – In range(60)
- **datetime.second** – In range(60)
- **datetime.microsecond** – In range(1000000).
- **datetime.tzinfo** – The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.
- **datetime.fold** – In [0, 1]. Used to disambiguate wall times during a repeated interval.

## Example

&lt;/&gt;

Open Compiler

```
from datetime import datetime
dt = datetime.now()

print("Day: ", dt.day)
print("Month: ", dt.month)
print("Year: ", dt.year)
print("Hour: ", dt.hour)
print("Minute: ", dt.minute)
print("Second: ", dt.second)
```

It will produce the following **output** –

Day: 20  
Month: 4  
Year: 2023  
Hour: 15  
Minute: 5  
Second: 52

## Class Methods

- **today()** – Return the current local datetime, with tzinfo None.
- **now(tz=None)** – Return the current local date and time.
- **utcnow()** – Return the current UTC date and time, with tzinfo None.
- **utcfromtimestamp(timestamp)** – Return the UTC datetime corresponding to the POSIX timestamp, with tzinfo None
- **fromtimestamp(timestamp, timezone.utc)** – On the POSIX compliant platforms, it is equivalent to datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
- **fromordinal(ordinal)** – Return the datetime corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.
- **fromisoformat(date\_string)** – Return a datetime corresponding to a date\_string in any valid ISO 8601 format.

## Instance Methods

- **date()** – Return date object with same year, month and day.
- **time()** – Return time object with same hour, minute, second, microsecond and fold.
- **timetz()** – Return time object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method time().
- **replace()** – Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified.
- **astimezone(tz=None)** – Return a datetime object with new tzinfo attribute tz
- **utcoffset()** – If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(self)
- **dst()** – If tzinfo is None, returns None, else returns self.tzinfo.dst(self)
- **tzname()** – If tzinfo is None, returns None, else returns self.tzinfo.tzname(self)
- **timetuple()** – Return a time.struct\_time such as returned by time.localtime().
- **atime.toordinal()** – Return the proleptic Gregorian ordinal of the date.
- **timestamp()** – Return POSIX timestamp corresponding to the datetime instance.
- **isoweekday()** – Return day of the week as an integer, where Monday is 1, Sunday is 7.
- **isocalendar()** – Return a named tuple with three components: year, week and weekday.
- **isoformat(sep='T', timespec='auto')** – Return a string representing the date and time in ISO 8601 format
- **\_\_str\_\_()** – For a datetime instance d, str(d) is equivalent to d.isoformat(' ').

- **ctime()** – Return a string representing the date and time:
- **strftime(format)** – Return a string representing the date and time, controlled by an explicit format string.
- **\_\_format\_\_(format)** – Same as strftime().

## Example

&lt;/&gt;

Open Compiler

```
from datetime import datetime, date, time, timezone

# Using datetime.combine()
d = date(2022, 4, 20)
t = time(12, 30)
datetime.combine(d, t)

# Using datetime.now()
d = datetime.now()
print (d)

# Using datetime.strptime()
dt = datetime.strptime("23/04/20 16:30", "%d/%m/%y %H:%M")

# Using datetime.timetuple() to get tuple of all attributes
tt = dt.timetuple()
for it in tt:
    print(it)

# Date in ISO format
ic = dt.isocalendar()
for it in ic:
    print(it)
```

It will produce the following **output** –

```
2023-04-20 15:12:49.816343
2020
4
23
16
```

```
30  
0  
3  
114  
-1  
2020  
17  
4
```

## timedelta

The timedelta object represents the duration between two dates or two time objects.

### Syntax

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0)
```

Internally, the attributes are stored in days, seconds and microseconds. Other arguments are converted to those units –

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

While days, seconds and microseconds are then normalized so that the representation is unique.

### Example

The following example shows that Python internally stores days, seconds and microseconds only.

```
</>
```

Open Compiler

```
from datetime import timedelta  
delta = timedelta(  
    days=100,  
    seconds=27,  
    microseconds=10,
```

```
milliseconds=29000,  
minutes=5,  
hours=12,  
weeks=2  
)  
# Only days, seconds, and microseconds remain  
print(delta)
```

It will produce the following **output** –

```
114 days, 12:05:56.000010
```

## Example

The following example shows how to add timedelta object to a datetime object.

```
</>
```

Open Compiler

```
from datetime import datetime, timedelta  
  
date1 = datetime.now()  
  
date2= date1+timedelta(days = 4)  
print("Date after 4 days:", date2)  
  
date3 = date1-timedelta(15)  
print("Date before 15 days:", date3)
```

It will produce the following **output** –

```
Date after 4 days: 2023-04-24 18:05:39.509905  
Date before 15 days: 2023-04-05 18:05:39.509905
```

## Class Attributes

- **timedelta.min** – The most negative timedelta object, timedelta(-999999999).
- **timedelta.max** – The most positive timedelta object, timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999).

- **timedelta.resolution** – The smallest possible difference between non-equal timedelta objects, timedelta(microseconds=1)

## Example

```
</> Open Compiler  
  
from datetime import timedelta  
  
# Getting minimum value  
min = timedelta.min  
print("Minimum value:", min)  
  
max = timedelta.max  
print("Maximum value", max)
```

It will produce the following **output** –

```
Minimum value: -999999999 days, 0:00:00  
Maximum value 999999999 days, 23:59:59.999999
```

## Instance Attributes

Since only day, second and microseconds are stored internally, those are the only instance attributes for a **timedelta** object.

- **days** – Between -999999999 and 999999999 inclusive
- **seconds** – Between 0 and 86399 inclusive
- **microseconds** – Between 0 and 999999 inclusive

## Instance Methods

**timedelta.total\_seconds()** – Return the total number of seconds contained in the duration.

## Example

```
</>
```

Open Compiler

```

from datetime import timedelta
year = timedelta(days=365)
years = 5 * year
print (years)
print (years.days // 365)
646
year_1 = years // 5
print(year_1.days)

```

It will produce the following **output** –

```

1825 days, 0:00:00
5
365

```

## Python - Maths

Python's standard library provides math module. This module includes many pre-defined functions for performing different mathematical operations. These functions do not work with complex numbers. There is a cmath module contains mathematical functions for complex numbers.

### Functions in Math Module

Here is the list of functions available in the math module –

| Sr.No | Method & Description                                           |
|-------|----------------------------------------------------------------|
| 1     | <b>acos (x)</b><br>Return the arc cosine of x, in radians.     |
| 2     | <b>acosh (x)</b><br>Return the inverse hyperbolic cosine of x. |
| 3     | <b>asin</b><br>Return the arc sine of x, in radians.           |
| 4     | <b>asinh (x)</b><br>Return the inverse hyperbolic sine of x.   |
| 5     | <b>atan</b><br>Return the arc tangent of x, in radians.        |

|    |                                                                                                            |
|----|------------------------------------------------------------------------------------------------------------|
| 6  | <b>atan2</b><br>Return atan(y/x), in radians.                                                              |
| 7  | <b>atanh (x)</b><br>Return the inverse hyperbolic tangent of x.                                            |
| 8  | <b>cbrt (x)</b><br>Return the cube root of x.                                                              |
| 9  | <b>ceil(x)</b><br>The ceiling of x: the smallest integer not less than x.                                  |
| 10 | <b>comb (x,y)</b><br>Return the number of ways to choose x items from y iter repetition and without order. |
| 11 | <b>copysign(x,y)</b><br>Return a float with the magnitude of x but the sign of y.                          |
| 12 | <b>cos (x)</b><br>Return the cosine of x radians.                                                          |
| 13 | <b>cosh (x)</b><br>Return the hyperbolic cosine of x.                                                      |
| 14 | <b>degrees</b><br>Converts angle x from radians to degrees.                                                |
| 15 | <b>dist (x,y)</b><br>Return the Euclidean distance between two points x and y.                             |
| 16 | <b>e</b><br>The mathematical constant e = 2.718281..., to available precision.                             |
| 17 | <b>erf (x)</b><br>Return the error function at x.                                                          |
| 18 | <b>erfc (x)</b><br>Return the complementary error function at x.                                           |
| 19 | <b>exp (x)</b><br>Return e raised to the power x, where e = 2.718281...                                    |
| 20 | <b>exp2 (x)</b><br>Return 2 raised to the power x.                                                         |
| 21 | <b>expm1 (x)</b><br>Return e raised to the power x, minus 1.                                               |

|    |                                                                                                           |
|----|-----------------------------------------------------------------------------------------------------------|
| 22 | <b>fabs(x)</b><br>The absolute value of x in float                                                        |
| 23 | <b>factorial(x)</b><br>Return x factorial as an Integer.                                                  |
| 24 | <b>floor (x)</b><br>The floor of x: the largest integer not greater than x.                               |
| 25 | <b>fmod (x,y)</b><br>Always returns float, similar to $x\%y$                                              |
| 26 | <b>frexp (x)</b><br>Returns the mantissa and exponent for a given number x.                               |
| 27 | <b>fsum (iterable)</b><br>Sum of all numbers in any iterable, returns float.                              |
| 28 | <b>gamma (x)</b><br>Return the Gamma function at x..                                                      |
| 29 | <b>gcd (x,y,z)</b><br>Return the greatest common divisor of the specified integer arguments.              |
| 30 | <b>hypot</b><br>Return the Euclidean norm, $\sqrt{x^2 + y^2}$ .                                           |
| 31 | <b>inf</b><br>A floating-point positive infinity. Equivalent to the output of <code>float('inf')</code> . |
| 32 | <b>isclose (x,y)</b><br>Return True if the values x and y are close to each other and False otherwise.    |
| 33 | <b>isfinite (x)</b><br>Returns True if neither an infinity nor a NaN, and False otherwise.                |
| 34 | <b>isinf (x)</b><br>Return True if x is a positive or negative infinity, and False otherwise.             |
| 35 | <b>isnan (x)</b><br>Return True if x is a NaN (not a number), and False otherwise.                        |
| 36 | <b>isqrt (x)</b><br>Return the integer square root of the nonnegative integer x                           |
| 37 | <b>lcm (x1, x2, ..)</b><br>Return the least common multiple of the specified integer arguments.           |

|    |                                                                                                                                                            |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 38 | <b>ldexp (x,y)</b><br>Return $x * (2^{**y})$ . This is the inverse of function frexp().                                                                    |
| 39 | <b>lgamma (x)</b><br>Return the natural logarithm of the absolute value of the Gamma function at x.                                                        |
| 40 | <b>log (x)</b><br>Return the natural logarithm of x (to base e).                                                                                           |
| 41 | <b>log10 (x)</b><br>Return the base-10 logarithm of x.                                                                                                     |
| 42 | <b>log1p (x)</b><br>Return the natural logarithm of $1+x$ (base e).                                                                                        |
| 43 | <b>log2 (x)</b><br>Return the base-2 logarithm of x.                                                                                                       |
| 44 | <b>modf (x)</b><br>The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float. |
| 45 | <b>nan</b><br>A floating-point "not a number" (NaN) value.                                                                                                 |
| 46 | <b>nextafter (x,y)</b><br>Return the next floating-point value after x towards y.                                                                          |
| 47 | <b>perm (x,y)</b><br>Return the number of ways to choose x items from y items without repetition and with order.                                           |
| 48 | <b>pi</b><br>The mathematical constant $\pi = 3.141592\dots$ , to available precision.                                                                     |
| 49 | <b>pow (x,y)</b><br>Returns x raised to y                                                                                                                  |
| 50 | <b>prod (iterable)</b><br>Return the product of all the elements in the input iterable.                                                                    |
| 51 | <b>radians</b><br>Converts angle x from degrees to radians.                                                                                                |
| 52 | <b>remainder (x,y)</b><br>Returns the remainder of x with respect to y                                                                                     |

|    |                                                                                          |
|----|------------------------------------------------------------------------------------------|
| 53 | <b>sin (x)</b><br>Return the sine of x radians.                                          |
| 54 | <b>sinh (x)</b><br>Return the inverse hyperbolic sine of x.                              |
| 55 | <b>sqrt (x)</b><br>Return the square root of x.                                          |
| 56 | <b>tan (x)</b><br>Return the tangent of x radians.                                       |
| 57 | <b>tanh (x)</b><br>Return the hyperbolic tangent of x.                                   |
| 58 | <b>tau</b><br>The mathematical constant $\tau = 6.283185\dots$ , to available precision. |
| 59 | <b>trunc (x)</b><br>Return x with the fractional part removed, leaving the integer part. |
| 60 | <b>ulp</b><br>Return the value of the least significant bit of the float x.              |

These functions can be classified in following categories –

- Theoretic and Representation functions
- Power and Logarithmic functions
- Trigonometric functions
- Angular Conversion functions
- Hyperbolic functions
- Special functions
- Mathematical constants

## Python - Iterators

Iterator in Python is an object representing a stream of data. It follows iterator protocol which requires it to support `__iter__()` and `__next__()` methods. Python's built-in method `iter()` implements `__iter__()` method. It receives an iterable and returns iterator object. The built-in `next()` function internally calls to the iterator's `__next__()` method returns successive items in the stream. When no more data are available a `StopIteration` exception is raised.

Python uses iterators implicitly while working with collection data types such as list, tuple or string. That's why these data types are called iterables. We normally use for loop to iterate through an iterable as follows –

```
for element in sequence:  
    print (element)
```

Python's built-in method `iter()` implements `__iter__()` method. It receives an iterable and returns iterator object.

## Example

Following code obtains iterator object from sequence types list, string and tuple. The `iter()` function also returns keyiterator from dictionary. However, int is not iterable, hence it produces `TypeError`.

</>

Open Compiler

```
print (iter("aa"))  
print (iter([1,2,3]))  
print (iter((1,2,3)))  
print (iter({}))  
print (iter(100))
```

It will produce the following **output** –

```
<str_ascii_iterator object at 0x000001BB03FFAB60>  
<list_iterator object at 0x000001BB03FFAB60>  
<tuple_iterator object at 0x000001BB03FFAB60>  
<dict_keyiterator object at 0x000001BB04181670>  
Traceback (most recent call last):  
  File "C:\Users\user\example.py", line 5, in <module>  
    print (iter(100))  
    ^^^^^^^^^^  
TypeError: 'int' object is not iterable
```

Iterator object has `__next__()` method. Every time it is called, it returns next element in iterator stream. When the stream gets exhausted, `StopIteration` error is raised. Call to `next()` function is equivalent to calling `__next__()` method of iterator object.

## Example

```
</>  
it = iter([1,2,3])  
print (next(it))  
print (it.__next__())  
print (it.__next__())  
print (next(it))
```

It will produce the following **output** –

```
1  
2  
3
```

Traceback (most recent call last):

```
  File "C:\Users\user\example.py", line 5, in <module>  
    print (next(it))  
           ^^^^^^^^^^
```

```
StopIteration
```

## Example

You can use exception handling mechanism to catch StopIteration.

```
</>  
it = iter([1,2,3, 4, 5])  
print (next(it))  
while True:  
    try:  
        no = next(it)  
        print (no)  
    except StopIteration:  
        break
```

It will produce the following **output** –

```
1  
2  
3
```

4

5

To define a custom iterator class in Python, the class must define `__iter__()` and `__next__()` methods.

In the following example, the `Oddnumbers` is a class implementing `__iter__()` and `__next__()` methods. On every call to `__next__()`, the number increments by 2, thereby streaming odd numbers in the range 1 to 10.

## Example

&lt;/&gt;

Open Compiler

```
class Oddnumbers:

    def __init__(self, end_range):
        self.start = -1
        self.end = end_range

    def __iter__(self):
        return self

    def __next__(self):
        if self.start < self.end-1:
            self.start += 2
            return self.start
        else:
            raise StopIteration

countiter = Oddnumbers(10)
while True:
    try:
        no = next(countiter)
        print (no)
    except StopIteration:
        break
```

It will produce the following **output** –

1

3

5  
7  
9

## Asynchronous Iterator

Two built-in functions `aiter()` and `anext()` have been added in Python 3.10 version onwards. The `aiter()` function returns an asynchronous iterator object. It is an asynchronous counter part of the classical iterator. Any asynchronous iterator must support `__aiter__()` and `__anext__()` methods. These methods are internally called by the two built-in functions.

Like the classical iterator, the asynchronous iterator gives a stream of objects. When the stream is exhausted, the `StopAsyncIteration` exception is raised.

In the example give below, an asynchronous iterator class `Oddnumbers` is declared. It implements `__aiter__()` and `__anext__()` method. On each iteration, a next odd number is returned, and the program waits for one second, so that it can perform any other process asynchronously.

Unlike regular functions, asynchronous functions are called coroutines and are executed with `asyncio.run()` method. The `main()` coroutine contains a while loop that successively obtains odd numbers and raises `StopAsyncIteration` if the number exceeds 9.

### Example

</>

Open Compiler

```
import asyncio

class Oddnumbers():
    def __init__(self):
        self.start = -1

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self.start >= 9:
            raise StopAsyncIteration
        self.start += 2
        await asyncio.sleep(1)
        return self.start
```

```
async def main():
    it = Oddnumbers()
    while True:
        try:
            awaitable = anext(it)
            result = await awaitable
            print(result)
        except StopAsyncIteration:
            break

asyncio.run(main())
```

It will produce the following **output** –

```
1
3
5
7
9
```

## Python - Generators

A generator in Python is a special type of function that returns an iterator object. It appears similar to a normal Python function in that its definition also starts with `def` keyword. However, instead of `return` statement at the end, generator uses the `yield` keyword.

### Syntax

```
def generator():
    ...
    ...
    yield obj
it = generator()
next(it)
...
```

The `return` statement at the end of function indicates that the execution of the function body is over, all the local variables in the function go out of the scope. If the function is called again, the local variables are re-initialized.

Generator function behaves differently. It is invoked for the first time like a normal function, but when its yield statement comes, its execution is temporarily paused, transferring the control back. The yielded result is consumed by the caller. The call to next() built-in function restarts the execution of generator from the point it was paused, and generates the next object for the iterator. The cycle repeats as subsequent yield provides next item in the iterator it is exhausted.

## Example 1

The function in the code below is a generator that successively yield integers from 1 to 5. When called, it returns an iterator. Every call to next() transfers the control back to the generator and fetches next integer.

```
</> Open Compiler  
  
def generator(num):  
    for x in range(1, num+1):  
        yield x  
    return  
  
it = generator(5)  
while True:  
    try:  
        print(next(it))  
    except StopIteration:  
        break
```

It will produce the following **output** –

```
1  
2  
3  
4  
5
```

The generator function returns a dynamic iterator. Hence, it is more memory efficient than a normal iterator that you obtain from a Python sequence object. For example, if you want to get first n numbers in Fibonacci series. You can write a normal function and build a list of Fibonacci numbers, and then iterate the list using a loop.

## Example 2

Given below is the normal function to get a list of Fibonacci numbers –

&lt;/&gt;

[Open Compiler](#)

```
def fibonacci(n):
    fibo = []
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
            break
        fibo.append(c)
        a, b = b, c
    return fibo
f = fibonacci(10)
for i in f:
    print (i)
```

It will produce the following output –

```
1
2
3
5
8
```

The above code collects all Fibonacci series numbers in a list and then the list is traversed using a loop. Imagine that we want Fibonacci series going upto a large number. In which case, all the numbers must be collected in a list requiring huge memory. This is where generator is useful, as it generates a single number in the list and gives it for consumption.

### Example 3

Following code is the generator-based solution for list of Fibonacci numbers –

&lt;/&gt;

[Open Compiler](#)

```
def fibonacci(n):
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
```

```
        break
    yield c
    a, b = b, c
return

f = fibonacci(10)
while True:
    try:
        print(next(f))
    except StopIteration:
        break
```

## Asynchronous Generator

An asynchronous generator is a coroutine that returns an asynchronous iterator. A coroutine is a Python function defined with `async` keyword, and it can schedule and await other coroutines and tasks. Just like a normal generator, the asynchronous generator yields incremental item in the iterator for every call to `anext()` function, instead of `next()` function.

## Syntax

```
async def generator():
    ...
    ...
    yield obj
    it = generator()
    anext(it)
    ...
    ...
```

## Example 4

Following code demonstrates a coroutine generator that yields incrementing integers on every iteration of an **async for** loop.

&lt;/&gt;

Open Compiler

```
import asyncio

async def async_generator(x):
    for i in range(1, x+1):
```

```
await asyncio.sleep(1)
yield i

async def main():
    async for item in async_generator(5):
        print(item)

asyncio.run(main())
```

It will produce the following **output** –

```
1
2
3
4
5
```

## Example 5

Let us now write an asynchronous generator for Fibonacci numbers. To simulate some asynchronous task inside the coroutine, the program calls sleep() method for a duration of 1 second before yielding the next number. As a result, you will get the numbers printed on the screen after a delay of one second.

</>

Open Compiler

```
import asyncio

async def fibonacci(n):
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
            break
        await asyncio.sleep(1)
        yield c
        a, b = b, c
    return

async def main():
    f = fibonacci(10)
```

```
async for num in f:  
    print (num)  
  
asyncio.run(main())
```

It will produce the following **output** –

```
1  
2  
3  
5  
8
```

## Python - Closures

In this chapter, let us discuss the concept of closures in Python. In Python, functions are said to be first order objects. Just like the primary data types, functions can also be used assigned to variables, or passed as arguments.

### Nested Functions

You can also have a nested declaration of functions, wherein a function is defined inside the body of another function.

### Example

```
</>
```

Open Compiler

```
def functionA():  
    print ("Outer function")  
    def functionB():  
        print ("Inner function")  
    functionB()  
  
functionA()
```

It will produce the following **output** –

```
Outer function
```

## Inner function

In the above example, functionB is defined inside functionA. Inner function is then called from inside the outer function's scope.

If the outer function receives any argument, it can be passed to the inner function.

&lt;/&gt;

Open Compiler

```
def functionA(name):
    print ("Outer function")
    def functionB():
        print ("Inner function")
        print ("Hi {}".format(name))
    functionB()

functionA("Python")
```

It will produce the following output –

Outer function

Inner function

Hi Python

## What is a Closure?

A closure is a nested function which has access to a variable from an enclosing function that has finished its execution. Such a variable is not bound in the local scope. To use immutable variables (number or string), we have to use the nonlocal keyword.

The main advantage of Python closures is that we can help avoid the using global values and provide some form of data hiding. They are used in Python decorators.

## Example

&lt;/&gt;

Open Compiler

```
def functionA(name):
    name ="New name"
    def functionB():
        print (name)
```

```
    return functionB

myfunction = functionA("My name")
myfunction()
```

It will produce the following **output** –

New name

In the above example, we have a functionA function, which creates and returns another function functionB. The nested functionB function is the closure.

The outer functionA function returns a functionB function and assigns it to the myfunction variable. Even if it has finished its execution. However, the printer closure still has access to the name variable.

## nonlocal Keyword

In Python, nonlocal keyword allows a variable outside the local scope to be accessed. This is used in a closure to modify an immutable variable present in the scope of outer variable.

## Example

```
</> Open Compiler

def functionA():
    counter = 0
    def functionB():
        nonlocal counter
        counter+=1
        return counter
    return functionB

myfunction = functionA()

retval = myfunction()
print ("Counter:", retval)

retval = myfunction()
print ("Counter:", retval)
```

```
retval = myfunction()
print ("Counter:", retval)
```

It will produce the following **output** –

```
Counter: 1
Counter: 2
Counter: 3
```

## Python - Decorators

A Decorator in Python is a function that receives another function as argument. The argument function is the one to be decorated by decorator. The behaviour of argument function is extended by the decorator without actually modifying it.

In this chapter, we will learn how to use Python decorator.

Function in Python is a first order object. It means that it can be passed as argument to another function just as other data types such as number, string or list etc. It is also possible to define a function inside another function. Such a function is called nested function. Moreover, a function can return other function as well.

### Syntax

The typical definition of a decorator function is as under –

```
def decorator(arg_function): #arg_function to be decorated
    def nested_function():
        #this wraps the arg_function and extends its behaviour
        #call arg_function
        arg_function()
    return nested_function
```

Here a normal Python function –

```
def function():
    print ("hello")
```

You can now decorate this function to extend its behaviour by passing it to decorator –

```
function=decorator(function)
```

If this function is now executed, it will show output extended by decorator.

## Example 1

Following code is a simple example of decorator –

```
</> Open Compiler

def my_function(x):
    print("The number is=",x)

def my_decorator(some_function,num):
    def wrapper(num):
        print("Inside wrapper to check odd/even")
        if num%2 == 0:
            ret= "Even"
        else:
            ret= "Odd!"
        some_function(num)
        return ret
    print ("wrapper function is called")
    return wrapper

no=10
my_function = my_decorator(my_function, no)
print ("It is ",my_function(no))
```

The `my_function()` just prints out the received number. However, its behaviour is modified by passing it to a `my_decorator`. The inner function receives the number and returns whether it is odd/even. Output of above code is –

```
wrapper function is called
Inside wrapper to check odd/even
The number is= 10
It is Even
```

## Example 2

An elegant way to decorate a function is to mention just before its definition, the name of decorator prepended by @ symbol. The above example is re-written using this notation –

```
</>
```

Open Compiler

```

def my_decorator(some_function):
    def wrapper(num):
        print("Inside wrapper to check odd/even")
        if num%2 == 0:
            ret= "Even"
        else:
            ret= "Odd!"
        some_function(num)
        return ret
    print ("wrapper function is called")
    return wrapper

@my_decorator
def my_function(x):
    print("The number is=",x)
no=10
print ("It is ",my_function(no))

```

Python's standard library defines following built-in decorators –

## @classmethod Decorator

The `classmethod` is a built-in function. It transforms a method into a class method. A class method is different from an instance method. Instance method defined in a class is called by its object. The method received an implicit object referred to by `self`. A class method on the other hand implicitly receives the class itself as first argument.

## Syntax

In order to declare a class method, the following notation of decorator is used –

```

class Myclass:
    @classmethod
    def mymethod(cls):
        #...

```

The `@classmethod` form is that of function decorator as described earlier. The `mymethod` receives reference to the class. It can be called by the class as well as its object. That means `Myclass.mymethod` as well as `Myclass().mymethod` both are valid calls.

## Example 3

Let us understand the behaviour of class method with the help of following example –

```

class counter:
    count=0
    def __init__(self):
        print ("init called by ", self)
        counter.count=counter.count+1
        print ("count=",counter.count)
    @classmethod
    def showcount(cls):
        print ("called by ",cls)
        print ("count=",cls.count)

c1=counter()
c2=counter()
print ("class method called by object")
c1.showcount()
print ("class method called by class")
counter.showcount()

```

In the class definition count is a class attribute. The `__init__()` method is the constructor and is obviously an instance method as it received `self` as object reference. Every object declared calls this method and increments count by 1.

The `@classmethod` decorator transforms `showcount()` method into a class method which receives reference to the class as argument even if it is called by its object. It can be seen even when `c1` object calls `showcount`, it displays reference of `counter` class.

It will display the following **output** –

```

init called by <__main__.counter object at 0x000001D32DB4F0F0>
count= 1
init called by <__main__.counter object at 0x000001D32DAC8710>
count= 2
class method called by object
called by <class '__main__.counter'>
count= 2
class method called by class
called by <class '__main__.counter'>

```

## @staticmethod Decorator

The `staticmethod` is also a built-in function in Python standard library. It transforms a method into a static method. Static method doesn't receive any reference argument

whether it is called by instance of class or class itself. Following notation used to declare a static method in a class –

## Syntax

```
class Myclass:  
    @staticmethod  
    def mymethod():  
        #....
```

Even though `Myclass.mymethod` as well as `Myclass().mymethod` both are valid calls, the static method receives reference of neither.

## Example 4

The counter class is modified as under –

```
class counter:  
    count=0  
    def __init__(self):  
        print ("init called by ", self)  
        counter.count=counter.count+1  
        print ("count=",counter.count)  
    @staticmethod  
    def showcount():  
        print ("count=",counter.count)  
  
c1=counter()  
c2=counter()  
print ("class method called by object")  
c1.showcount()  
print ("class method called by class")  
counter.showcount()
```

As before, the class attribute `count` is increment on declaration of each object inside the `__init__()` method. However, since `mymethod()`, being a static method doesn't receive either `self` or `cls` parameter. Hence value of class attribute `count` is displayed with explicit reference to `counter`.

The **output** of the above code is as below –

```
init called by <__main__.counter object at 0x000002512EDCF0B8>  
count= 1
```

```
init called by <__main__.counter object at 0x000002512ED48668>
count= 2
class method called by object
count= 2
class method called by class
count= 2
```

## @property Decorator

Python's `property()` built-in function is an interface for accessing instance variables of a class. The `@property` decorator turns an instance method into a "getter" for a read-only attribute with the same name, and it sets the docstring for the property to "Get the current value of the instance variable."

You can use the following three decorators to define a property –

- `@property` – Declares the method as a property.
- `@<property-name>.setter`: – Specifies the setter method for a property that sets the value to a property.
- `@<property-name>.deleter` – Specifies the delete method as a property that deletes a property.

A property object returned by `property()` function has `getter`, `setter`, and `delete` methods.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

The `fget` argument is the getter method, `fset` is setter method. It optionally can have `fdel` as method to delete the object and `doc` is the documentation string.

The `property()` object's setter and getter may also be assigned with the following syntax also.

```
speed = property()
speed=speed.getter(speed, get_speed)
speed=speed.setter(speed, set_speed)
```

Where `get_speed()` and `set_speeds()` are the instance methods that retrieve and set the value to an instance variable `speed` in `Car` class.

The above statements can be implemented by `@property` decorator. Using the decorator `car` class is re-written as –

```
</>

class car:
    def __init__(self, speed=40):
        self._speed=speed
        return
    @property
    def speed(self):
        return self._speed
    @speed.setter
    def speed(self, speed):
        if speed<0 or speed>100:
            print ("speed limit 0 to 100")
            return
        self._speed=speed
        return

c1=car()
print (c1.speed) #calls getter
c1.speed=60 #calls setter
```

Property decorator is very convenient and recommended method of handling instance attributes.

## Python - Recursion

A function that calls itself is called a recursive function. This method is used when a certain problem is defined in terms of itself. Although this involves iteration, using iterative approach to solve such problem can be tedious. Recursive approach provides a very concise solution to seemingly complex problem.

The most popular example of recursion is calculation of factorial. Mathematically factorial is defined as –

$$n! = n \times (n-1)!$$

It can be seen that we use factorial itself to define factorial. Hence this is a fit case to write a recursive function. Let us expand above definition for calculation of factorial value of 5.

$$\begin{aligned} 5! &= 5 \times 4! \\ &5 \times 4 \times 3! \\ &5 \times 4 \times 3 \times 2! \end{aligned}$$

```
5 × 4 × 3 × 2 × 1!
5 × 4 × 3 × 2 × 1
= 120
```

While we can perform this calculation using a loop, its recursive function involves successively calling it by decrementing the number till it reaches 1.

## Example 1

The following example shows how you can use a recursive function to calculate factorial –

&lt;/&gt;

Open Compiler

```
def factorial(n):

    if n == 1:
        print (n)
        return 1
    else:
        print (n, '*', end=' ')
        return n * factorial(n-1)

print ('factorial of 5=', factorial(5))
```

It will produce the following **output** –

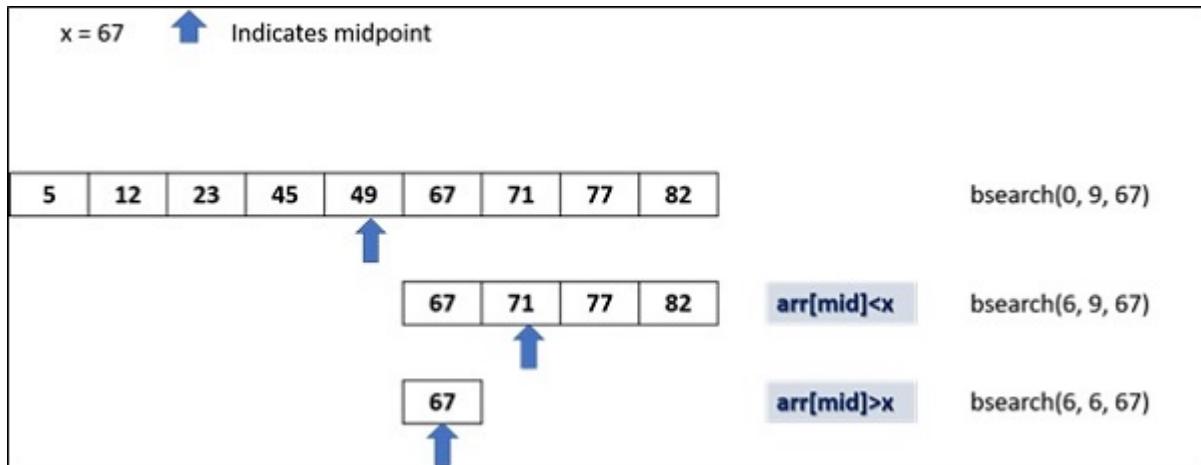
```
5 * 4 * 3 * 2 * 1
factorial of 5= 120
```

Let us have a look at another example to understand how recursion works. The problem at hand is to check whether a given number is present in a list.

While we can perform a sequential search for a certain number in the list using a for loop and comparing each number, the sequential search is not efficient especially if the list is too large. The binary search algorithm that checks if the index 'high' is greater than index 'low'. Based on value present at 'mid' variable, the function is called again to search for the element.

We have a list of numbers, arranged in ascending order. Then we find the midpoint of the list and restrict the checking to either left or right of midpoint depending on whether the desired number is less than or greater than the number at midpoint.

The following diagram shows how binary search works –



## Example 2

The following code implements the recursive binary searching technique –

&lt;/&gt;

Open Compiler

```
def bsearch(my_list, low, high, elem):
    if high >= low:
        mid = (high + low) // 2
        if my_list[mid] == elem:
            return mid
        elif my_list[mid] > elem:
            return bsearch(my_list, low, mid - 1, elem)
        else:
            return bsearch(my_list, mid + 1, high, elem)
    else:
        return -1

my_list = [5,12,23, 45, 49, 67, 71, 77, 82]
num = 67
print("The list is")
print(my_list)
print ("Check for number:", num)
my_result = bsearch(my_list,0,len(my_list)-1,num)

if my_result != -1:
    print("Element found at index ", str(my_result))
else:
    print("Element not found!")
```

It will produce the following **output** –

```
The list is  
[5, 12, 23, 45, 49, 67, 71, 77, 82]  
Check for number: 67  
Element found at index 5
```

You can check the output for different numbers in the list, as well as not in the list.

## Python - Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. A regular expression also known as regex is a sequence of characters that defines a search pattern. Popularly known as as regex or regexp; it is a sequence of characters that specifies a match pattern in text. Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

Large scale text processing in data science projects requires manipulation of textual data. The regular expressions processing is supported by many programming languages including Python. Python's standard library has 're' module for this purpose.

Since most of the functions defined in re module work with raw strings, let us first understand what the raw strings are.

### Raw Strings

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. Python on the other hand uses the same character as escape character. Hence Python uses the raw string notation.

A string become a raw string if it is prefixed with r or R before the quotation symbols. Hence 'Hello' is a normal string were are r'Hello' is a raw string.

```
>>> normal="Hello"  
>>> print (normal)  
Hello  
>>> raw=r"Hello"  
>>> print (raw)  
Hello
```

In normal circumstances, there is no difference between the two. However, when the escape character is embedded in the string, the normal string actually interprets the escape sequence, whereas the raw string doesn't process the escape character.

```
>>> normal="Hello\nWorld"
>>> print (normal)
Hello
World
>>> raw=r"Hello\nWorld"
>>> print (raw)
Hello\nWorld
```

In the above example, when a normal string is printed the escape character '\n' is processed to introduce a newline. However because of the raw string operator 'r' the effect of escape character is not translated as per its meaning.

## Metacharacters

Most letters and characters will simply match themselves. However, some characters are special metacharacters, and don't match themselves. Meta characters are characters having a special meaning, similar to \* in wild card.

Here's a complete list of the metacharacters –

```
. ^ $ * + ? { } [ ] \ | ( )
```

The square bracket symbols[ and ] indicate a set of characters that you wish to match. Characters can be listed individually, or as a range of characters separating them by a '-'.

| Sr.No. | Metacharacters & Description                                                         |
|--------|--------------------------------------------------------------------------------------|
| 1      | [abc]<br>match any of the characters a, b, or c                                      |
| 2      | [a-c]<br>which uses a range to express the same set of characters.                   |
| 3      | [a-z]<br>match only lowercase letters.                                               |
| 4      | [0-9]<br>match only digits.                                                          |
| 5      | '^'<br>complements the character set in [].[^5] will match any character except '5'. |

'\ is an escaping metacharacter. When followed by various characters it forms various special sequences. If you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \[ or \\.

Predefined sets of characters represented by such special sequences beginning with '\' are listed below –

| Sr.No. | Metacharacters & Description                                                                                           |
|--------|------------------------------------------------------------------------------------------------------------------------|
| 1      | \d<br>Matches any decimal digit; this is equivalent to the class [0-9].                                                |
| 2      | \D<br>Matches any non-digit character; this is equivalent to the class [^0-9].                                         |
| 3      | \s<br>Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v].                                  |
| 4      | \S<br>Matches any non-whitespace character; this is equivalent to the class [^\t\n\r\f\v].                             |
| 5      | \w<br>Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_].                                |
| 6      | \W<br>Matches any non-alphanumeric character. equivalent to the class [^a-zA-Z0-9_].                                   |
| 7      | .                                                                                                                      |
|        | Matches with any single character except newline '\n'.                                                                 |
| 8      | ?                                                                                                                      |
|        | match 0 or 1 occurrence of the pattern to its left                                                                     |
| 9      | +                                                                                                                      |
|        | 1 or more occurrences of the pattern to its left                                                                       |
| 10     | *                                                                                                                      |
|        | 0 or more occurrences of the pattern to its left                                                                       |
| 11     | \b<br>boundary between word and non-word and /B is opposite of /b                                                      |
| 12     | [..]<br>Matches any single character in a square bracket and [^..] matches any single character not in square bracket. |

|    |       |                                                                                          |
|----|-------|------------------------------------------------------------------------------------------|
| 13 | \     | It is used for special meaning characters like \. to match a period or \+ for plus sign. |
| 14 | {n,m} | Matches at least n and at most m occurrences of preceding                                |
| 15 | a  b  | Matches either a or b                                                                    |

Python's re module provides useful functions for finding a match, searching for a pattern, and substitute a matched string with other string etc.

## re.match() Function

This function attempts to match RE pattern at the start of string with optional flags.

Here is the **syntax** for this function –

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters –

| Sr.No. | Parameter & Description                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>pattern</b><br>This is the regular expression to be matched.                                                                 |
| 2      | <b>String</b><br>This is the string, which would be searched to match the pattern at the beginning of string.                   |
| 3      | <b>Flags</b><br>You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below. |

The re.match function returns a **match** object on success, **None** on failure. A match object instance contains information about the match: where it starts and ends, the substring it matched, etc.

The match object's start() method returns the starting position of pattern in the string, and end() returns the endpoint.

If the pattern is not found, the match object is None.

We use group(num) or groups() function of **match** object to get matched expression.

| Sr.No. | Match Object Methods & Description                                                                 |
|--------|----------------------------------------------------------------------------------------------------|
| 1      | <b>group(num=0)</b> This method returns entire match (or specific subgroup num)                    |
| 2      | <b>groups()</b> This method returns all matching subgroups in a tuple (empty if there weren't any) |

## Example

&lt;/&gt;

Open Compiler

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'Cats', line)
print (matchObj.start(), matchObj.end())
print ("matchObj.group() : ", matchObj.group())
```

It will produce the following **output** –

```
0 4
matchObj.group() : Cats
```

## re.search() Function

This function searches for first occurrence of RE pattern within the string, with optional flags.

Here is the **syntax** for this function –

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters –

| Sr.No. | Parameter & Description                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------|
| 1      | <b>Pattern</b><br>This is the regular expression to be matched.                                           |
| 2      | <b>String</b><br>This is the string, which would be searched to match the pattern anywhere in the string. |

**Flags**

- 3 You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The `re.search` function returns a **match** object on success, **none** on failure. We use `group(num)` or `groups()` function of **match** object to get the matched expression.

| Sr.No. | Match Object Methods & Description                                                                 |
|--------|----------------------------------------------------------------------------------------------------|
| 1      | <b>group(num=0)</b> This method returns entire match (or specific subgroup num)                    |
| 2      | <b>groups()</b> This method returns all matching subgroups in a tuple (empty if there weren't any) |

## Example

```
</> Open Compiler

import re
line = "Cats are smarter than dogs"
matchObj = re.search( r'than', line)
print (matchObj.start(), matchObj.end())
print ("matchObj.group() : ", matchObj.group())
```

It will produce the following **output** –

```
17 21
matchObj.group() : than
```

## Matching Vs Searching

Python offers two different primitive operations based on regular expressions :`match` checks for a match only at the beginning of the string, while `search` checks for a match anywhere in the string (this is what Perl does by default).

## Example

```
</> Open Compiler
```

```

import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print ("match --> matchObj.group() : ", matchObj.group())
else:
    print ("No match!!")
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
    print ("search --> searchObj.group() : ", searchObj.group())
else:
    print ("Nothing found!!")

```

When the above code is executed, it produces the following **output** –

```

No match!!
search --> matchObj.group() : dogs

```

## re.findall() Function

The `findall()` function returns all non-overlapping matches of pattern in string, as a list of strings or tuples. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

### Syntax

```
re.findall(pattern, string, flags=0)
```

### Parameters

| Sr.No. | Parameter & Description                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------|
| 1      | <b>Pattern</b><br>This is the regular expression to be matched.                                           |
| 2      | <b>String</b><br>This is the string, which would be searched to match the pattern anywhere in the string. |

**Flags**

3 You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

## Example

&lt;/&gt;

Open Compiler

```
import re
string="Simple is better than complex."
obj=re.findall(r"ple", string)
print (obj)
```

It will produce the following **output** –

```
['ple', 'ple']
```

Following code obtains the list of words in a sentence with the help of `findall()` function.

&lt;/&gt;

Open Compiler

```
import re
string="Simple is better than complex."
obj=re.findall(r"\w*", string)
print (obj)
```

It will produce the following **output** –

```
['Simple', '', 'is', '', 'better', '', 'than', '', 'complex', '', '']
```

## re.sub() Function

One of the most important `re` methods that use regular expressions is **sub**.

## Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max is provided. This method returns modified string.

## Example

```
</> Open Compiler  
  
import re  
phone = "2004-959-559 # This is Phone Number"  
  
# Delete Python-style comments  
num = re.sub(r'#.*$', "", phone)  
print ("Phone Num : ", num)  
  
# Remove anything other than digits  
num = re.sub(r'\D', "", phone)  
print ("Phone Num : ", num)
```

It will produce the following **output** –

```
Phone Num : 2004-959-559  
Phone Num : 2004959559
```

## Example

The following example uses sub() function to substitute all occurrences of is with was word

–

```
</> Open Compiler  
  
import re  
string="Simple is better than complex. Complex is better than complicated."  
obj=re.sub(r'is', r'was',string)  
print (obj)
```

It will produce the following **output** –

```
Simple was better than complex. Complex was better than complicated.
```

## re.compile() Function

The compile() function compiles a regular expression pattern into a regular expression object, which can be used for matching using its match(), search() and other methods.

### Syntax

```
re.compile(pattern, flags=0)
```

### Flags

| Sr.No. | Modifier & Description                                                                                                                                                              |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>re.I</b><br>Performs case-insensitive matching.                                                                                                                                  |
| 2      | <b>re.L</b><br>Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).       |
| 3      | <b>re.M</b><br>M Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).                      |
| 4      | <b>re.S</b><br>Makes a period (dot) match any character, including a newline.                                                                                                       |
| 5      | <b>re.U</b><br>Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.                                                         |
| 6      | <b>re.X</b><br>Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

The sequence –

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to –

```
result = re.match(pattern, string)
```

But using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

## Example

&lt;/&gt;

Open Compiler

```
import re
string="Simple is better than complex. Complex is better than complicated."
pattern=re.compile(r'is')
obj=pattern.match(string)
obj=pattern.search(string)
print (obj.start(), obj.end())

obj=pattern.findall(string)
print (obj)

obj=pattern.sub(r'was', string)
print (obj)
```

It will produce the following output –

7 9

['is', 'is']

Simple was better than complex. Complex was better than complicated.

## re.finditer() Function

This function returns an iterator yielding match objects over all non-overlapping matches for the RE pattern in string.

## Syntax

```
re.finditer(pattern, string, flags=0)
```

## Example

&lt;/&gt;

Open Compiler

```
import re
string="Simple is better than complex. Complex is better than
complicated."
pattern=re.compile(r'is')
iterator = pattern.finditer(string)
print (iterator )

for match in iterator:
    print(match.span())
```

It will produce the following **output** –

```
(7, 9)
(39, 41)
```

## Use Cases of Python Regex

### Finding all Adverbs

`.findall()` matches all occurrences of a pattern, not just the first one as `search()` does. For example, if a writer wanted to find all of the adverbs in some text, they might use `.findall()` in the following manner –

```
</>
```

Open Compiler

```
import re
text = "He was carefully disguised but captured quickly by police."
obj = re.findall(r"\w+ly\b", text)
print (obj)
```

It will produce the following **output** –

```
['carefully', 'quickly']
```

### Finding words starting with vowels

```
</>
```

Open Compiler

```
import re  
text = 'Errors should never pass silently. Unless explicitly silenced.'  
obj=re.findall(r'\b[aeiouAEIOU]\w+', text)  
print (obj)
```

It will produce the following **output** –

```
['Errors', 'Unless', 'explicitly']
```

## Python - PIP

Python's standard library is a large collection of ready-to-use modules and packages. In addition to these packages, a Python programmer often needs to use certain third-party libraries. Third-party Python packages are hosted on a repository called Python Package Index (<https://pypi.org/>).

To install a package from this repository, you need a package manager tool. PIP is one of the most popular package managers.

The PIP utility is automatically installed with Python's standard distribution especially with version 3.4 onwards. It is present in the scripts folder inside Python's installation directory. For example, when Python 3.11 is installed on a Windows computer, you can find pip3.exe in C:\Python311\Scripts folder.

If **pip** is not installed by default, it can be installed by the following procedure.

Download get-pip.py script from following URL –

```
https://bootstrap.pypa.io/get-pip.py
```

To install run above script from command prompt –

```
c:\Python311>python get-pip.py
```

In scripts folder both pip and pip3 are present. If pip is used to install a certain package, its Python 2.x compatible version will be installed. Hence to install Python 3 compatible version, use pip3.

## Install a Package

To install a certain package from PyPi, use install command with PIP. Following command installs Flask library in the current Python installation.

```
pip3 install flask
```

The package, along with its dependencies if any, will be installed from the PyPI repository. The above command produces following log in the terminal –

```
Collecting flask
  Downloading Flask-2.2.3-py3-none-any.whl (101 kB)
  ━━━━━━━━━━━━━━━━
  101.8/101.8 kB 3.0 MB/s eta 0:00:00
Collecting Werkzeug>=2.2.2
  Downloading Werkzeug-2.2.3-py3-none-any.whl (233 kB)
  ━━━━━━━━━━━━━━━━
  233.6/233.6 kB 7.2 MB/s eta 0:00:00
Collecting Jinja2>=3.0
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
  ━━━━━━━━━━━━━━━━
  133.1/133.1 kB 8.2 MB/s eta 0:00:00
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting click>=8.0
  Downloading click-8.1.3-py3-none-any.whl (96 kB)
  ━━━━━━━━━━━━━━━━
  96.6/96.6 kB 5.4 MB/s eta 0:00:00
Requirement already satisfied: colorama in
c:\users\mlath\appdata\roaming\python\python311\site-packages (from
click>=8.0->flask) (0.4.6)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.2-cp311-cp311-win_amd64.whl (16 kB)
Installing collected packages: MarkupSafe, itsdangerous, click,
Werkzeug, Jinja2, flask
Successfully installed Jinja2-3.1.2 MarkupSafe-2.1.2 Werkzeug-2.2.3
click-8.1.3 flask-2.2.3 itsdangerous-2.1.2
```

By default, the latest available version of the desired package is installed. To specify the version required,

```
pip3 install flask==2.0.0
```

To test if the package installation is complete, open Python interpreter and try to import it and check the version. If the package hasn't been successfully installed, you get a `ModuleNotFoundError`.

```
>>> import flask
>>> print (flask.__version__)
2.2.3
>>> import dummypackage
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'dummypackage'
```

PIP utility works with –

- PyPI (and other indexes) using requirement specifiers.
- VCS project urls.
- Local project directories.
- Local or remote source archives.

## Use requirements.txt

You can perform package installation at once by mentioning the list of required packages in a text file named as requirements.txt.

For example, the following requirements.txt file contains list of dependencies to be installed for FastAPI library.

```
anyio==3.6.2
click==8.1.3
colorama==0.4.6
fastapi==0.88.0
gunicorn==20.1.0
h11==0.14.0
idna==3.4
pydantic==1.10.4
sniffio==1.3.0
starlette==0.22.0
typing_extensions==4.4.0
uvicorn==0.20.0
```

Now use the -r switch in PIP install command.

```
pip3 install -r requirements.txt
```

The PIP utility is used with along with following commands –

## pip uninstall

This command is used to uninstall one or more packages already installed.

### Syntax

```
pip3 uninstall package, [package2, package3, . . .]
```

This will uninstall the packages along with the dependencies.

### Example

```
pip3 uninstall flask
```

You will be asked confirmation for removal before proceeding.

```
pip3 uninstall flask
Found existing installation: Flask 2.2.3
Uninstalling Flask-2.2.3:
Would remove:
c:\python311\lib\site-packages\flask-2.2.3.dist-info\*
c:\python311\lib\site-packages\flask\*
c:\python311\scripts\flask.exe
Proceed (Y/n)?
```

## pip list

This command gives a list installed packages, including editables. Packages are listed in a case-insensitive sorted order.

### Syntax

```
pip3 list
```

Following switches are available with pip list command –

**-o, --outdated: List outdated packages**

```
pip3 list --outdated
```

| Package    | Version | Latest | Type  |
|------------|---------|--------|-------|
| debugpy    | 1.6.6   | 1.6.7  | wheel |
| ipython    | 8.11.0  | 8.12.0 | wheel |
| pip        | 22.3.1  | 23.0.1 | wheel |
| Pygments   | 2.14.0  | 2.15.0 | wheel |
| setuptools | 65.5.0  | 67.6.1 | wheel |

### -u, --uptodate : List update packages

```
pip3 list --uptodate
```

| Package         | Version |
|-----------------|---------|
| click           | 8.1.3   |
| colorama        | 0.4.6   |
| executing       | 1.2.0   |
| Flask           | 2.2.3   |
| jedi            | 0.18.2  |
| Jinja2          | 3.1.2   |
| python-dateutil | 2.8.2   |
| pyzmq           | 25.0.2  |
| six             | 1.16.0  |
| Werkzeug        | 2.2.3   |

## pip show

This command shows information about one or more installed packages. The output is in RFC-compliant mail header format.

### Syntax

```
pip3 show package
```

### Example

```
pip3 show flask
```

Name: Flask

Version: 2.2.3

Summary: A simple framework for building complex web applications.

Home-page: <https://palletsprojects.com/p/flask>

```
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: C:\Python311\Lib\site-packages
Requires: click, itsdangerous, Jinja2, Werkzeug
Required-by:
```

## pip freeze

This command outputs installed packages in requirements format. All the packages are listed in a case-insensitive sorted order.

### Syntax

```
pip3 freeze
```

The output of this command can be redirected to a text file with the following command –

```
pip3 freeze > requirements.txt
```

## pip download

This command downloads packages from –

- PyPI (and other indexes) using requirement specifiers.
- VCS project urls.
- Local project directories.
- Local or remote source archives.

In fact, pip download does the same resolution and downloading as pip install, but instead of installing the dependencies, it collects the downloaded distributions into the directory provided (defaulting to the current directory). This directory can later be passed as the value to pip install --find-links to facilitate offline or locked down package installation.

### Syntax

```
pip3 download somepackage
```

## pip search

This command searches for PyPI packages whose name or summary contains the given query.

### Syntax

```
pip3 search query
```

## pip config

This command is used to manage local and global configuration.

### Subcommands

- **list** – List the active configuration (or from the file specified).
- **edit** – Edit the configuration file in an editor.
- **get** – Get the value associated with command.option.
- **set** – Set the command.option=value.
- **unset** – Unset the value associated with command.option.
- **debug** – List the configuration files and values defined under them.

Configuration keys should be dot separated command and option name, with the special prefix "global" affecting any command.

### Example

```
pip config set global.index-url https://example.org/
```

This would configure the index url for all commands.

```
pip config set download.timeout 10
```

This would configure a 10 second timeout only for "pip download" commands.

# Python - Database Access

Data input and generated during execution of a program is stored in RAM. If it is to be stored persistently, it needs to be stored in database tables. There are various relational

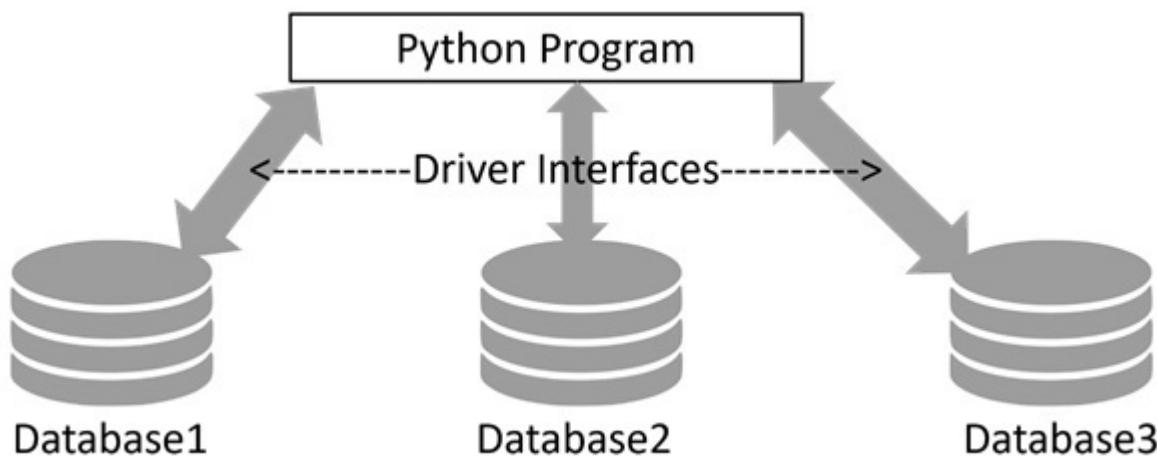
database management systems (RDBMS) available.

- GadFly
- MySQL
- PostgreSQL
- Microsoft SQL Server
- Informix
- Oracle
- Sybase
- SQLite
- and many more...

In this chapter, we shall learn how to access database using Python, how to store data of Python objects in a SQLite database, and how to retrieve data from SQLite database and process it using Python program.

Relational databases use SQL (Structured Query Language) for performing INSERT/DELETE/UPDATE operations on the database tables. However, implementation of SQL varies from one type of database to other. This raises incompatibility issues. SQL instructions for one database do not match with other.

To overcome this incompatibility, a common interface was proposed in PEP (Python Enhancement Proposal) 249. This proposal is called DB-API and requires that a database driver program used to interact with Python should be DB-API compliant.



Python's standard library includes `sqlite3` module which is a DB-API compatible driver for SQLite3 database, it is also a reference implementation of DB-API.

Since the required DB-API interface is built-in, we can easily use SQLite database with a Python application. For other types of databases, you will have to install the relevant Python package.

| Database   | Python Package                  |
|------------|---------------------------------|
| Oracle     | cx_oracle, pyodbc               |
| SQL Server | pymssql, pyodbc                 |
| PostgreSQL | psycopg2                        |
| MySQL      | MySQL Connector/Python, pymysql |

A DB-API module such as sqlite3 contains connection and cursor classes. The connection object is obtained with connect() method by providing required connection credentials such as name of server and port number, and username and password if applicable. The connection object handles opening and closing the database, and transaction control mechanism of committing or rolling back a transaction.

The cursor object, obtained from the connection object, acts as the handle of the database when performing all the CRUD operations.

## The sqlite3 Module

SQLite is a server-less, file-based lightweight transactional relational database. It doesn't require any installation and no credentials such as username and password are needed to access the database.

Python's sqlite3 module contains DB-API implementation for SQLite database. It is written by Gerhard Häring. Let us learn how to use sqlite3 module for database access with Python.

Let us start by importing sqlite3 and check its version.

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.39.4'
```

## The Connection Object

A connection object is set up by connect() function in sqlite3 module. First positional argument to this function is a string representing path (relative or absolute) to a SQLite database file. The function returns a connection object referring to the database.

```
>>> conn=sqlite3.connect('testdb.sqlite3')
>>> type(conn)
<class 'sqlite3.Connection'>
```

Various methods are defined in connection class. One of them is cursor() method that returns a cursor object, about which we shall know in next section. Transaction control is achieved by commit() and rollback() methods of connection object. Connection class has important methods to define custom functions and aggregates to be used in SQL queries.

## The Cursor Object

Next, we need to get the cursor object from the connection object. It is your handle to the database when performing any CRUD operation on the database. The cursor() method on connection object returns the cursor object.

```
>>> cur=conn.cursor()  
>>> type(cur)  
<class 'sqlite3.Cursor'>
```

We can now perform all SQL query operations, with the help of its execute() method available to cursor object. This method needs a string argument which must be a valid SQL statement.

## Creating a Database Table

We shall now add Employee table in our newly created 'testdb.sqlite3' database. In following script, we call execute() method of cursor object, giving it a string with CREATE TABLE statement inside.

```
</>
```

Open Compiler

```
import sqlite3  
conn=sqlite3.connect('testdb.sqlite3')  
cur=conn.cursor()  
qry='''  
CREATE TABLE Employee (  
    EmpID INTEGER PRIMARY KEY AUTOINCREMENT,  
    FIRST_NAME TEXT (20),  
    LAST_NAME TEXT(20),  
    AGE INTEGER,  
    SEX TEXT(1),  
    INCOME FLOAT  
);  
'''  
try:  
    cur.execute(qry)
```

```
    print ('Table created successfully')
except:
    print ('error in creating table')
conn.close()
```

When the above program is run, the database with Employee table is created in the current working directory.

We can verify by listing out tables in this database in SQLite console.

```
sqlite> .open mydb.sqlite
sqlite> .tables
Employee
```

## INSERT Operation

The INSERT Operation is required when you want to create your records into a database table.

### Example

The following example, executes SQL INSERT statement to create a record in the EMPLOYEE table –

```
</> Open Compiler

import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
    LAST_NAME, AGE, SEX, INCOME)
    VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
    cur.execute(qry)
    conn.commit()
    print ('Record inserted successfully')
except:
    conn.rollback()
print ('error in INSERT operation')
conn.close()
```

You can also use the parameter substitution technique to execute the INSERT query as follows –

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
    LAST_NAME, AGE, SEX, INCOME)
VALUES (?, ?, ?, ?, ?)"""
try:
    cur.execute(qry, ('Makrand', 'Mohan', 21, 'M', 5000))
    conn.commit()
    print ('Record inserted successfully')
except Exception as e:
    conn.rollback()
    print ('error in INSERT operation')
conn.close()
```

## READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once the database connection is established, you are ready to make a query into this database. You can use either `fetchone()` method to fetch a single record or `fetchall()` method to fetch multiple values from a database table.

- **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall()** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **rowcount** – This is a read-only attribute and returns the number of rows that were affected by an `execute()` method.

## Example

In the following code, the cursor object executes `SELECT * FROM EMPLOYEE` query. The resultset is obtained with `fetchall()` method. We print all the records in the resultset with a `for` loop.

```

import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="SELECT * FROM EMPLOYEE"

try:
    # Execute the SQL command
    cur.execute(qry)
    # Fetch all the rows in a list of lists.
    results = cur.fetchall()
    for row in results:
        fname = row[1]
        lname = row[2]
        age = row[3]
        sex = row[4]
        income = row[5]
        # Now print fetched result
        print ("fname={},lname={},age={},sex={},income={}".format(fname, lname, age,
except Exception as e:
    print (e)
    print ("Error: unable to fetch data")

conn.close()

```

It will produce the following **output** –

```

fname=Mac,lname=Mohan,age=20,sex=M,income=2000.0
fname=Makrand,lname=Mohan,age=21,sex=M,income=5000.0

```

## Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having income=2000. Here, we increase the income by 1000.

```

import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="UPDATE EMPLOYEE SET INCOME = INCOME+1000 WHERE INCOME=?"

```

```
try:  
    # Execute the SQL command  
    cur.execute(qry, (1000,))  
    # Fetch all the rows in a list of lists.  
    conn.commit()  
    print ("Records updated")  
except Exception as e:  
    print ("Error: unable to update data")  
conn.close()
```

## DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where INCOME is less than 2000.

```
import sqlite3  
conn=sqlite3.connect('testdb.sqlite3')  
cur=conn.cursor()  
qry="DELETE FROM EMPLOYEE WHERE INCOME<?"  
  
try:  
    # Execute the SQL command  
    cur.execute(qry, (2000,))  
    # Fetch all the rows in a list of lists.  
    conn.commit()  
    print ("Records deleted")  
except Exception as e:  
    print ("Error: unable to delete data")  
  
conn.close()
```

## Performing Transactions

Transactions are a mechanism that ensure data consistency. Transactions have the following four properties –

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.

- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.



The Python DB API 2.0 provides two methods to either commit or rollback a transaction.

## Example

You already know how to implement transactions. Here is a similar example –

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > ?"
try:
    # Execute the SQL command
    cursor.execute(sql, (20,))
    # Commit your changes in the database
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()
```

## COMMIT Operation

Commit is an operation, which gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the commit method.

```
db.commit()
```

## ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the `rollback()` method.

Here is a simple example to call the `rollback()` method.

```
db.rollback()
```

## The PyMySQL Module

PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb.

## Installing PyMySQL

Before proceeding further, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it –

```
import PyMySQL
```

If it produces the following result, then it means MySQLdb module is not installed –

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    Import PyMySQL
ImportError: No module named PyMySQL
```

The last stable release is available on PyPI and can be installed with pip –

```
pip install PyMySQL
```

**Note** – Make sure you have root privilege to install the above module.

## MySQL Database Connection

Before connecting to a MySQL database, make sure of the following points –

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST\_NAME, LAST\_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.

- Python module PyMySQL is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

## Example

To use MySQL database instead of SQLite database in earlier examples, we need to change the connect() function as follows –

```
import PyMySQL
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
```

Apart from this change, every database operation can be performed without difficulty.

## Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already cancelled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

| Sr.No. | Exception & Description                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Warning</b><br>Used for non-fatal issues. Must subclass StandardError.                                                                                                                              |
| 2      | <b>Error</b><br>Base class for errors. Must subclass StandardError.                                                                                                                                    |
| 3      | <b>InterfaceError</b><br>Used for errors in the database module, not the database itself. Must subclass Error.                                                                                         |
| 4      | <b>DatabaseError</b><br>Used for errors in the database. Must subclass Error.                                                                                                                          |
| 5      | <b>DataError</b><br>Subclass of DatabaseError that refers to errors in the data.                                                                                                                       |
| 6      | <b>OperationalError</b><br>Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter. |

|    |                                                                                                                                      |
|----|--------------------------------------------------------------------------------------------------------------------------------------|
|    | <b>IntegrityError</b>                                                                                                                |
| 7  | Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys. |
|    | <b>InternalError</b>                                                                                                                 |
| 8  | Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.            |
|    | <b>ProgrammingError</b>                                                                                                              |
| 9  | Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.          |
|    | <b>NotSupportedError</b>                                                                                                             |
| 10 | Subclass of DatabaseError that refers to trying to call unsupported functionality.                                                   |

## Python - Weak References

Python uses reference counting mechanism while implementing garbage collection policy. Whenever an object in the memory is referred, the count is incremented by one. On the other hand, when the reference is removed, the count is decremented by 1. If the garbage collector running in the background finds any object with count as 0, it is removed and the memory occupied is reclaimed.

Weak reference is a reference that does not protect the object from getting garbage collected. It proves important when you need to implement caches for large objects, as well as in a situation where reduction of Pain from circular references is desired.

To create weak references, Python has provided us with a module named `weakref`.

The `ref` class in this module manages the weak reference to an object. When called, it retrieves the original object.

To create a weak reference –

```
weakref.ref(class())
```

### Example

```
</>
```

Open Compiler

```
import weakref
class Myclass:
    def __del__(self):
```

```

        print('(Deleting {})'.format(self))
obj = Myclass()
r = weakref.ref(obj)

print('object:', obj)
print('reference:', r)
print('call r():', r())

print('deleting obj')
del obj
print('r():', r())

```

Calling the reference object after deleting the referent returns None.

It will produce the following **output** –

```

object: <__main__.Myclass object at 0x00000209D7173290>
reference: <weakref at 0x00000209D7175940; to 'Myclass' at
0x00000209D7173290>
call r(): <__main__.Myclass object at 0x00000209D7173290>
deleting obj
(Deleting <__main__.Myclass object at 0x00000209D7173290>)
r(): None

```

## The callback Function

The constructor of ref class has an optional parameter called callback function, which gets called when the referred object is deleted.

&lt;/&gt;

Open Compiler

```

import weakref
class Myclass:
    def __del__(self):
        print('(Deleting {})'.format(self))
def mycallback(rfr):
    """called when referenced object is deleted"""
    print('calling {}'.format(rfr))
obj = Myclass()
r = weakref.ref(obj, mycallback)

```

```

print('object:', obj)
print('reference:', r)
print('call r():', r())

print('deleting obj')
del obj
print('r():', r())

```

It will produce the following **output** –

```

object: <__main__.Myclass object at 0x000002A0499D3590>
reference: <weakref at 0x000002A0499D59E0; to 'Myclass' at
0x000002A0499D3590>
call r(): <__main__.Myclass object at 0x000002A0499D3590>
deleting obj
(Deleting <__main__.Myclass object at 0x000002A0499D3590>)
calling (<weakref at 0x000002A0499D59E0; dead>)
r(): None

```

## Finalizing Objects

The **weakref** module provides finalize class. Its object is called when the garbage collector collects the object. The object survives until the reference object is called.

&lt;/&gt;

Open Compiler

```

import weakref
class Myclass:
    def __del__(self):
        print('(Deleting {})'.format(self))

    def finalizer(*args):
        print('Finalizer{}'.format(args))

obj = Myclass()
r = weakref.finalize(obj, finalizer, "Call to finalizer")

print('object:', obj)
print('reference:', r)
print('call r():', r())

```

```
print('deleting obj')
del obj
print('r():', r())
```

It will produce the following **output** –

```
object: <__main__.Myclass object at 0x0000021015103590>
reference: <finalize object at 0x21014eabe80; for 'Myclass' at
0x21015103590>
Finalizer('Call to finalizer',)
call r(): None
deleting obj
(Deleting <__main__.Myclass object at 0x0000021015103590>)
r(): None
```

The `weakref` module provides `WeakKeyDictionary` and `WeakValueDictionary` classes. They don't keep the objects alive as they appear in the mapping objects. They are more appropriate for creating a cache of several objects.

## WeakKeyDictionary

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key.

An instance of `WeakKeyDictionary` class is created with an existing dictionary or without any argument. The functionality is the same as a normal dictionary to add and remove mapping entries to it.

In the code given below three `Person` instances are created. It then creates an instance of `WeakKeyDictionary` with a dictionary where the key is the `Person` instance and the value is the `Person`'s name.

We call the `keyrefs()` method to retrieve weak references. When the reference to `Peron1` is deleted, dictionary keys are printed again. A new `Person` instance is added to a dictionary with weakly referenced keys. At last, we are printing keys of dictionary again.

## Example

</>

Open Compiler

```
import weakref

class Person:
```

```

def __init__(self, person_id, name, age):
    self.emp_id = person_id
    self.name = name
    self.age = age

def __repr__(self):
    return "{} : {} : {}".format(self.person_id, self.name, self.age)

Person1 = Person(101, "Jeevan", 30)
Person2 = Person(102, "Ramanna", 35)
Person3 = Person(103, "Simran", 28)
weak_dict = weakref.WeakKeyDictionary({Person1: Person1.name, Person2: Person2.name})
print("Weak Key Dictionary : {}\n".format(weak_dict.data))
print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))
del Person1
print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))
Person4 = Person(104, "Partho", 32)
weak_dict.update({Person4: Person4.name})

print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))

```

It will produce the following **output** –

```

Weak Key Dictionary : {<weakref at 0x7f542b6d4180; to 'Person' at 0x7f542b8bbfd0>: 'Jeevan'}
Dictionary Keys : ['Jeevan', 'Ramanna', 'Simran']
Dictionary Keys : ['Ramanna', 'Simran']
Dictionary Keys : ['Ramanna', 'Simran', 'Partho']

```

## WeakValueDictionary

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

We shall demonstrate how to create a dictionary with weakly referenced values using WeakValueDictionary.

The code is similar to previous example but this time we are using Person name as key and Person instance as values. We are using valuerrefs() method to retrieve weakly referenced values of the dictionary.

## Example

&lt;/&gt;

Open Compiler

```
import weakref

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

    def __repr__(self):
        return "{} : {} : {}".format(self.person_id, self.name, self.age)

Person1 = Person(101, "Jeevan", 30)
Person2 = Person(102, "Ramanna", 35)
Person3 = Person(103, "Simran", 28)

weak_dict = weakref.WeakValueDictionary({Person1.name:Person1, Person2.name:Person2})
print("Weak Value Dictionary : {}\n".format(weak_dict.data))
print("Dictionary Values : {}\n".format([value().name for value in weak_dict.values()]))
del Person1
print("Dictionary Values : {}\n".format([value().name for value in weak_dict.values()]))
Person4 = Person(104, "Partho", 32)
weak_dict.update({Person4.name: Person4})
print("Dictionary Values : {}\n".format([value().name for value in weak_dict.values()]))
```

It will produce the following **output** –

```
Weak Value Dictionary : {'Jeevan': <weakref at 0x7f3af9fe4180; to 'Person' at 0x7f3afa10000>, 'Ramanna': <weakref at 0x7f3af9fe41c0; to 'Person' at 0x7f3afa100040>, 'Simran': <weakref at 0x7f3af9fe4200; to 'Person' at 0x7f3afa100080>}

Dictionary Values : ['Jeevan', 'Ramanna', 'Simran']

Dictionary Values : ['Ramanna', 'Simran']

Dictionary Values : ['Ramanna', 'Simran', 'Partho']
```

# Python - Serialization

The term "object serialization" refers to process of converting state of an object into byte stream. Once created, this byte stream can further be stored in a file or transmitted via sockets etc. On the other hand, reconstructing the object from the byte stream is called deserialization.

Python's terminology for serialization and deserialization is pickling and unpickling respectively. The pickle module available in Python's standard library provides functions for serialization (`dump()` and `dumps()`) and deserialization (`load()` and `loads()`).

The pickle module uses very Python specific data format. Hence, programs not written in Python may not be able to deserialize the encoded (pickled) data properly. Also it is not considered to be secure to unpickle data from un-authenticated source.

## Pickle Protocols

Protocols are the conventions used in constructing and deconstructing Python objects to/from binary data. Currently pickle module defines 5 different protocols as listed below –

| Sr.No. | Protocol & Description                                                                                                     |
|--------|----------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Protocol version 0</b><br>Original "human-readable" protocol backwards compatible with earlier versions.                |
| 2      | <b>Protocol version 1</b><br>Old binary format also compatible with earlier versions of Python.                            |
| 3      | <b>Protocol version 2</b><br>Introduced in Python 2.3 provides efficient pickling of new-style classes.                    |
| 4      | <b>Protocol version 3</b><br>Added in Python 3.0. recommended when compatibility with other Python 3 versions is required. |
| 5      | <b>Protocol version 4</b><br>Was added in Python 3.4. It adds support for very large objects.                              |

To know the highest and default protocol version of your Python installation, use the following constants defined in the **pickle** module –

```
>>> import pickle
>>> pickle.HIGHEST_PROTOCOL
4
```

```
>>> pickle.DEFAULT_PROTOCOL  
3
```

The `dump()` and `load()` functions of the **pickle** module perform pickling and unpickling Python data. The `dump()` function writes pickled object to a file and `load()` function unpickles data from file to Python object.

## dump() and load()

Following program pickle a dictionary object into a binary file.

```
import pickle  
f=open("data.txt","wb")  
dct={"name":"Ravi", "age":23, "Gender":"M","marks":75}  
pickle.dump(dct,f)  
f.close()
```

When above code is executed, the dictionary object's byte representation will be stored in `data.txt` file.

To unpickle or deserialize data from a binary file back to dictionary, run following program.

```
import pickle  
f=open("data.txt","rb")  
d=pickle.load(f)  
print (d)  
f.close()
```

Python console shows the dictionary object read from file.

```
{'age': 23, 'Gender': 'M', 'name': 'Ravi', 'marks': 75}
```

## dumps() and loads()

The pickle module also consists of `dumps()` function that returns a string representation of pickled data.

```
>>> from pickle import dump  
>>> dct={"name":"Ravi", "age":23, "Gender":"M","marks":75}  
>>> dctstring=dumps(dct)
```

```
>>> dctstring
b'\x80\x03\0\00(X\x04\x00\x00\x00named\x01X\x04\x00\x00\x00Ravin\x02X\x03\x00\x00
```

Use loads() function to unpickle the string and obtain original dictionary object.

```
from pickle import load
dct=loads(dctstring)
print (dct)
```

It will produce the following **output** –

```
{'name': 'Ravi', 'age': 23, 'Gender': 'M', 'marks': 75}
```

## Pickler Class

The pickle module also defines Pickler and Unpickler classes. Pickler class writes pickle data to file. Unpickler class reads binary data from file and constructs Python object.

To write Python object's pickled data –

```
from pickle import Pickler
f=open("data.txt","wb")
dct={'name': 'Ravi', 'age': 23, 'Gender': 'M', 'marks': 75}
Pickler(f).dump(dct)
f.close()
```

## Unpickler Class

To read back data by unpickling binary file –

```
from pickle import Unpickler
f=open("data.txt","rb")
dct=Unpickler(f).load()
print (dct)
f.close()
```

Objects of all Python standard data types are picklable. Moreover, objects of custom class can also be pickled and unpickled.

```
from pickle import *
class person:
```

```

def __init__(self):
    self.name="XYZ"
    self.age=22
def show(self):
    print ("name:", self.name, "age:", self.age)
p1=person()
f=open("data.txt","wb")
dump(p1,f)
f.close()
print ("unpickled")
f=open("data.txt","rb")
p1=load(f)
p1.show()

```

Python library also has marshal module that is used for internal serialization of Python objects.

## Python - Templating

Python provides different text formatting features. It including formatting operators, Python's format() function and the f-string. In addition, Python's standard library includes string module that comes with more formatting options.

The Template class in string module is useful for forming a string object dynamically by substitution technique described in PEP 292. Its the simpler syntax and functionality makes it easier to translate in case of internalization than other built-in string formatting facilities in Python.

Template strings use \$ symbol for substitution. The symbol is immediately followed by an identifier that follows the rules of forming a valid Python identifier.

### Syntax

```

from string import Template

tempStr = Template('Hello $name')

```

The Template class defines the following methods –

#### substitute()

This method performs substitution of value the identifiers in the Template object. Using keyword arguments or a dictionary object can be used to map the identifiers in the

template. The method returns a new string.

## Example 1

Following code uses keyword arguments for substitute() method.

&lt;/&gt;

Open Compiler

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
newStr = tempStr.substitute(name = 'Pushpa', age = 26)
print (newStr)
```

It will produce the following **output** –

Hello. My name is Pushpa and my age is 26

## Example 2

In the following example, we use a dictionary object to map the substitution identifiers in the template string.

&lt;/&gt;

Open Compiler

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
dct = {'name' : 'Pushpalata', 'age' : 25}
newStr = tempStr.substitute(dct)
print (newStr)
```

It will produce the following **output** –

Hello. My name is Pushpalata and my age is 25

## Example 3

If the substitute() method is not provided with sufficient parameters to be matched against the identifiers in the template string, Python raises KeyError.

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
dct = {'name' : 'Pushpalata'}
newStr = tempStr.substitute(dct)
print (newStr)
```

It will produce the following **output** –

## Traceback (most recent call last):

```
File "C:\Users\user\example.py", line 5, in   
    newStr = tempStr.substitute(dct)  
    ^^^^^^
```

```
File "C:\Python311\Lib\string.py", line 114, in convert
    return str(mapping[named])
```

16 *Environ Biol Fish* (2007) 80:15–16

safe substitute()

This method behaves similarly to `substitute()` method, except for the fact that it doesn't throw error if the keys are not sufficient or are not matching. Instead, the original placeholder will appear in the resulting string intact.

## Example 4

```
from string import Template  
tempStr = Template('Hello. My name is $name and my age is $age')  
dct = {'name' : 'Pushpalata'}  
newStr = tempStr.safe_substitute(dct)  
print (newStr)
```

It will produce the following **output** –

```
Hello. My name is Pushpalata and my age is $age
```

## is\_valid()

Returns false if the template has invalid placeholders that will cause substitute() to raise ValueError.

## get\_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

### Example 5

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $23')
print (tempStr.is_valid())
tempStr = Template('Hello. My name is $name and my age is $age')
print (tempStr.get_identifiers())
```

It will produce the following **output** –

False

['name', 'age']

### Example 6

The "" symbol has been defined as the substitution character. If you need itself to appear in the string, it has to be escaped. In other words, use \$\$ to use it in the string.

</>

Open Compiler

```
from string import Template

tempStr = Template('The symbol for Dollar is $$')
print (tempStr.substitute())
```

It will produce the following **output** –

The symbol for Dollar is \$

## Example 7

If you wish to use any other character instead of "\$" as the substitution symbol, declare a subclass of Template class and assign –

```
</> Open Compiler  
  
from string import Template  
  
class myTemplate(Template):  
    delimiter = '#'  
  
tempStr = myTemplate('Hello. My name is #name and my age is #age')  
print (tempStr.substitute(name='Harsha', age=30))
```

# Python - Output Formatting

In this chapter, different techniques for formatting the output will be discussed.

## String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the lack of having functions from C's printf() family. Format specification symbols (%d %c %f %s etc) used in C language are used as placeholders in a string.

Following is a simple example –

```
</> Open Compiler  
  
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

It will produce the following **output** –

My name is Zara and weight is 21 kg!

## The format() Method

Python 3.0, introduced format() method to str class for handling complex string formatting more efficiently. This method has since been backported to Python 2.6 and Python 2.7.

This method of in-built string class provides ability to do complex variable substitutions and value formatting. This new formatting technique is regarded as more elegant.

### Syntax

The general syntax of format() method is as follows –

```
str.format(var1, var2,...)
```

### Return Value

The method returns a formatted string.

The string itself contains placeholders {} in which values of variables are successively inserted.

### Example

```
</>
```

Open Compiler

```
name="Rajesh"  
age=23  
print ("my name is {} and my age is {} years".format(name, age))
```

It will produce the following **output** –

```
my name is Rajesh and my age is 23 years
```

You can use variables as keyword arguments to format() method and use the variable name as the placeholder in the string.

```
print ("my name is {name} and my age is {age}  
years".format(name="Rajesh", age=23))
```

You can also specify C style formatting symbols. Only change is using : instead of %. For example, instead of %s use {:s} and instead of %d use (:d)

```
name="Rajesh"  
age=23  
print ("my name is {:s} and my age is {:d} years".format(name, age))
```

## f-strings

In Python, f-strings or Literal String Interpolation is another formatting facility. With this formatting method you can use embedded Python expressions inside string constants. Python f-strings are a faster, more readable, more concise, and less error prone.

The string starts with a 'f' prefix, and one or more place holders are inserted in it, whose value is filled dynamically.

&lt;/&gt;

Open Compiler

```
name = 'Rajesh'  
age = 23  
  
fstring = f'My name is {name} and I am {age} years old'  
print (fstring)
```

It will produce the following **output** –

```
My name is Rajesh and I am 23 years old
```

## Template Strings

The Template class in string module provides an alternative method to format the strings dynamically. One of the benefits of Template class is to be able to customize the formatting rules.

A valid template string, or placeholder, consists of two parts: The "\$" symbol followed by a valid Python identifier.

You need to create an object of Template class and use the template string as an argument to the constructor.

Next, call the substitute() method of Template class. It puts the values provided as the parameters in place of template strings.

## Example

&lt;/&gt;

[Open Compiler](#)

```
from string import Template

temp_str = "My name is $name and I am $age years old"
tempobj = Template(temp_str)
ret = tempobj.substitute(name='Rajesh', age=23)
print (ret)
```

It will produce the following **output** –

```
My name is Rajesh and I am 23 years old
```

## The textwrap Module

The wrap class in Python's textwrap module contains functionality to format and wrap plain texts by adjusting the line breaks in the input paragraph. It helps in making the text wellformatted and beautiful.

The textwrap module has the following convenience functions –

**textwrap.wrap(text, width=70)**

Wraps the single paragraph in text (a string) so every line is at most width characters long. Returns a list of output lines, without final newlines. Optional keyword arguments correspond to the instance attributes of TextWrapper. width defaults to 70.

**textwrap.fill(text, width=70)**

Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph.

Both methods internally create an object of TextWrapper class and calling a single method on it. Since the instance is not reused, it will be more efficient for you to create your own TextWrapper object.

## Example

&lt;/&gt;

[Open Compiler](#)

```
import textwrap
```

```
text = ''  
Python is a high-level, general-purpose programming language. Its design philosoph  
  
Python is dynamically typed and garbage-collected. It supports multiple programmin  
...  
  
wrapper = textwrap.TextWrapper(width=40)  
wrapped = wrapper.wrap(text = text)  
  
# Print output  
for element in wrapped:  
    print(element)
```

It will produce the following **output** –

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation via the off-side rule. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Following attributes are defined for a TextWrapper object –

- **width** – (default: 70) The maximum length of wrapped lines.
- **expand\_tabs** – (default: True) If true, then all tab characters in text will be expanded to spaces using the expandtabs() method of text.
- **tabsize** – (default: 8) If expand\_tabs is true, then all tab characters in text will be expanded to zero or more spaces, depending on the current column and the given tab size.
- **replace\_whitespace** – (default: True) If true, after tab expansion but before wrapping, the wrap() method will replace each whitespace character with a single space.
- **drop\_whitespace** – (default: True) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace

at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

- **initial\_indent** – (default: "") String that will be prepended to the first line of wrapped output.
- **subsequent\_indent** – (default: "") String that will be prepended to all lines of wrapped output except the first.
- **fix\_sentence\_endings** – (default: False) If true, TextWrapper attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font.
- **break\_long\_words** – (default: True) If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width.
- **break\_on\_hyphens** – (default: True) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks.

## The shorten() Function

Collapse and truncate the given text to fit in the given width. The text first has its whitespace collapsed. If it then fits in the \*width\*, it is returned as is. Otherwise, as many words as possible are joined and then the placeholder is appended –

### Example

```
</> Open Compiler  
  
import textwrap  
  
python_desc = """Python is a general-purpose interpreted, interactive, object-orie  
  
my_wrap = textwrap.TextWrapper(width = 40)  
  
short_text = textwrap.shorten(text = python_desc, width=150)  
print('\n\n' + my_wrap.fill(text = short_text))
```

It will produce the following **output** –

Python is a general-purpose interpreted, interactive, object-oriented, and high level programming language. It was created by Guido van Rossum [...]

## The pprint Module

The pprint module in Python's standard library enables aesthetically good looking appearance of Python data structures. The name pprint stands for pretty printer. Any data structure that can be correctly parsed by Python interpreter is elegantly formatted.

The formatted expression is kept in one line as far as possible, but breaks into multiple lines if the length exceeds the width parameter of formatting. One unique feature of pprint output is that the dictionaries are automatically sorted before the display representation is formatted.

### PrettyPrinter Class

The pprint module contains definition of PrettyPrinter class. Its constructor takes following format –

### Syntax

```
pprint.PrettyPrinter(indent, width, depth, stream, compact)
```

### Parameters

- **indent** – defines indentation added on each recursive level. Default is 1.
- **width** – by default is 80. Desired output is restricted by this value. If the length is greater than width, it is broken in multiple lines.
- **depth** – controls number of levels to be printed.
- **stream** – is by default std.out – the default output device. It can take any stream object such as file.
- **compact** – is set to False by default. If true, only the data adjustable within width will be displayed.

The PrettyPrinter class defines following methods –

### pprint() method

prints the formatted representation of PrettyPrinter object.

## pformat() method

Returns the formatted representation of object, based on parameters to the constructor.

### Example

The following example demonstrates a simple use of PrettyPrinter class –

&lt;/&gt;

Open Compiler

```
import pprint
students={"Dilip": ["English", "Maths", "Science"], "Raju": {"English": 50, "Maths": 60,
pp=pprint.PrettyPrinter()
print ("normal print output")
print (students)
print ("----")
print ("pprint output")
pp.pprint(students)
```

The output shows normal as well as pretty print display –

normal print output

```
{'Dilip': ['English', 'Maths', 'Science'], 'Raju': {'English': 50, 'Maths': 60, 'Science': 70}, 'I
```

----

pprint output

```
{'Dilip': ['English', 'Maths', 'Science'],
'Kalpana': (50, 60, 70),
'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}
```

The **pprint** module also defines convenience functions pprint() and pformat() corresponding to PrettyPrinter methods. The example below uses pprint() function.

&lt;/&gt;

Open Compiler

```
from pprint import pprint
students={"Dilip": ["English", "Maths", "Science"],
 "Raju": {"English": 50, "Maths": 60, "Science": 70},
 "Kalpana": (50, 60, 70)}
```

```
print ("normal print output")
print (students)
print ("----")
print ("pprint output")
pprint (students)
```

## Example

The next example uses pformat() method as well as pformat() function. To use pformat() method, PrettyPrinter object is first set up. In both cases, the formatted representation is displayed using normal print() function.

&lt;/&gt;

Open Compiler

```
import pprint
students={"Dilip": ["English", "Maths", "Science"],
          "Raju": {"English": 50, "Maths": 60, "Science": 70},
          "Kalpana": (50, 60, 70)}
print ("using pformat method")
pp=pprint.PrettyPrinter()
string=pp.pformat(students)
print (string)
print ('-----')
print ("using pformat function")
string=pprint.pformat(students)
print (string)
```

Here is the output of the above code –

```
using pformat method
{'Dilip': ['English', 'Maths', 'Science'],
 'Kalpana': (50, 60, 70),
 'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}
-----
using pformat function
{'Dilip': ['English', 'Maths', 'Science'],
 'Kalpana': (50, 60, 70),
 'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}
```

Pretty printer can also be used with custom classes. Inside the class \_\_repr\_\_() method is overridden. The \_\_repr\_\_() method is called when repr() function is used. It is the official

string representation of Python object. When we use object as parameter to print() function it prints return value of repr() function.

## Example

In this example, the \_\_repr\_\_() method returns the string representation of player object –

```
</> Open Compiler

import pprint
class player:
    def __init__(self, name, formats=[], runs=[]):
        self.name=name
        self.formats=formats
        self.runs=runs
    def __repr__(self):
        dct={}
        dct[self.name]=dict(zip(self.formats,self.runs))
        return (repr(dct))

l1=['Tests','ODI','T20']
l2=[[140, 45, 39],[15,122,36,67, 100, 49],[78,44, 12, 0, 23, 75]]
p1=player("virat",l1,l2)
pp=pprint.PrettyPrinter()
pp.pprint(p1)
```

The **output** of above code is –

```
{'virat': {'Tests': [140, 45, 39], 'ODI': [15, 122, 36, 67, 100, 49],
'T20': [78, 44, 12, 0, 23, 75]}}
```

## Python - Performance Measurement

A given problem may be solved by more than one alternative algorithms. Hence, we need to optimize the performance of the solution. Python's **timeit** module is a useful tool to measure the performance of a Python application.

The **timit()** function in this module measures execution time of your Python code.

## Syntax

```
timeit.timeit(stmt, setup, timer, number)
```

## Parameters

- **stmt** – code snippet for measurement of performance.
- **setup** – setup details arguments to be passed or variables.
- **timer** – uses default timer, so, it may be skipped.
- **number** – the code will be executed this number of times. The default is 1000000.

## Example

The following statement uses list comprehension to return a list of multiple of 2 for each number in the range upto 100.

```
>>> [n*2 for n in range(100)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100,
102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126,
128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152,
154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178,
180, 182, 184, 186, 188, 190, 192, 194, 196, 198]
```

To measure the execution time of the above statement, we use the timeit() function as follows –

```
>>> from timeit import timeit
>>> timeit('[n*2 for n in range(100)]', number=10000)
0.0862189000035869
```

Compare the execution time with the process of appending the numbers using a for loop.

```
>>> string = ''
... numbers=[]
... for n in range(100):
...     numbers.append(n*2)
...
>>> timeit(string, number=10000)
0.1010853999905521
```

The result shows that list comprehension is more efficient.

The statement string can contain a Python function to which one or more arguments may be passed as setup code.

We shall find and compare the execution time of a factorial function using a loop with that of its recursive version.

The normal function using for loop is –

```
def fact(x):
    fact = 1
    for i in range(1, x+1):
        fact*=i
    return fact
```

Definition of recursive factorial.

```
def rfact(x):
    if x==1:
        return 1
    else:
        return x*fact(x-1)
```

Test these functions to calculate factorial of 10.

```
print ("Using loop:",fact(10))
print ("Using Recursion",rfact(10))
Result
Using loop: 3628800
Using Recursion 3628800
```

Now we shall find their respective execution time with timeit() function.

```
import timeit

setup1"""
from __main__ import fact
x = 10
"""

setup2"""
from __main__ import rfact
x = 10
"""
```

```
print ("Performance of factorial function with loop")
print(timeit.timeit(stmt = "fact(x)", setup=setup1, number=10000))

print ("Performance of factorial function with Recursion")
print(timeit.timeit(stmt = "rfact(x)", setup=setup2, number=10000))
```

## Output

```
Performance of factorial function with loop
0.00330029999895487
Performance of factorial function with Recursion
0.006506800003990065
```

The recursive function is slower than the function with loop.

In this way, we can perform performance measurement of Python code.

## Python - Data Compression

Python's standard library has a rich collection of modules for data compression and archiving. One can select whichever is suitable for his job.

There are following modules related to data compression –

| Sr.No. | Module & Description                                 |
|--------|------------------------------------------------------|
| 1      | <b>zlib</b><br>Compression compatible with gzip.     |
| 2      | <b>gzip</b><br>Support for gzip files.               |
| 3      | <b>bz2</b><br>Support for bz2 compression.           |
| 4      | <b>Izma</b><br>Compression using the LZMA algorithm. |
| 5      | <b>zipfile</b><br>Work with ZIP archives.            |
| 6      | <b>tarfilev</b><br>Read and write tar archive files. |

# Python - CGI Programming

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA.

## What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

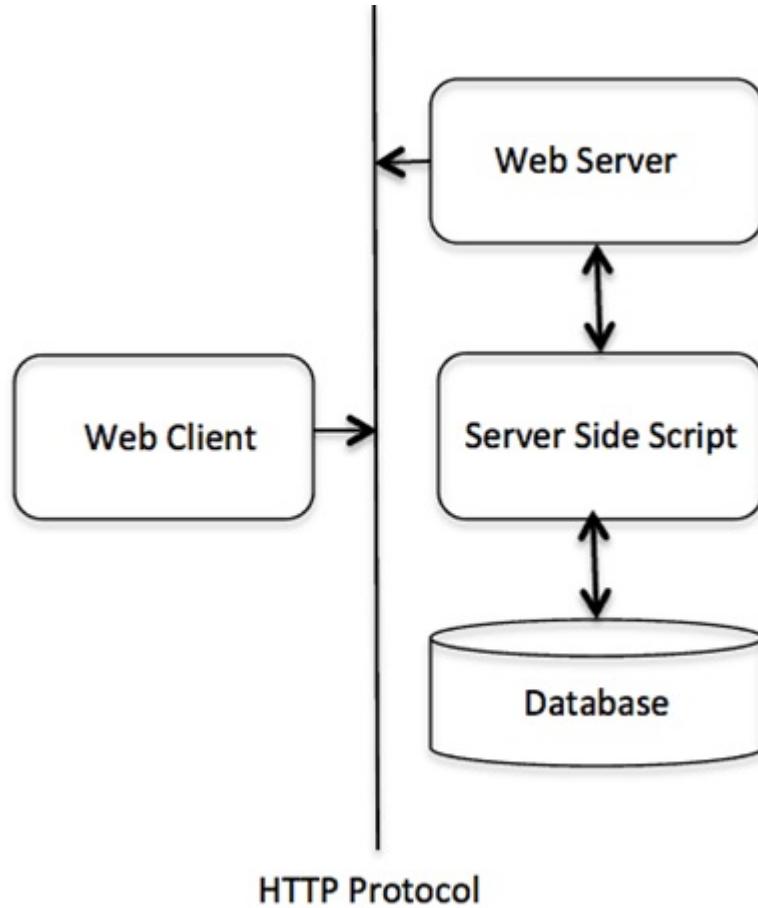
## Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

## CGI Architecture Diagram



## Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as. **cgi**, but you can keep your files with python extension **.py** as well.

By default, the Linux server is configured to run only the scripts in the cgi-bin directory in /var/www. If you want to specify any other directory to run your CGI scripts, comment the following lines in the httpd.conf file –

```
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
    Options All
</Directory>
```

The following line should also be added for apache server to treat .py file as cgi script.

```
AddHandler cgi-script .py
```

Here, we assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell, etc.

## First CGI Program

Here is a simple link, which is linked to a CGI script called `hello.py`. This file is kept in `/var/www/cgi-bin` directory and it has following content. Before running your CGI program, make sure you have change mode of file using **chmod 755 hello.py** UNIX command to make file executable.

```
print ("Content-type:text/html\r\n\r\n")
print ('<html>')
print ('<head>')
print ('<title>Hello Word - First CGI Program</title>')
print ('</head>')
print ('<body>')
print ('<h2>Hello Word! This is my first CGI program</h2>')
print ('</body>')
print ('</html>')
```

**Note** – First line in the script must be the path to Python executable. It appears as a comment in Python program, but it is called shebang line.

In Linux, it should be `#!/usr/bin/python3`.

In Windows, it should be `#!c:/python311/python.exe`.

Enter the following URL in your browser –

```
http://localhost/cgi-bin/hello.py
```

Hello Word! This is my first CGI program

This `hello.py` script is a simple Python script, which writes its output on STDOUT file, i.e., screen. There is one important and extra feature available which is first line to be printed **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and it specifies the content type to be displayed on the browser screen.

By now you must have understood basic concept of CGI and you can write many complicated CGI programs using Python. This script can interact with any other external system also to exchange information such as RDBMS.

## HTTP Header

The line **Content-type:text/html\r\n\r\n** is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form –

HTTP Field Name: Field Content

For Example

Content-type: text/html\r\n\r\n

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

| Sr.No. | Header & Description                                                                                                                                                                                    |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Content-type:</b><br>A MIME string defining the format of the file being returned. Example is Content-type:text/html                                                                                 |
| 2      | <b>Expires: Date</b><br>The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT. |
| 3      | <b>Location: URL</b><br>The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file.                                                                |
| 4      | <b>Last-modified: Date</b><br>The date of last modification of the resource.                                                                                                                            |
| 5      | <b>Content-length: N</b><br>The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.                                             |
| 6      | <b>Set-Cookie: String</b><br>Set the cookie passed through the string                                                                                                                                   |

## CGI Environment Variables

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

| Sr.No. | Variable Name & Description                                                                                                                                          |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>CONTENT_TYPE</b><br>The data type of the content. Used when the client is sending attached content to the server. For example, file upload.                       |
| 2      | <b>CONTENT_LENGTH</b><br>The length of the query information. It is available only for POST requests.                                                                |
| 3      | <b>HTTP_COOKIE</b><br>Returns the set cookies in the form of key & value pair.                                                                                       |
| 4      | <b>HTTP_USER_AGENT</b><br>The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.      |
| 5      | <b>PATH_INFO</b><br>The path for the CGI script.                                                                                                                     |
| 6      | <b>QUERY_STRING</b><br>The URL-encoded information that is sent with GET method request.                                                                             |
| 7      | <b>REMOTE_ADDR</b><br>The IP address of the remote host making the request. This is useful for logging or for authentication.                                        |
| 8      | <b>REMOTE_HOST</b><br>The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address. |
| 9      | <b>REQUEST_METHOD</b><br>The method used to make the request. The most common methods are GET and POST.                                                              |
| 10     | <b>SCRIPT_FILENAME</b><br>The full path to the CGI script.                                                                                                           |
| 11     | <b>SCRIPT_NAME</b><br>The name of the CGI script.                                                                                                                    |
| 12     | <b>SERVER_NAME</b><br>The server's hostname or IP Address                                                                                                            |
| 13     | <b>SERVER_SOFTWARE</b><br>The name and version of the software the server is running.                                                                                |

Here is small CGI program to list out all the CGI variables. Click this link to see the result  
[Get Environment](#)

```
import os

print ("Content-type: text/html\r\n\r\n");
print ("<font size=+1>Environment</font><br>");
for param in os.environ.keys():
    print ("<b>%20s</b>: %s<br>" % (param, os.environ[param]))
```

## GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

### Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows –

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

- The GET method is the default method to pass information from the browser to the web server and it produces a long string that appears in your browser's Location:box.
- Never use GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limitation: only 1024 characters can be sent in a request string.
- The GET method sends information using QUERY\_STRING header and will be accessible in your CGI Program through QUERY\_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

### Simple URL Example: Get Method

Here is a simple URL, which passes two values to hello\_get.py program using GET method.

```
/cgi-bin/hello_get.py?first_name=Malhar&last_name=Lathkar
```

Given below is the **hello\_get.py** script to handle the input given by web browser. We are going to use the cgi module, which makes it very easy to access the passed information –

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print ("Content-type:text/html")
print()
print ("<html>")
print ('<head>')
print ("<title>Hello - Second CGI Program</title>")
print ('</head>')
print ('<body>')
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ('</body>')
print ('</html>')
```

This would generate the following result –

Hello Malhar Lathkar

## Simple FORM Example:GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

```
<form action = "/cgi-bin/hello_get.py" method = "get">
    First Name: <input type = "text" name = "first_name"> <br />

    Last Name: <input type = "text" name = "last_name" />
```

```
<input type = "submit" value = "Submit" />
</form>
```

Here is the actual output of the above form, you enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

## Passing Information Using POST Method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Below is same hello\_get.py script which handles GET as well as POST method.

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

Let us take again same example as above which passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

```
<form action = "/cgi-bin/hello_get.py" method = "post">
First Name: <input type = "text" name = "first_name"><br />
```

```
Last Name: <input type = "text" name = "last_name" />

<input type = "submit" value = "Submit" />
</form>
```

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

## Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes –

```
<form action = "/cgi-bin/checkbox.cgi" method = "POST" target = "_blank">
    <input type = "checkbox" name = "maths" value = "on" /> Maths
    <input type = "checkbox" name = "physics" value = "on" /> Physics
    <input type = "submit" value = "Select Subject" />
</form>
```

The result of this code is the following form –

Maths  Physics

Below is checkbox.cgi script to handle input given by web browser for checkbox button.

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('maths'):
    math_flag = "ON"
else:
    math_flag = "OFF"

if form.getvalue('physics'):
    physics_flag = "ON"
else:
```

```

physics_flag = "OFF"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Checkbox - Third CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> CheckBox Maths is : %s</h2>" % math_flag
print "<h2> CheckBox Physics is : %s</h2>" % physics_flag
print "</body>"
print "</html>"

```

## Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio buttons –

```

<form action = "/cgi-bin/radiobutton.py" method = "post" target = "_blank">
    <input type = "radio" name = "subject" value = "maths" /> Maths
    <input type = "radio" name = "subject" value = "physics" /> Physics
    <input type = "submit" value = "Select Subject" />
</form>

```

The result of this code is the following form –

Maths  Physics

Below is radiobutton.py script to handle input given by web browser for radio button –

```

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('subject'):
    subject = form.getvalue('subject')
else:
    subject = "Not set"

```

```

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Radio - Fourth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"

```

## Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box –

```

<form action = "/cgi-bin/textarea.py" method = "post" target = "_blank">
    <textarea name = "textcontent" cols = "40" rows = "4">
        Type your text here...
    </textarea>
    <input type = "submit" value = "Submit" />
</form>

```

The result of this code is the following form –

Type your text here...

Submit

Below is textarea.cgi script to handle input given by web browser –

```

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('textcontent'):
    text_content = form.getvalue('textcontent')
else:
    text_content = "Not entered"

```

```
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Entered Text Content is %s</h2>" % text_content
print "</body>"
```

## Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box –

```
<form action = "/cgi-bin/dropdown.py" method = "post" target = "_blank">
    <select name = "dropdown">
        <option value = "Maths" selected>Maths</option>
        <option value = "Physics">Physics</option>
    </select>
    <input type = "submit" value = "Submit"/>
</form>
```

The result of this code is the following form –

Maths ▾ Submit

Below is dropdown.py script to handle input given by web browser.

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('dropdown'):
    subject = form.getvalue('dropdown')
else:
    subject = "Not entered"
```

```
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Dropdown Box - Sixth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"
```

## Using Cookies in CGI

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

## How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields –

- **Expires** – The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** – The domain name of your site.
- **Path** – The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure** – If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name = Value** – Cookies are set and retrieved in the form of key and value pairs.

## Setting up Cookies

It is very easy to send cookies to browser. These cookies are sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. Setting the cookies is done as follows –

```
print "Set-Cookie:UserID = XYZ;\r\n"
print "Set-Cookie:Password = XYZ123;\r\n"
print "Set-Cookie:Expires = Tuesday, 31-Dec-2007 23:12:40 GMT;\r\n"
print "Set-Cookie:Domain = www.tutorialspoint.com;\r\n"
print "Set-Cookie:Path = /perl;\r\n"
print "Content-type:text/html\r\n\r\n"
.....Rest of the HTML Content....
```

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

It is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line "**Content-type:text/html\r\n\r\n**".

## Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable **HTTP\_COOKIE** and they will have following form –

```
key1 = value1;key2 = value2;key3 = value3....
```

Here is an example of how to retrieve cookies.

```
# Import modules for CGI handling
from os import environ
import cgi, cgitb

if environ.has_key('HTTP_COOKIE'):
    for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
        (key, value) = split(cookie, '=')
        if key == "UserID":
            user_id = value

        if key == "Password":
            password = value

print "User ID = %s" % user_id
print "Password = %s" % password
```

This produces the following result for the cookies set by above script –

```
User ID = XYZ  
Password = XYZ123
```

## File Upload Example

To upload a file, the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type creates a "Browse" button.

```
<html>  
  <body>  
    <form enctype = "multipart/form-data" action = "save_file.py" method = "post">  
      <p>File: <input type = "file" name = "filename" /></p>  
      <p><input type = "submit" value = "Upload" /></p>  
    </form>  
  </body>  
</html>
```

The result of this code is the following form –

```
File:  Choose File No file chosen  

```

Above example has been disabled intentionally to save people uploading file on our server, but you can try above code with your server.

Here is the script **save\_file.py** to handle file upload –

```
import cgi, os  
import cgitb; cgitb.enable()  
  
form = cgi.FieldStorage()  
  
# Get filename here.  
fileitem = form['filename']  
  
# Test if the file was uploaded  
if fileitem.filename:  
    # strip leading path from file name to avoid  
    # directory traversal attacks  
    fn = os.path.basename(fileitem.filename)
```

```

open('/tmp/' + fn, 'wb').write(fileitem.file.read())

message = 'The file "' + fn + '" was uploaded successfully'

else:
    message = 'No file was uploaded'

print """
Content-Type: text/html\n
<html>
<body>
<p>%s</p>
</body>
</html>
""" % (message,)

```

If you run the above script on Unix/Linux, then you need to take care of replacing file separator as follows, otherwise on your windows machine above open() statement should work fine.

```
fn = os.path.basename(fileitem.filename.replace("\\", "/"))
```

## How To Raise a "File Download" Dialog Box?

Sometimes, it is desired that you want to give option where a user can click a link and it will pop up a "File Download" dialogue box to the user instead of displaying actual content. This is very easy and can be achieved through HTTP header. This HTTP header is be different from the header mentioned in previous section.

For example, if you want make a **FileName** file downloadable from a given link, then its syntax is as follows –

```

# HTTP Header
print "Content-Type:application/octet-stream; name = \"FileName\"\r\n";
print "Content-Disposition: attachment; filename = \"FileName\"\r\n\r\n";

# Actual File Content will go here.
fo = open("foo.txt", "rb")

str = fo.read();
print str

```

```
# Close opend file  
fo.close()
```

# Python - XML Processing

XML is a portable, open-source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

## What is XML?

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is extremely useful for keeping track of small to medium amounts of data without requiring an SQL- based backbone.

XML Parser Architectures and APIs.

The Python standard library provides a minimal but useful set of interfaces to work with XML. All the submodules for XML processing are available in the `xml` package.

- **`xml.etree.ElementTree`** – the ElementTree API, a simple and lightweight XML processor
- **`xml.dom`** – the DOM API definition.
- **`xml.dom.minidom`** – a minimal DOM implementation.
- **`xml.dom.pulldom`** – support for building partial DOM trees.
- **`xml.sax`** – SAX2 base classes and convenience functions.
- **`xml.parsers.expat`** – the Expat parser binding.

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

- **Simple API for XML (SAX)** – Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from the disk and the entire file is never stored in the memory.
- **Document Object Model (DOM)** – This is a World Wide Web Consortium recommendation wherein the entire file is read into the memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously cannot process information as fast as DOM, when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on many small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

For all our XML code examples, let us use a simple XML file **movies.xml** as an input –

```
<collection shelf="New Arrivals">
  <movie title="Enemy Behind">
    <type>War, Thriller</type>
    <format>DVD</format>
    <year>2003</year>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Talk about a US-Japan war</description>
  </movie>
  <movie title="Transformers">
    <type>Anime, Science Fiction</type>
    <format>DVD</format>
    <year>1989</year>
    <rating>R</rating>
    <stars>8</stars>
    <description>A schientific fiction</description>
  </movie>
  <movie title="Trigun">
    <type>Anime, Action</type>
    <format>DVD</format>
    <episodes>4</episodes>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Vash the Stampede!</description>
  </movie>
  <movie title="Ishtar">
    <type>Comedy</type>
    <format>VHS</format>
    <rating>PG</rating>
    <stars>2</stars>
    <description>Viewable boredom</description>
  </movie>
</collection>
```

## Parsing XML with SAX APIs

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing `xml.sax.ContentHandler`.

Your ContentHandler handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods `startDocument` and `endDocument` are called at the start and the end of the XML file. The method `characters(text)` is passed the character data of the XML file via the parameter `text`.

The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods `startElement(tag, attributes)` and `endElement(tag)` are called; otherwise, the corresponding methods `startElementNS` and `endElementNS` are called. Here, `tag` is the element tag, and `attributes` is an `Attributes` object.

Here are other important methods to understand before proceeding –

### The `make_parser` Method

The following method creates a new parser object and returns it. The parser object created will be of the first parser type, the system finds.

```
xml.sax.make_parser( [parser_list] )
```

Here is the detail of the parameters –

- **parser\_list** – The optional argument consisting of a list of parsers to use, which must all implement the `make_parser` method.

### The `parse` Method

The following method creates a SAX parser and uses it to parse a document.

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler] )
```

Here are the details of the parameters –

- **xmlfile** – This is the name of the XML file to read from.
- **contenthandler** – This must be a ContentHandler object.
- **errorhandler** – If specified, errorhandler must be a SAX ErrorHandler object.

## The parseString Method

There is one more method to create a SAX parser and to parse the specified XML string.

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

Here are the details of the parameters –

- **xmlstring** – This is the name of the XML string to read from.
- **contenthandler** – This must be a ContentHandler object.
- **errorhandler** – If specified, errorhandler must be a SAX ErrorHandler object.

## Example

```
import xml.sax
class MovieHandler( xml.sax.ContentHandler ):
    def __init__(self):
        self.CurrentData = ""
        self.type = ""
        self.format = ""
        self.year = ""
        self.rating = ""
        self.stars = ""
        self.description = ""

    # Call when an element starts
    def startElement(self, tag, attributes):
        self.CurrentData = tag
        if tag == "movie":
            print ("*****Movie*****")
            title = attributes["title"]
            print ("Title:", title)

    # Call when an elements ends
    def endElement(self, tag):
```

```
if self.CurrentData == "type":
    print ("Type:", self.type)
elif self.CurrentData == "format":
    print ("Format:", self.format)
elif self.CurrentData == "year":
    print ("Year:", self.year)
elif self.CurrentData == "rating":
    print ("Rating:", self.rating)
elif self.CurrentData == "stars":
    print ("Stars:", self.stars)
elif self.CurrentData == "description":
    print ("Description:", self.description)
self.CurrentData = ""

# Call when a character is read
def characters(self, content):
    if self.CurrentData == "type":
        self.type = content
    elif self.CurrentData == "format":
        self.format = content
    elif self.CurrentData == "year":
        self.year = content
    elif self.CurrentData == "rating":
        self.rating = content
    elif self.CurrentData == "stars":
        self.stars = content
    elif self.CurrentData == "description":
        self.description = content

if ( __name__ == "__main__"):

    # create an XMLReader
    parser = xml.sax.make_parser()

    # turn off namespaces
    parser.setFeature(xml.sax.handler.feature_namespaces, 0)

    # override the default ContextHandler
    Handler = MovieHandler()
    parser.setContentHandler( Handler )

    parser.parse("movies.xml")
```

This would produce the following result –

\*\*\*\*\*Movie\*\*\*\*\*

Title: Enemy Behind

Type: War, Thriller

Format: DVD

Year: 2003

Rating: PG

Stars: 10

Description: Talk about a US-Japan war

\*\*\*\*\*Movie\*\*\*\*\*

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Year: 1989

Rating: R

Stars: 8

Description: A schientific fiction

\*\*\*\*\*Movie\*\*\*\*\*

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Stars: 10

Description: Vash the Stampede!

\*\*\*\*\*Movie\*\*\*\*\*

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Stars: 2

Description: Viewable boredom

For a complete detail on SAX API documentation, please refer to standard [Python SAX APIs](#).

## Parsing XML with DOM APIs

The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying the XML documents.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to load an XML document quickly and to create a minidom object using the `xml.dom` module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

The sample phrase calls the `parse( file [,parser] )` function of the minidom object to parse the XML file, designated by file into a DOM tree object.

```
from xml.dom.minidom import parse
import xml.dom.minidom

# Open XML document using minidom parser
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
    print ("Root element : %s" % collection.getAttribute("shelf"))

# Get all the movies in the collection
movies = collection.getElementsByTagName("movie")

# Print detail of each movie.
for movie in movies:
    print ("*****Movie*****")
    if movie.hasAttribute("title"):
        print ("Title: %s" % movie.getAttribute("title"))

    type = movie.getElementsByTagName('type')[0]
    print ("Type: %s" % type.childNodes[0].data)
    format = movie.getElementsByTagName('format')[0]
    print ("Format: %s" % format.childNodes[0].data)
    rating = movie.getElementsByTagName('rating')[0]
    print ("Rating: %s" % rating.childNodes[0].data)
    description = movie.getElementsByTagName('description')[0]
    print ("Description: %s" % description.childNodes[0].data)
```

This would produce the following **output** –

Root element : New Arrivals

\*\*\*\*\*Movie\*\*\*\*\*

Title: Enemy Behind

Type: War, Thriller

Format: DVD

Rating: PG

Description: Talk about a US-Japan war

\*\*\*\*\*Movie\*\*\*\*\*

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Rating: R

Description: A schientific fiction

\*\*\*\*\*Movie\*\*\*\*\*

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Description: Vash the Stampede!

\*\*\*\*\*Movie\*\*\*\*\*

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Description: Viewable boredom

For a complete detail on DOM API documentation, please refer to standard [Python DOM APIs](#).

## ElementTree XML API

The `xml` package has an `ElementTree` module. This is a simple and lightweight XML processor API.

XML is a tree-like hierarchical data format. The '`ElementTree`' in this module treats the whole XML document as a tree. The '`Element`' class represents a single node in this tree. Reading and writing operations on XML files are done on the `ElementTree` level.

Interactions with a single XML element and its sub-elements are done on the `Element` level.

## Create an XML File

The tree is a hierarchical structure of elements starting with root followed by other elements. Each element is created by using the `Element()` function of this module.

```
import xml.etree.ElementTree as et
e=et.Element('name')
```

Each element is characterized by a tag and attrib attribute which is a dict object. For tree's starting element, attrib is an empty dictionary.

```
>>> root=xml.Element('employees')
>>> root.tag
'employees'
>>> root.attrib
{}
```

You may now set up one or more child elements to be added under the root element. Each child may have one or more sub elements. Add them using the SubElement() function and define its text attribute.

```
child=xml.Element("employee")
nm = xml.SubElement(child, "name")
nm.text = student.get('name')
age = xml.SubElement(child, "salary")
age.text = str(student.get('salary'))
```

Each child is added to root by append() function as –

```
root.append(child)
```

After adding required number of child elements, construct a tree object by elementTree() function –

```
tree = et.ElementTree(root)
```

The entire tree structure is written to a binary file by tree object's write() function –

```
f=open('employees.xml', "wb")
tree.write(f)
```

## Example

In this example, a tree is constructed out of a list of dictionary items. Each dictionary item holds key-value pairs describing a student data structure. The tree so constructed is written to 'myfile.xml'

```

import xml.etree.ElementTree as et
employees=[{'name':'aaa','age':21,'sal':5000},{'name':xyz,'age':22,'sal':6000}]
root = et.Element("employees")
for employee in employees:
    child=xml.Element("employee")
    root.append(child)
    nm = xml.SubElement(child, "name")
    nm.text = student.get('name')
    age = xml.SubElement(child, "age")
    age.text = str(student.get('age'))
    sal=xml.SubElement(child, "sal")
    sal.text=str(student.get('sal'))
tree = et.ElementTree(root)
with open('employees.xml', "wb") as fh:
    tree.write(fh)

```

The 'myfile.xml' is stored in current working directory.

```
<employees><employee><name>aaa</name><age>21</age><sal>5000</sal></employee><empl>
```

## Parse an XML File

Let us now read back the 'myfile.xml' created in above example. For this purpose, following functions in ElementTree module will be used –

**ElementTree()** – This function is overloaded to read the hierarchical structure of elements to a tree objects.

```
tree = et.ElementTree(file='students.xml')
```

**getroot()** – This function returns root element of the tree.

```
root = tree.getroot()
```

You can obtain the list of sub-elements one level below of an element.

```
children = list(root)
```

In the following example, elements and sub-elements of the 'myfile.xml' are parsed into a list of dictionary items.

## Example

```
</> Open Compiler

import xml.etree.ElementTree as et
tree = et.ElementTree(file='employees.xml')
root = tree.getroot()
employees=[]
    children = list(root)
for child in children:
    employee={}
    pairs = list(child)
    for pair in pairs:
        employee[pair.tag]=pair.text
    employees.append(employee)
print (employees)
```

It will produce the following **output** –

```
[{'name': 'aaa', 'age': '21', 'sal': '5000'}, {'name': 'xyz', 'age': '22', 'sal': '6000'}]
```

## Modify an XML file

We shall use `iter()` function of `Element`. It creates a tree iterator for given tag with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order.

Let us build iterator for all 'marks' subelements and increment text of each sal tag by 100.

```
</> Open Compiler

import xml.etree.ElementTree as et
tree = et.ElementTree(file='students.xml')
root = tree.getroot()
for x in root.iter('sal'):
    s=int (x.text)
    s=s+100
    x.text=str(s)
with open("employees.xml", "wb") as fh:
    tree.write(fh)
```

Our 'employees.xml' will now be modified accordingly. We can also use set() to update value of a certain key.

```
x.set(marks, str(mark))
```

## Python - GUI Programming

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below.

- **Tkinter** – Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look at this option in this chapter.
- **wxPython** – This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython [here](#).
- **PyQt** – This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt5 [here](#).
- **PyGTK** – PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available [here](#).
- **PySimpleGUI** – PySimpleGui is an open source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide and WxPython toolkits. For a detailed PySimpleGUI tutorial, click [here](#).
- **Pygame** – Pygame is a popular Python library used for developing video games. It is free, open source and cross-platform wrapper around Simple DirectMedia Library (SDL). For a comprehensive tutorial on Pygame, [visit](#) this link.
- **Jython** – Jython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machine <http://www.jython.org>.

There are many other interfaces available, which you can find them on the net.

## Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

The tkinter package includes following modules –

- **Tkinter** – Main Tkinter module.
- **tkinter.colorchooser** – Dialog to let the user choose a color.

- **tkinter.commondialog** – Base class for the dialogs defined in the other modules listed here.
- **tkinter.filedialog** – Common dialogs to allow the user to specify a file to open or save.
- **tkinter.font** – Utilities to help work with fonts.
- **tkinter.messagebox** – Access to standard Tk dialog boxes.
- **tkinter.scrolledtext** – Text widget with a vertical scroll bar built in.
- **tkinter.simpledialog** – Basic dialogs and convenience functions.
- **tkinter.ttk** – Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main tkinter module.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps.

- Import the Tkinter module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

## Example

```
# note that module name has changed from Tkinter in Python 2
# to tkinter in Python 3

import tkinter
top = tkinter.Tk()

# Code to add widgets will go here...
top.mainloop()
```

This would create a following window –



When the program becomes more complex, using an object-oriented programming approach makes the code more organized.

```
import tkinter as tk
class App(tk.Tk):
    def __init__(self):
        super().__init__()

app = App()
app.mainloop()
```

## Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table –

| Sr.No. | Operator & Description                                                                                                                             |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Button</b><br>The Button widget is used to display the buttons in your application.                                                             |
| 2      | <b>Canvas</b><br>The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.                     |
| 3      | <b>Checkbutton</b><br>The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time. |
| 4      | <b>Entry</b>                                                                                                                                       |

|    |                                                                                                                                                      |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | The Entry widget is used to display a single-line text field for accepting values from a user.                                                       |
| 5  | <b>Frame</b><br>The Frame widget is used as a container widget to organize other widgets.                                                            |
| 6  | <b>Label</b><br>The Label widget is used to provide a single-line caption for other widgets. It can also contain images.                             |
| 7  | <b>Listbox</b><br>The Listbox widget is used to provide a list of options to a user.                                                                 |
| 8  | <b>Menubutton</b><br>The Menubutton widget is used to display menus in your application.                                                             |
| 9  | <b>Menu</b><br>The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.                        |
| 10 | <b>Message</b><br>The Message widget is used to display multiline text fields for accepting values from a user.                                      |
| 11 | <b>Radiobutton</b><br>The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | <b>Scale</b><br>The Scale widget is used to provide a slider widget.                                                                                 |
| 13 | <b>Scrollbar</b><br>The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.                                 |
| 14 | <b>Text</b><br>The Text widget is used to display text in multiple lines.                                                                            |
| 15 | <b>Toplevel</b><br>The Toplevel widget is used to provide a separate window container.                                                               |
| 16 | <b>Spinbox</b><br>The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.   |
| 17 | <b>PanedWindow</b><br>A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.                 |

|    |                                                                                                                               |
|----|-------------------------------------------------------------------------------------------------------------------------------|
|    | <b>LabelFrame</b>                                                                                                             |
| 18 | A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| 19 | <b>tkMessageBox</b><br>This module is used to display message boxes in your applications.                                     |

Let us study these widgets in detail.

## Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

Let us study them briefly –

## Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- **The pack() Method** – This geometry manager organizes widgets in blocks before placing them in the parent widget.
- **The grid() Method** – This geometry manager organizes widgets in a table-like structure in the parent widget.
- **The place() Method** – This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly –

## SimpleDialog

The simpledialog module in tkinter package includes a dialog class and convenience functions for accepting user input through a modal dialog. It consists of a label, an entry widget and two buttons Ok and Cancel. These functions are –

- **askfloat(title, prompt, \*\*kw)** – Accepts a floating point number.
- **askinteger(title, prompt, \*\*kw)** – Accepts an integer input.
- **askstring(title, prompt, \*\*kw)** – Accepts a text input from the user.

The above three functions provide dialogs that prompt the user to enter a value of the desired type. If Ok is pressed, the input is returned, if Cancel is pressed, None is returned.

### askinteger

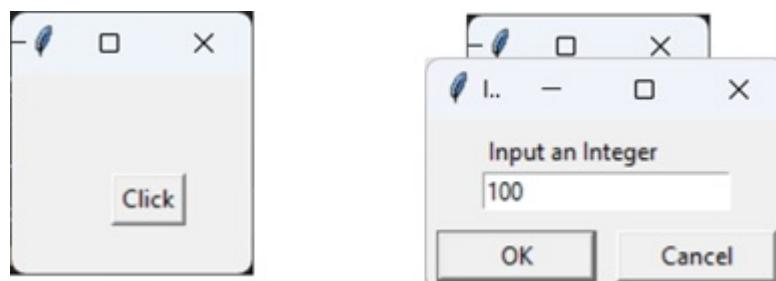
```
from tkinter.simpledialog import askinteger
from tkinter import *
from tkinter import messagebox
top = Tk()

top.geometry("100x100")
def show():
    num = askinteger("Input", "Input an Integer")
    print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** –



### askfloat

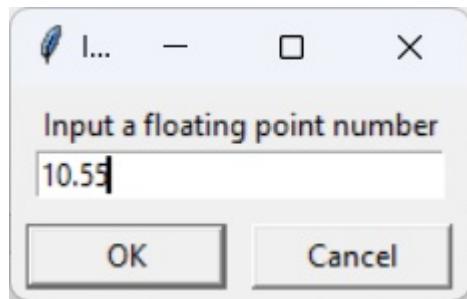
```
from tkinter.simpledialog import askfloat
from tkinter import *
top = Tk()

top.geometry("100x100")
def show():
    num = askfloat("Input", "Input a floating point number")
    print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** –



## askstring

```
from tkinter.simpledialog import askstring
from tkinter import *

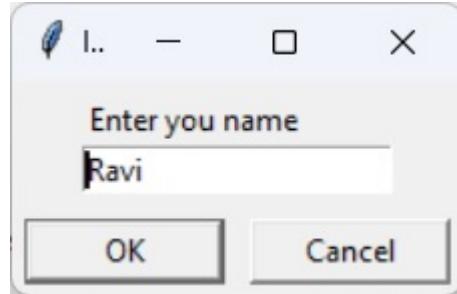
top = Tk()

top.geometry("100x100")
def show():
    name = askstring("Input", "Enter your name")
    print(name)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** –



## The FileDialog Module

The `filedialog` module in Tkinter package includes a `FileDialog` class. It also defines convenience functions that enable the user to perform open file, save file, and open directory activities.

- `filedialog.asksaveasfilename()`
- `filedialog.asksaveasfile()`
- `filedialog.askopenfilename()`
- `filedialog.askopenfile()`
- `filedialog.askdirectory()`
- `filedialog.askopenfilenames()`
- `filedialog.askopenfiles()`

### askopenfile

This function lets the user choose a desired file from the filesystem. The file dialog window has Open and Cancel buttons. The file name along with its path is returned when Ok is pressed, None if Cancel is pressed.

```
from tkinter.filedialog import askopenfile
from tkinter import *

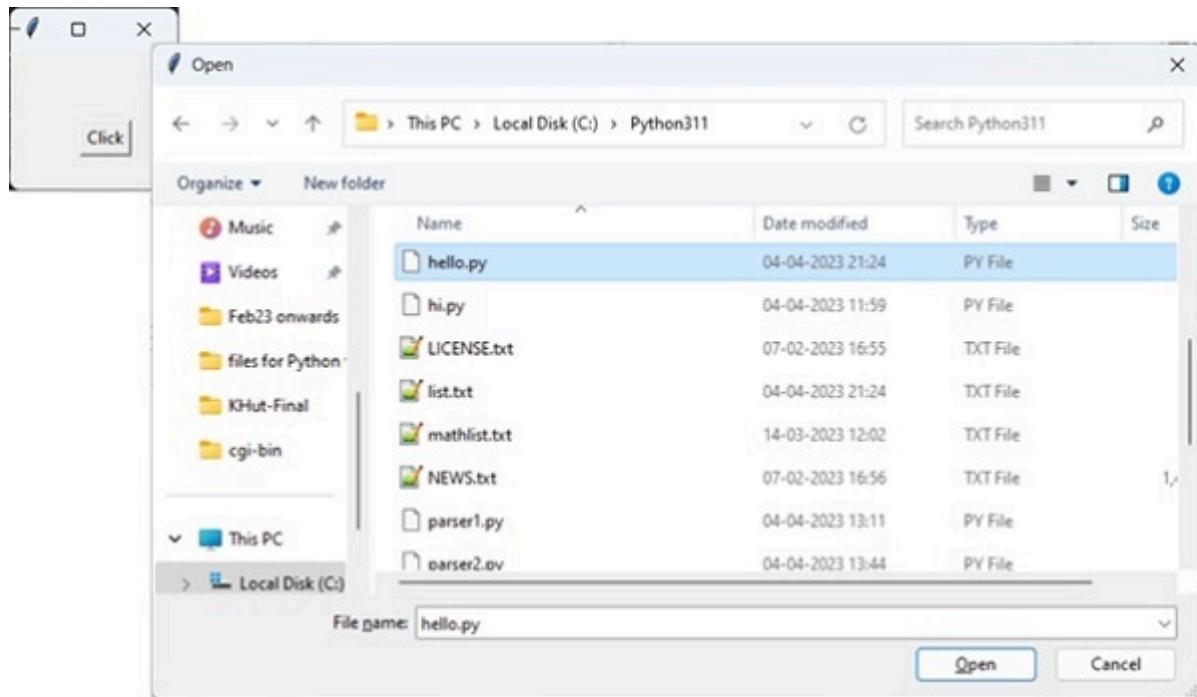
top = Tk()

top.geometry("100x100")
def show():
    filename = askopenfile()
    print(filename)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)
```

```
top.mainloop()
```

It will produce the following **output** –



## ColorChooser

The colorchooser module included in tkinter package has the feature of letting the user choose a desired color object through the color dialog. The `askcolor()` function presents with the color dialog with predefined color swatches and facility to choose custome color by setting RGB values. The dialog returns a tuple of RGB values of chosen color as well as its hex value.

```
from tkinter.colorchooser import askcolor
from tkinter import *

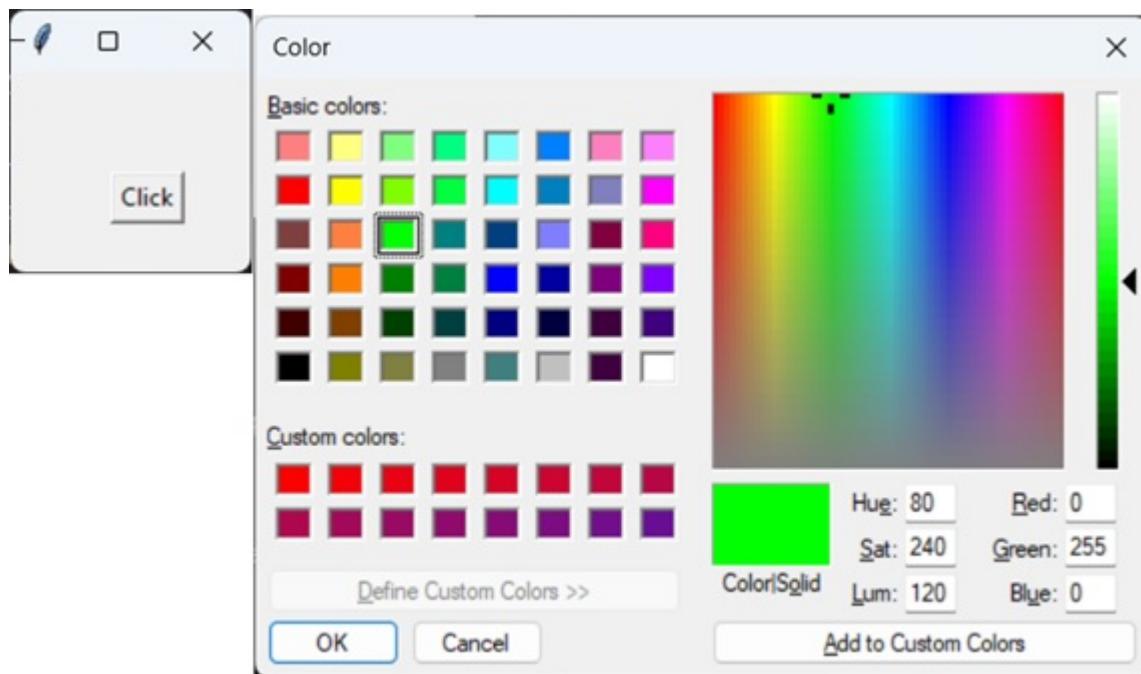
top = Tk()

top.geometry("100x100")
def show():
    color = askcolor()
    print(color)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following **output** –



```
((0, 255, 0), '#00ff00')
```

## ttk module

The term ttk stands from Tk Themed widgets. The ttk module was introduced with Tk 8.5 onwards. It provides additional benefits including anti-aliased font rendering under X11 and window transparency. It provides theming and styling support for Tkinter.

The ttk module comes bundled with 18 widgets, out of which 12 are already present in Tkinter. Importing ttk over-writes these widgets with new ones which are designed to have a better and more modern look across all platforms.

The 6 new widgets in ttk are, the Combobox, Separator, Sizegrip, Treeview, Notebook and ProgressBar.

To override the basic Tk widgets, the import should follow the Tk import –

```
from tkinter import *
from tkinter.ttk import *
```

The original Tk widgets are automatically replaced by tkinter.ttk widgets. They are Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar.

New widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget

options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

The new widgets in `ttk` module are –

- **Notebook** – This widget manages a collection of "tabs" between which you can swap, changing the currently displayed window.
- **ProgressBar** – This widget is used to show progress or the loading process through the use of animations.
- **Separator** – Used to separate different widgets using a separator line.
- **Treeview** – This widget is used to group together items in a tree-like hierarchy. Each item has a textual label, an optional image, and an optional list of data values.
- **ComboBox** – Used to create a dropdown list of options from which the user can select one.
- **Sizegrip** – Creates a little handle near the bottom-right of the screen, which can be used to resize the window.

## Combobox Widget

The Python `ttk` Combobox presents a drop down list of options and displays them one at a time. It is a sub class of the widget `Entry`. Hence it inherits many options and methods from the `Entry` class.

### Syntax

```
from tkinter import ttk  
  
Combo = ttk.Combobox(master, values.....)
```

The `get()` function to retrieve the current value of the Combobox.

### Example

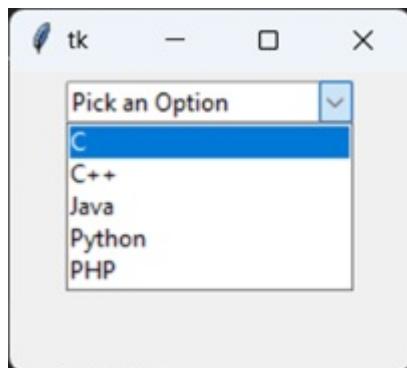
```
from tkinter import *  
from tkinter import ttk  
  
top = Tk()  
top.geometry("200x150")  
  
frame = Frame(top)
```

```
frame.pack()

langs = ["C", "C++", "Java",
          "Python", "PHP"]

Combo = ttk.Combobox(frame, values = langs)
Combo.set("Pick an Option")
Combo.pack(padx = 5, pady = 5)
top.mainloop()
```

It will produce the following **output** –



## Progressbar

The ttk Progressbar widget, and how it can be used to create loading screens or show the progress of a current task.

### Syntax

```
ttk.Progressbar(parent, orient, length, mode)
```

### Parameters

- **Parent** – The container in which the Progressbar is to be placed, such as root or a Tkinter frame.
- **Orient** – Defines the orientation of the Progressbar, which can be either vertical or horizontal.
- **Length** – Defines the width of the Progressbar by taking in an integer value.
- **Mode** – There are two options for this parameter, determinate and indeterminate.

### Example

The code given below creates a progressbar with three buttons which are linked to three different functions.

The first function increments the "value" or "progress" in the progressbar by 20. This is done with the step() function which takes an integer value to change progress amount. (Default is 1.0)

The second function decrements the "value" or "progress" in the progressbar by 20.

The third function prints out the current progress level in the progressbar.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
frame= ttk.Frame(root)
def increment():
    progressBar.step(20)

def decrement():
    progressBar.step(-20)

def display():
    print(progressBar["value"])

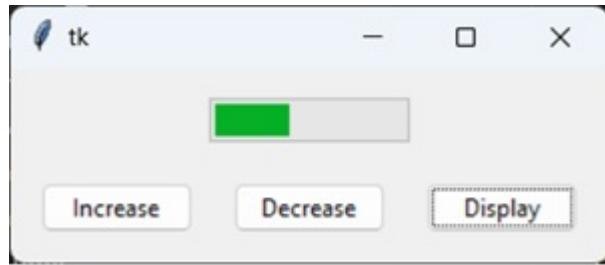
progressBar= ttk.Progressbar(frame, mode='determinate')
progressBar.pack(padx = 10, pady = 10)

button= ttk.Button(frame, text= "Increase", command= increment)
button.pack(padx = 10, pady = 10, side = tk.LEFT)

button= ttk.Button(frame, text= "Decrease", command= decrement)
button.pack(padx = 10, pady = 10, side = tk.LEFT)
button= ttk.Button(frame, text= "Display", command= display)
button.pack(padx = 10, pady = 10, side = tk.LEFT)

frame.pack(padx = 5, pady = 5)
root.mainloop()
```

It will produce the following **output** –



## Notebook

Tkinter ttk module has a new useful widget called Notebook. It is a collection of containers (e.g frames) which have many widgets as children inside.

Each "tab" or "window" has a tab ID associated with it, which is used to determine which tab to swap to.

You can swap between these containers like you would on a regular text editor.

## Syntax

```
notebook = ttk.Notebook(master, *options)
```

## Example

In this example, add 3 windows to our Notebook widget in two different ways. The first method involves the add() function, which simply appends a new tab to the end. The other method is the insert() function which can be used to add a tab to a specific position.

The add() function takes one mandatory parameter which is the container widget to be added, and the rest are optional parameters such as text (text to be displayed as tab title), image and compound.

The insert() function requires a tab\_id, which defines the location where it should be inserted. The tab\_id can be either an index value or it can be string literal like "end", which will append it to the end.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
nb = ttk.Notebook(root)

# Frame 1 and 2
frame1 = ttk.Frame(nb)
frame2 = ttk.Frame(nb)
```

```

label1 = ttk.Label(frame1, text = "This is Window One")
label1.pack(pady = 50, padx = 20)
label2 = ttk.Label(frame2, text = "This is Window Two")
label2.pack(pady = 50, padx = 20)

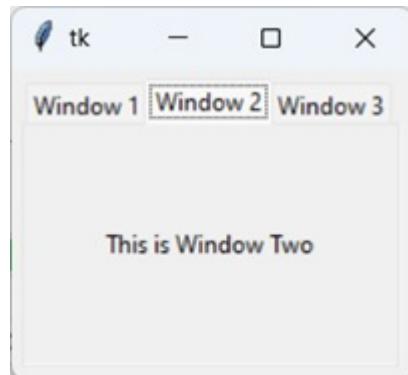
frame1.pack(fill= tk.BOTH, expand=True)
frame2.pack(fill= tk.BOTH, expand=True)
nb.add(frame1, text = "Window 1")
nb.add(frame2, text = "Window 2")

frame3 = ttk.Frame(nb)
label3 = ttk.Label(frame3, text = "This is Window Three")
label3.pack(pady = 50, padx = 20)
frame3.pack(fill= tk.BOTH, expand=True)
nb.insert("end", frame3, text = "Window 3")
nb.pack(padx = 5, pady = 5, expand = True)

root.mainloop()

```

It will produce the following **output** –



## Treeview

The Treeview widget is used to display items in a tabular or hierarchical manner. It has support for features like creating rows and columns for items, as well as allowing items to have children as well, leading to a hierarchical format.

## Syntax

```
tree = ttk.Treeview(container, **options)
```

## Options

| Sr.No. | Option & Description |
|--------|----------------------|
|--------|----------------------|

|   |                                                                                                                                                                                                                                                |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | <b>columns</b><br>A list of column names                                                                                                                                                                                                       |
| 2 | <b>displaycolumns</b><br>A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".                                               |
| 3 | <b>height</b><br>The number of rows visible.                                                                                                                                                                                                   |
| 4 | <b>padding</b><br>Specifies the internal padding for the widget. Can be either an integer or a list of 4 values.                                                                                                                               |
| 5 | <b>selectmode</b><br>One of "extended", "browse" or "none". If set to "extended" (default), multiple items can be selected. If "browse", only a single item can be selected at a time. If "none", the selection cannot be changed by the user. |
| 6 | <b>show</b><br>A list containing zero or more of the following values, specifying which elements of the tree to display. The default is "tree headings", i.e., show all elements.                                                              |

## Example

In this example we will create a simple Treeview ttk Widget and fill in some data into it. We have some data already stored in a list which will be reading and adding to the Treeview widget in our `read_data()` function.

We first need to define a list/tuple of column names. We have left out the column "Name" because there already exists a (default) column with a blank name.

We then assign that list/tuple to the `columns` option in Treeview, followed by defining the "headings", where the column is the actual column, whereas the heading is just the title of the column that appears when the widget is displayed. We give each a column a name. "#0" is the name of the default column.

The `tree.insert()` function has the following parameters –

- **Parent** – which is left as an empty string if there is none.
- **Position** – where we want to add the new item. To append, use tk.END
- **Iid** – which is the item ID used to later track the item in question.
- **Text** – to which we will assign the first value in the list (the name).

Value we will pass the the other 2 values we obtained from the list.

## The Complete Code

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import simpledialog

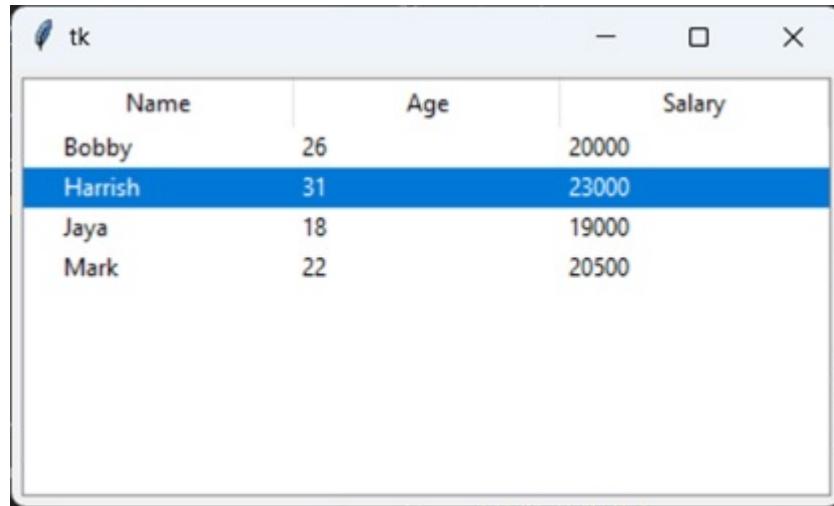
root = tk.Tk()
data = [
    ["Bobby", 26, 20000],
    ["Harrish", 31, 23000],
    ["Jaya", 18, 19000],
    ["Mark", 22, 20500],
]
index=0
def read_data():
    for index, line in enumerate(data):
        tree.insert('', tk.END, iid = index,
                    text = line[0], values = line[1:])
columns = ("age", "salary")

tree= ttk.Treeview(root, columns=columns ,height = 20)
tree.pack(padx = 5, pady = 5)

tree.heading('#0', text='Name')
tree.heading('age', text='Age')
tree.heading('salary', text='Salary')

read_data()
root.mainloop()
```

It will produce the following **output** –



A screenshot of a Tkinter application window titled "tk". Inside the window, there is a table with three columns: "Name", "Age", and "Salary". The table contains four rows of data: Bobby (Age 26, Salary 20000), Harrish (Age 31, Salary 23000, highlighted with a blue background), Jaya (Age 18, Salary 19000), and Mark (Age 22, Salary 20500). At the bottom-right corner of the window frame, there is a small arrow-shaped "Sizegrip" widget.

| Name    | Age | Salary |
|---------|-----|--------|
| Bobby   | 26  | 20000  |
| Harrish | 31  | 23000  |
| Jaya    | 18  | 19000  |
| Mark    | 22  | 20500  |

## Sizegrip

The Sizegrip widget is basically a small arrow-like grip that is typically placed at the bottom-right corner of the screen. Dragging the Sizegrip across the screen also resizes the container to which it is attached to.

### Syntax

```
sizegrip = ttk.Sizegrip(parent, **options)
```

### Example

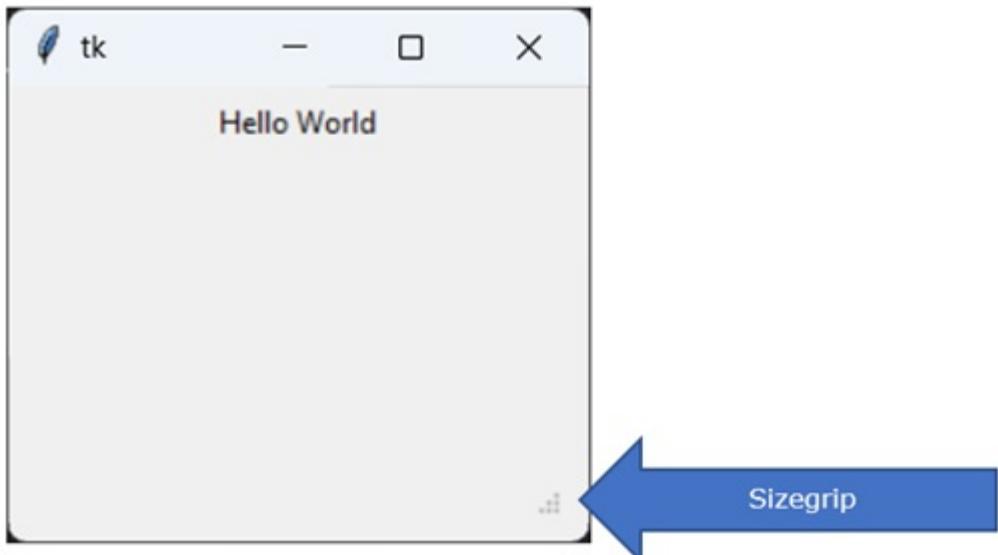
```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("100x100")

frame = ttk.Frame(root)
label = ttk.Label(root, text = "Hello World")
label.pack(padx = 5, pady = 5)
sizegrip = ttk.Sizegrip(frame)
sizegrip.pack(expand = True, fill = tk.BOTH, anchor = tk.SE)
frame.pack(padx = 10, pady = 10, expand = True, fill = tk.BOTH)

root.mainloop()
```

It will produce the following **output** –



## Separator

The ttk Separator widget is a very simple widget, that has just one purpose and that is to help "separate" widgets into groups/partitions by drawing a line between them. We can change the orientation of this line (separator) to either horizontal or vertical, and change its length/height.

## Syntax

```
separator = ttk.Separator(parent, **options)
```

The "orient", which can either be tk.VERTICAL or tk.HORIZONTAL, for a vertical and horizontal separator respectively.

## Example

Here we have created two Label widgets, and then created a Horizontal Separator between them.

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("200x150")

frame = ttk.Frame(root)

label = ttk.Label(frame, text = "Hello World")
label.pack(padx = 5)
```

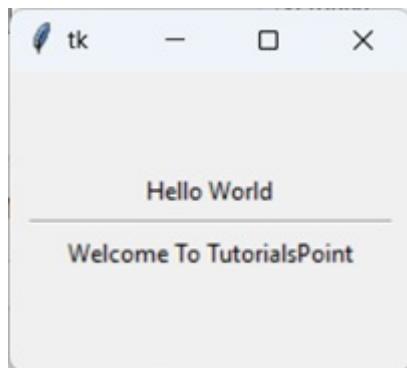
```
separator = ttk.Separator(frame, orient= tk.HORIZONTAL)
separator.pack(expand = True, fill = tk.X)

label = ttk.Label(frame, text = "Welcome To TutorialsPoint")
label.pack(padx = 5)

frame.pack(padx = 10, pady = 50, expand = True, fill = tk.BOTH)

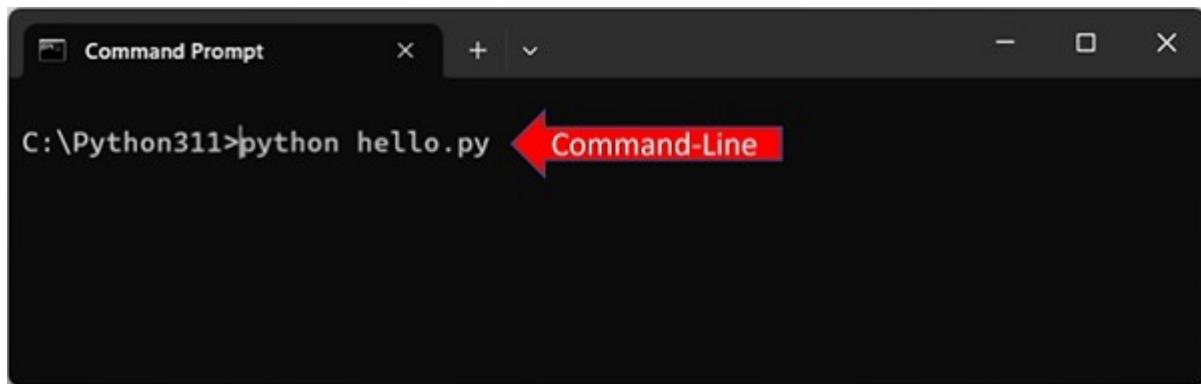
root.mainloop()
```

It will produce the following **output** –



## Python - Command-Line Arguments

To run a Python program, we execute the following command in the command prompt terminal of the operating system. For example, in windows, the following command is entered in Windows command prompt terminal.



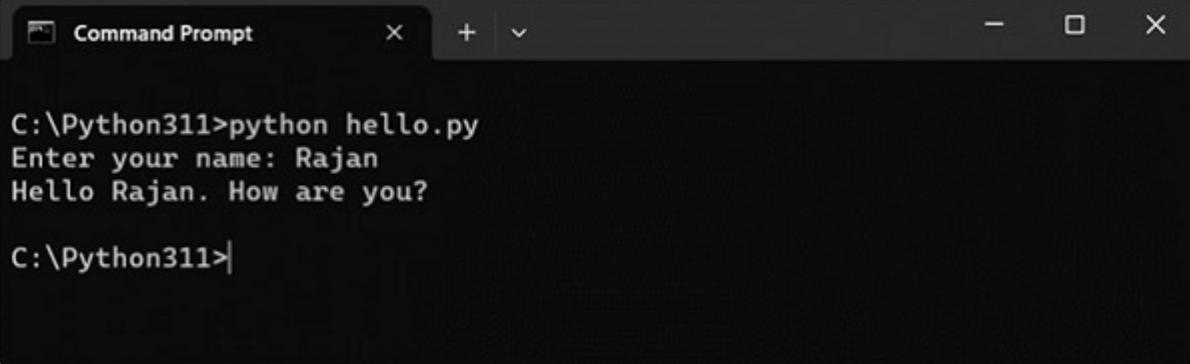
The line in front of the command prompt C:\> ( or \$ in case of Linux operating system) is called as command-line.

If the program needs to accept input from the user, Python's `input()` function is used. When the program is executed from command line, user input is accepted from the command terminal.

### Example

```
name = input("Enter your name: ")
print ("Hello {}. How are you?".format(name))
```

The program is run from the command prompt terminal as follows –



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the command "C:\Python311>python hello.py" being run, followed by the user input "Enter your name: Rajan", and the program's output "Hello Rajan. How are you?". The prompt "C:\Python311>" is visible at the bottom.

Very often, you may need to put the data to be used by the program in the command line itself and use it inside the program. An example of giving the data in the command line could be any DOS commands in Windows or Linux.

In Windows, you use the following DOS command to rename a file hello.py to hi.py.

```
C:\Python311>ren hello.py hi.py
```

In Linux you may use the mv command –

```
$ mv hello.py hi.py
```

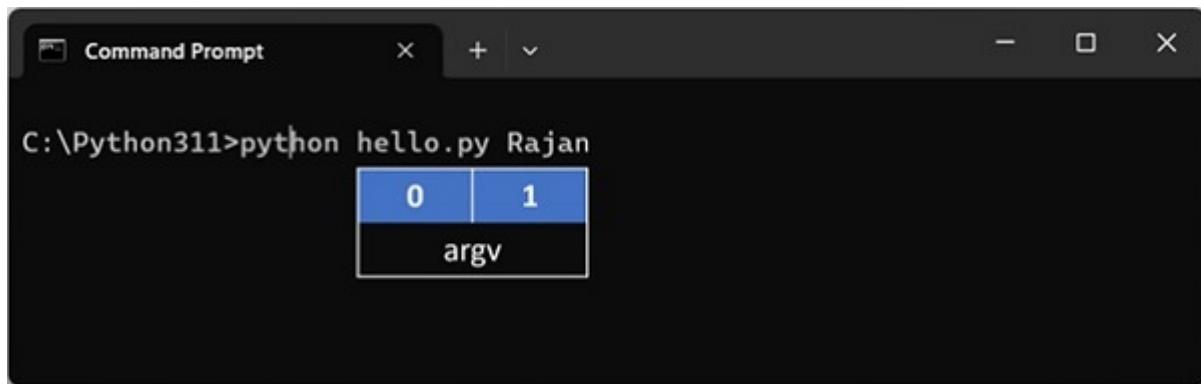
Here ren or mv are the commands which need the old and new file names. Since they are put in line with the command, they are called command-line arguments.

You can pass values to a Python program from command line. Python collects the arguments in a list object. Python's sys module provides access to any command-line arguments via the sys.argv variable. sys.argv is the list of command-line arguments and sys.argv[0] is the program i.e. the script name.

The hello.py script used input() function to accept user input after the script is run. Let us change it to accept input from command line.

```
import sys
print ('argument list', sys.argv)
name = sys.argv[1]
print ("Hello {}. How are you?".format(name))
```

Run the program from command-line as shown in the following figure –



The **output** is shown below –

```
C:\Python311>python hello.py Rajan
argument list ['hello.py', 'Rajan']
Hello Rajan. How are you?
```

The command-line arguments are always stored in string variables. To use them as numerics, you can convert them suitably with type conversion functions.

In the following example, two numbers are entered as command-line arguments. Inside the program, we use `int()` function to parse them as integer variables.

```
import sys
print ('argument list', sys.argv)
first = int(sys.argv[1])
second = int(sys.argv[2])
print ("sum = {}".format(first+second))
```

It will produce the following **output** –

```
C:\Python311>python hello.py 10 20
argument list ['hello.py', '10', '20']
sum = 30
```

Python's standard library includes a couple of useful modules to parse command line arguments and options –

- **getopt** – C-style parser for command line options.
- **argparse** – Parser for command-line options, arguments and sub-commands.

## The getopt Module

Python provides a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

## getopt.getopt method

This method parses the command line options and parameter list. Following is a simple syntax for this method –

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters –

- **args** – This is the argument list to be parsed.
- **options** – This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long\_options** – This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

This method returns a value consisting of two elements- the first is a list of (option, value) pairs, the second is a list of program arguments left after the option list was stripped.

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

## Exception getopt.GetoptError

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

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

## Example

Suppose we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows –

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

Here is the following script to test.py –

```

import sys, getopt

def main(argv):
    inputfile = ''
    outputfile = ''
    try:
        opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
    except getopt.GetoptError:
        print ('test.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print ('test.py -i <inputfile> -o <outputfile>')
            sys.exit()
        elif opt in ("-i", "--ifile"):
            inputfile = arg
        elif opt in ("-o", "--ofile"):
            outputfile = arg
    print ('Input file is ''', inputfile)
    print ('Output file is ''', outputfile)
if __name__ == "__main__":
    main(sys.argv[1:])

```

Now, run the above script as follows –

```

$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile -o outputfile
Input file is " inputfile
Output file is " outputfile

```

## The argparse Module

The **argparse** module provides tools for writing very easy to use command line interfaces. It handles how to parse the arguments collected in **sys.argv** list, automatically generate help and issues error message when invalid options are given.

First step to design the command line interface is to set up parser object. This is done by `ArgumentParser()` function in `argparse` module. The function can be given an explanatory string as `description` parameter.

To start with our script will be executed from command line without any arguments. Still use **parse\_args()** method of parser object, which does nothing because there aren't any arguments given.

```
import argparse  
parser=argparse.ArgumentParser(description="sample argument parser")  
args=parser.parse_args()
```

When the above script is run –

```
C:\Python311>python parser1.py  
C:\Python311>python parser1.py -h  
usage: parser1.py [-h]  
sample argument parser  
options:  
-h, --help show this help message and exit
```

The second command line usage gives **-help** option which produces a help message as shown. The **-help** parameter is available by default.

Now let us define an argument which is mandatory for the script to run and if not given script should throw error. Here we define argument 'user' by add\_argument() method.

```
import argparse  
parser=argparse.ArgumentParser(description="sample argument parser")  
parser.add_argument("user")  
args=parser.parse_args()  
if args.user=="Admin":  
    print ("Hello Admin")  
else:  
    print ("Hello Guest")
```

This script's help now shows one positional argument in the form of 'user'. The program checks if it's value is 'Admin' or not and prints corresponding message.

```
C:\Python311>python parser2.py --help  
usage: parser2.py [-h] user  
sample argument parser  
positional arguments:  
    user  
options:  
-h, --help show this help message and exit
```

Use the following command –

```
C:\Python311>python parser2.py Admin  
Hello Admin
```

But the following usage displays Hello Guest message.

```
C:\Python311>python parser2.py Rajan  
Hello Guest
```

## add\_argument() method

We can assign default value to an argument in add\_argument() method.

```
import argparse  
parser=argparse.ArgumentParser(description="sample argument parser")  
parser.add_argument("user", nargs='?', default="Admin")  
args=parser.parse_args()  
if args.user=="Admin":  
    print ("Hello Admin")  
else:  
    print ("Hello Guest")
```

Here nargs is the number of command-line arguments that should be consumed. '?'. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced.

By default, all arguments are treated as strings. To explicitly mention type of argument, use type parameter in the add\_argument() method. All Python data types are valid values of type.

```
import argparse  
parser=argparse.ArgumentParser(description="add numbers")  
parser.add_argument("first", type=int)  
parser.add_argument("second", type=int)  
args=parser.parse_args()  
x=args.first  
y=args.second  
z=x+y  
print ('addition of {} and {} = {}'.format(x,y,z))
```

It will produce the following **output** –

```
C:\Python311>python parser3.py 10 20
addition of 10 and 20 = 30
```

In the above examples, the arguments are mandatory. To add optional argument, prefix its name by double dash --. In following case surname argument is optional because it is prefixed by double dash (--surname).

```
import argparse
parser=argparse.ArgumentParser()
parser.add_argument("name")
parser.add_argument("--surname")
args=parser.parse_args()
print ("My name is ", args.name, end=' ')
if args.surname:
    print (args.surname)
```

A one letter name of argument prefixed by single dash acts as a short name option.

```
C:\Python311>python parser3.py Anup
My name is Anup
C:\Python311>python parser3.py Anup --surname Gupta
My name is Anup Gupta
```

If it is desired that an argument should value only from a defined list, it is defined as choices parameter.

```
import argparse
parser=argparse.ArgumentParser()
parser.add_argument("sub", choices=['Physics', 'Maths', 'Biology'])
args=parser.parse_args()
print ("My subject is ", args.sub)
```

Note that if value of parameter is not from the list, invalid choice error is displayed.

```
C:\Python311>python parser3.py Physics
My subject is Physics
C:\Python311>python parser3.py History
usage: parser3.py [-h] {Physics,Maths,Biology}
parser3.py: error: argument sub: invalid choice: 'History' (choose from
'Physics', 'Maths', 'Biology')
```

# Python - Docstrings

In Python, a docstring is a string literal that serves as the documentation of different Python objects such as functions, modules, class as well as its methods and packages. It is the first line in the definition of all these constructs and becomes the value of `__doc__` attribute.

## DocString of a Function

&lt;/&gt;

Open Compiler

```
def addition(x, y):
    '''This function returns the sum of two numeric arguments'''
    return x+y
print ("Docstring of addition function:", addition.__doc__)
```

It will produce the following **output** –

Docstring of addition function: This function returns the sum of two numeric arguments

The docstring can be written with single, double or triple quotation marks. However, most of the times you may want a descriptive text as the documentation, so using triple quotes is desirable.

All the built-in modules and functions have the `__doc__` property that returns their docstring.

## Docstring of math module

&lt;/&gt;

Open Compiler

```
import math

print ("Docstring of math module:", math.__doc__)
```

It will produce the following **output** –

Docstring of math module: This module provides access to the mathematical functions defined by the C standard.

## Docstring of Built-in functions

Following code displays the docstring of abs() function and randint() function in random module.

&lt;/&gt;

Open Compiler

```
print ("Docstring of built-in abs() function:", abs.__doc__)
import random

print ("Docstring of random.randint() function:",
random.randint.__doc__)
```

It will produce the following output –

Docstring of built-in abs() function: Return the absolute value of the argument.

Docstring of random.randint() function: Return random integer in range [a, b], including both end points.

## Docstring of built-in class

Docstrings of built-in classes are usually more explanatory, hence the text is over multiple lines. Below, we check the docstring of built-in dict class

&lt;/&gt;

Open Compiler

```
print ("Docstring of built-in dict class:", dict.__doc__)
```

It will produce the following **output** –

Docstring of built-in dict class: dict() -> new empty dictionary  
dict(mapping) -> new dictionary initialized from a mapping object's  
(key, value) pairs  
dict(iterable) -> new dictionary initialized as if via:  
d = {}  
for k, v in iterable:

```
d[k] = v  
dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword ar
```

## Docstring of Template class

Template class is defined in string module of Python's standard library. Its docstring is as follows –

&lt;/&gt;

Open Compiler

```
from string import Template  
  
print ("Docstring of Template class:", Template.__doc__)
```

It will produce the following **output** –

Docstring of Template class: A string class for supporting \$- substitutions.

## Docstring in help system

The docstring is also used by Python's built-in help service. For example check its help of abs() function in Python interpreter –

```
>>> help (abs)  
Help on built-in function abs in module builtins:  
abs(x, /)  
    Return the absolute value of the argument.
```

Similarly, define a function in the interpreter terminal and run help command.

```
>>> def addition(x,y):  
...     '''addtion(x,y)  
...     Returns the sum of x and y  
...     '''  
...     return x+y  
...  
>>> help (addition)  
Help on function addition in module __main__:  
addition(x, y)
```

```
addtion(x,y)
Returns the sum of x and y
```

Docstring also is used by IDEs to provide useful type ahead information while editing the code.

```
example.py > ...
1
2 def addtion(x,y):
3     ...
4     addition(x, y)
5     addition(x,y)
6     Returns the sum of x and y
7     ...
8     return x+y
9 addition
```

def addtion(  
 x: Any,  
 y: Any  
 ) -> Any

addition(x, y) addition(x,y) Returns the sum of x and y

## Docstring as Comment

A string literal appearing anywhere other than these objects (function, method, class, module or package) is ignored by the interpreter, hence they are similar to comments (which start with # symbol).

&lt;/&gt;

Open Compiler

```
# This is a comment
print ("Hello World")
'''This is also a comment'''
print ("How are you?")
```

## Python - JSON

JSON stands for JavaScript Object Notation. It is a lightweight data interchange format. It is similar to pickle. However, pickle serialization is Python specific whereas JSON format is implemented by many languages. The json module in Python's standard library implements object serialization functionality that is similar to pickle and marshal modules.

Just as in pickle module, the json module also provides dumps() and loads() function for serialization of Python object into JSON encoded string, and dump() and load() functions write and read serialized Python objects to/from file.

- **dumps()** – This function converts the object into JSON format.

- **loads()** – This function converts a JSON string back to Python object.

The following example demonstrates basic usage of these functions –

## Example 1

```
</> Open Compiler

import json

data=[ 'Rakesh', {'marks':(50,60,70)}]
s=json.dumps(data)
print (s, type(s))

data = json.loads(s)
print (data, type(data))
```

It will produce the following **output** –

```
["Rakesh", {"marks": [50, 60, 70]}] <class 'str'>
['Rakesh', {'marks': [50, 60, 70]}] <class 'list'>
```

The dumps() function can take optional sort\_keys argument. By default it is False. If set to True, the dictionary keys appear in sorted order in the JSON string.

```
data=[ 'Rakesh', {'marks':(50,60,70)}]
s=json.dumps(data, sort_keys=True)
```

## Example 2

The dumps() function has another optional parameter called **indent** which takes a number as value. It decides length of each segment of formatted representation of json string, similar to pprint output.

```
</> Open Compiler

import json
data=[ 'Rakesh', {'marks':(50,60,70)}]
s=json.dumps(data, indent = 2)
print (s)
```

It will produce the following output –

```
[  
    "Rakesh",  
    {  
        "marks": [  
            50,  
            60,  
            70  
        ]  
    }  
]
```

The json module also has object-oriented API corresponding to above functions. There are two classes defined in the module – JSONEncoder and JSONDecoder.

## JSONEncoder Class

Object of this class is encoder for Python data structures. Each Python data type is converted in corresponding JSON type as shown in following table –

| Python                                 | JSON   |
|----------------------------------------|--------|
| Dict                                   | object |
| list, tuple                            | array  |
| Str                                    | string |
| int, float, int- & float-derived Enums | number |
| True                                   | true   |
| False                                  | false  |
| None                                   | null   |

The JSONEncoder class is instantiated by JSONEncoder() constructor. Following important methods are defined in encoder class –

- **encode()** – serializes Python object into JSON format.
- **iterencode()** – Encodes the object and returns an iterator yielding encoded form of each item in the object.
- **indent** – Determines indent level of encoded string.

- **sort\_keys** – is either true or false to make keys appear in sorted order or not.
- **check\_circular** – if True, check for circular reference in container type object.

The following example encodes Python list object.

## Example

```
import json

data=['Rakesh',{'marks':(50,60,70)}]
e=json.JSONEncoder()
```

Using iterencode() method, each part of the encoded string is displayed as below –

</>

Open Compiler

```
import json
data=['Rakesh',{'marks':(50,60,70)}]
e=json.JSONEncoder()
for obj in e.iterencode(data):
    print (obj)
```

It will produce the following output –

```
["Rakesh"
',
{
"marks"
:
[50
, 60
, 70
]
}
]
```

## JSONDEcoder class

Object of this class helps in decoded in json string back to Python data structure. Main method in this class is decode(). Following example code retrieves Python list object from

encoded string in earlier step.

## Example

```
</> Open Compiler  
  
import json  
data=['Rakesh',{'marks':(50,60,70)}]  
e=json.JSONEncoder()  
s = e.encode(data)  
d=json.JSONDecoder()  
obj = d.decode(s)  
print (obj, type(obj))
```

It will produce the following **output** –

```
['Rakesh', {'marks': [50, 60, 70]}] <class 'list'>
```

## JSON with Files/Streams

The json module defines load() and dump() functions to write JSON data to a file like object – which may be a disk file or a byte stream and read data back from them.

### dump() Function

This function encodes Python object data in JSON format and writes it to a file. The file must be having write permission.

## Example

```
import json  
data=['Rakesh', {'marks': (50, 60, 70)}]  
fp=open('json.txt','w')  
json.dump(data,fp)  
fp.close()
```

This code will create 'json.txt' in current directory. It shows the contents as follows –

```
["Rakesh", {"marks": [50, 60, 70]}]
```

## load() Function

This function loads JSON data from the file and constructs Python object from it. The file must be opened with read permission.

### Example

```
import json
fp=open('json.txt','r')
ret=json.load(fp)
print (ret)
```

## Python - Sending Email

An application that handles and delivers e-mail over the Internet is called a "mail server". Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending an e-mail and routing e-mail between mail servers. It is an Internet standard for email transmission.

Python provides `smtplib` module, which defines an SMTP client session object that can be used to send mails to any Internet machine with an SMTP or ESMTP listener daemon.

### `smtplib.SMTP()` Function

To send an email, you need to obtain the object of `SMTP` class with the following function –

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

Here is the detail of the parameters –

- **host** – This is the host running your SMTP server. You can specify IP address of the host or a domain name like `tutorialspoint.com`. This is an optional argument.
- **port** – If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
- **local\_hostname** – If your SMTP server is running on your local machine, then you can specify just `localhost` as the option.

The `SMTP` object has following methods –

- **connect(host, port, source\_address)** – This method establishes connection to a host on a given port.
- **login(user, password)** – Log in on an SMTP server that requires authentication.
- **quit()** – terminate the SMTP session.
- **data(msg)** – sends message data to server.
- **docmd(cmd, args)** – send a command, and return its response code.
- **ehlo(name)** – Hostname to identify itself.
- **starttls()** – puts the connection to the SMTP server into TLS mode.
- **getreply()** – get a reply from the server consisting of server response code.
- **putcmd(cmd, args)** – sends a command to the server.
- **send\_message(msg, from\_addr, to\_addrs)** – converts message to a bytestring and passes it to sendmail.

## The smtpd Module

The **smtpd** module that comes pre-installed with Python has a local SMTP debugging server. You can test email functionality by starting it. It doesn't actually send emails to the specified address, it discards them and prints their content to the console. Running a local debugging server means it's not necessary to deal with encryption of messages or use credentials to log in to an email server.

You can start a local SMTP debugging server by typing the following in Command Prompt

```
python -m smtpd -c DebuggingServer -n localhost:1025
```

## Example

The following program sends a dummy email with the help of `smtplib` functionality.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
```

```

msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))
server = smtplib.SMTP('localhost', 1025)
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()

```

Basically we use the `sendmail()` method, specifying three parameters –

- **The sender** – A string with the address of the sender.
- **The receivers** – A list of strings, one for each recipient.
- **The message** – A message as a string formatted as specified in the various RFCs.

We have already started the SMTP debugging server. Run this program. User is asked to input the sender's ID, recipients and the message.

```

python example.py
From: abc@xyz.com
To: xyz@abc.com
Enter message, end with ^D (Unix) or ^Z (Windows):
Hello World
^Z

```

The console reflects the following log –

```

From: abc@xyz.com
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<xyz@abc.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'

```

```

send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'From: abc@xyz.com\r\nTo: xyz@abc.com\r\n\r\nHello
World\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'

```

The terminal in which the SMTPD server is running shows this **output** –

```

----- MESSAGE FOLLOWS -----
b'From: abc@xyz.com'
b'To: xyz@abc.com'
b'X-Peer: ::1'
b''
b'Hello World'
----- END MESSAGE -----

```

## Using gmail SMTP

Let us look at the script below which uses Google's **smtp** mail server to send an email message.

First of all SMTP object is set up using gmail's smtp server and port 527. The SMTP object then identifies itself by invoking ehlo() command. We also activate Transport Layer Security to the outgoing mail message.

Next the login() command is invoked by passing credentials as arguments to it. Finally the mail message is assembled by attaching it a header in prescribed format and it is sent using sendmail() method. The SMTP object is closed afterwards.

```

import smtplib
content="Hello World"
mail=smtplib.SMTP('smtp.gmail.com', 587)
mail.ehlo()
mail.starttls()
sender='mvl@gmail.com'

```

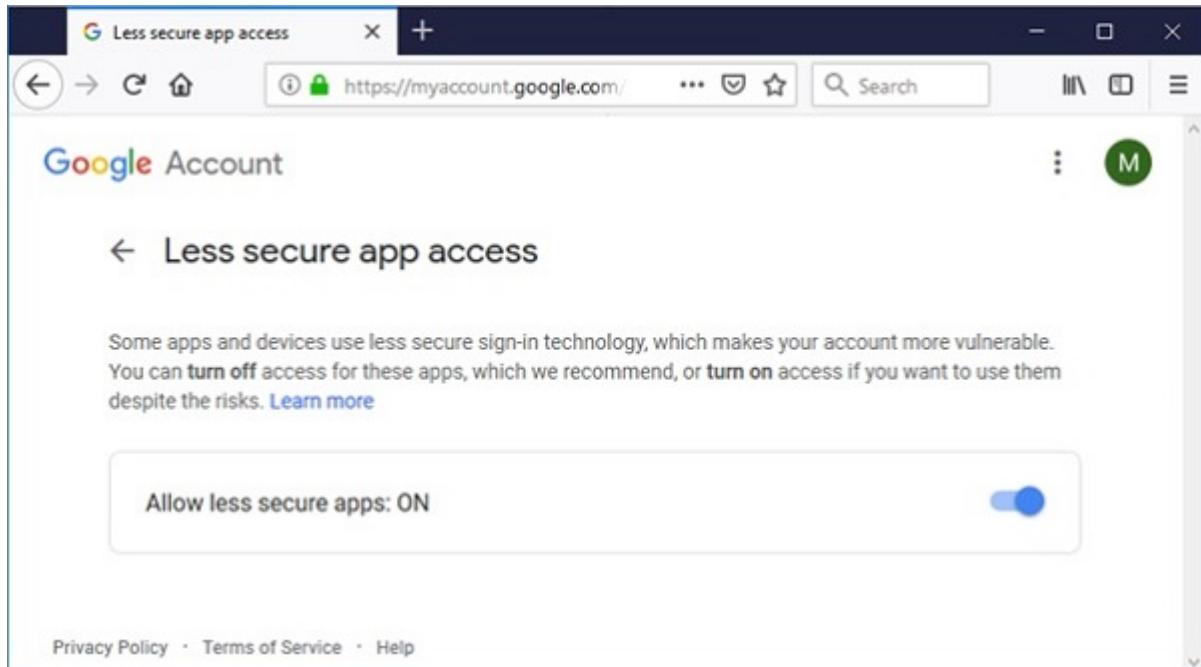
```

recipient='tester@gmail.com'
mail.login('mvl@gmail.com','*****')
header='To:' + recipient + '\n' + 'From:' \
+ sender + '\n' + 'subject:testmail\n'
content=header+content
mail.sendmail(sender, recipient, content)
mail.close()

```

Before running above script, sender's gmail account must be configured to allow 'less secure apps'. Visit following link.

<https://myaccount.google.com/lesssecureapps> Set the shown toggle button to ON.



If everything goes well, execute the above script. The message should be delivered to the recipient's inbox.

## Python - Further Extensions

Any code that you write using any compiled language like C, C++, or Java can be integrated or imported into another Python script. This code is considered as an "extension."

A Python extension module is nothing more than a normal C library. On Unix machines, these libraries usually end in **.so** (for shared object). On Windows machines, you typically see **.dll** (for dynamically linked library).

### Pre-Requisites for Writing Extensions

To start writing your extension, you are going to need the Python header files.

- On Unix machines, this usually requires installing a developer-specific package.
- Windows users get these headers as part of the package when they use the binary Python installer.

Additionally, it is assumed that you have a good knowledge of C or C++ to write any Python Extension using C programming.

## First look at a Python Extension

For your first look at a Python extension module, you need to group your code into four parts –

- The header file Python.h.
- The C functions you want to expose as the interface from your module..
- A table mapping the names of your functions as Python developers see them as C functions inside the extension module..
- An initialization function.

## The Header File Python.h

You need to include Python.h header file in your C source file, which gives you the access to the internal Python API used to hook your module into the interpreter.

Make sure to include Python.h before any other headers you might need. You need to follow the includes with the functions you want to call from Python.

## The C Functions

The signatures of the C implementation of your functions always takes one of the following three forms –

```
static PyObject *MyFunction(PyObject *self, PyObject *args);
static PyObject *MyFunctionWithKeywords(PyObject *self,
                                       PyObject *args,
                                       PyObject *kw);
static PyObject *MyFunctionWithNoArgs(PyObject *self);
```

Each one of the preceding declarations returns a Python object. There is no such thing as a void function in Python as there is in C. If you do not want your functions to return a

value, return the C equivalent of Python's **None** value. The Python headers define a macro, `Py_RETURN_NONE`, that does this for us.

The names of your C functions can be whatever you like as they are never seen outside of the extension module. They are defined as static function.

Your C functions usually are named by combining the Python module and function names together, as shown here –

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Do your stuff here. */
    Py_RETURN_NONE;
}
```

This is a Python function called `func` inside the `module` module. You will be putting pointers to your C functions into the method table for the module that usually comes next in your source code.

## The Method Mapping Table

This method table is a simple array of `PyMethodDef` structures. That structure looks something like this –

```
struct PyMethodDef {
    char *ml_name;
    PyCFunction ml_meth;
    int ml_flags;
    char *ml_doc;
};
```

Here is the description of the members of this structure –

- **ml\_name** – This is the name of the function as the Python interpreter presents when it is used in Python programs.
- **ml\_meth** – This is the address of a function that has any one of the signatures, described in the previous section.
- **ml\_flags** – This tells the interpreter which of the three signatures `ml_meth` is using.
  - This flag usually has a value of `METH_VARARGS`.
  - This flag can be bitwise OR'ed with `METH_KEYWORDS` if you want to allow keyword arguments into your function.

- This can also have a value of METH\_NOARGS that indicates you do not want to accept any arguments.
- **mml\_doc** – This is the docstring for the function, which could be NULL if you do not feel like writing one.

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members.

## Example

For the above-defined function, we have the following method mapping table –

```
static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```

## The Initialization Function

The last part of your extension module is the initialization function. This function is called by the Python interpreter when the module is loaded. It is required that the function be named **initModule**, where Module is the name of the module.

The initialization function needs to be exported from the library you will be building. The Python headers define PyMODINIT\_FUNC to include the appropriate incantations for that to happen for the particular environment in which we are compiling. All you have to do is use it when defining the function.

Your C initialization function generally has the following overall structure –

```
PyMODINIT_FUNC initModule() {
    Py_Initialize3(func, module_methods, "docstring...");
}
```

Here is the description of Py\_Initialize3 function –

- **func** – This is the function to be exported.
- **module\_methods** – This is the mapping table name defined above.
- **docstring** – This is the comment you want to give in your extension.

Putting all this together, it looks like the following –

```
#include <Python.h>

static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Do your stuff here. */
    Py_RETURN_NONE;
}

static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initModule() {
    Py_InitModule3(func, module_methods, "docstring...");
}
```

## Example

A simple example that makes use of all the above concepts –

```
#include <Python.h>

static PyObject* helloworld(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions!!");
}

static char helloworld_docs[] =
    "helloworld( ): Any message you want to put here!!\n";
static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld,
     METH_NOARGS, helloworld_docs},
    {NULL}
};
void initelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                  "Extension module example!");
}
```

Here the Py\_BuildValue function is used to build a Python value. Save above code in hello.c file. We would see how to compile and install this module to be called from Python script.

## Building and Installing Extensions

The distutils package makes it very easy to distribute Python modules, both pure Python and extension modules, in a standard way. Modules are distributed in the source form, built and installed via a setup script usually called setup.py.

For the above module, you need to prepare the following setup.py script –

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0', \
      ext_modules=[Extension('helloworld', ['hello.c'])])
```

Now, use the following command, which would perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into an appropriate directory –

```
$ python setup.py install
```

On Unix-based systems, you will most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually is not a problem on Windows.

## Importing Extensions

Once you install your extensions, you would be able to import and call that extension in your Python script as follows –

```
import helloworld
print helloworld.helloworld()
```

This would produce the following **output** –

```
Hello, Python extensions!!
```

## Passing Function Parameters

As you will most likely want to define functions that accept arguments, you can use one of the other signatures for your C functions. For example, the following function, that accepts some number of parameters, would be defined like this –

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Parse args and do something interesting here. */
    Py_RETURN_NONE;
}
```

The method table containing an entry for the new function would look like this –

```
static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { "func", module_func, METH_VARARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```

You can use the API `PyArg_ParseTuple` function to extract the arguments from the one `PyObject` pointer passed into your C function.

The first argument to `PyArg_ParseTuple` is the `args` argument. This is the object you will be parsing. The second argument is a format string describing the arguments as you expect them to appear. Each argument is represented by one or more characters in the format string as follows.

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    int i;
    double d;
    char *s;
    if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
        return NULL;
    }

    /* Do something interesting here. */
    Py_RETURN_NONE;
}
```

Compiling the new version of your module and importing it enables you to invoke the new function with any number of arguments of any type –

```
module.func(1, s="three", d=2.0)
module.func(i=1, d=2.0, s="three")
module.func(s="three", d=2.0, i=1)
```

You can probably come up with even more variations.

## The `PyArg_ParseTuple` Function

re is the standard signature for the **`PyArg_ParseTuple`** function –

```
int PyArg_ParseTuple(PyObject* tuple, char* format, ...)
```

This function returns 0 for errors, and a value not equal to 0 for success. Tuple is the PyObject\* that was the C function's second argument. Here format is a C string that describes mandatory and optional arguments.

Here is a list of format codes for the **PyArg\_ParseTuple** function –

| Code  | C type          | Meaning                                                   |
|-------|-----------------|-----------------------------------------------------------|
| c     | char            | A Python string of length 1 becomes a C char.             |
| d     | double          | A Python float becomes a C double.                        |
| f     | float           | A Python float becomes a C float.                         |
| i     | int             | A Python int becomes a C int.                             |
| l     | long            | A Python int becomes a C long.                            |
| L     | long long       | A Python int becomes a C long long.                       |
| O     | PyObject*       | Gets non-NULL borrowed reference to Python argument.      |
| S     | char*           | Python string without embedded nulls to C char*.          |
| s#    | char*+int       | Any Python string to C address and length.                |
| t#    | char*+int       | Read-only single-segment buffer to C address and length.  |
| u     | Py_UNICODE*     | Python Unicode without embedded nulls to C.               |
| u#    | Py_UNICODE*+int | Any Python Unicode C address and length.                  |
| w#    | char*+int       | Read/write single-segment buffer to C address and length. |
| z     | char*           | Like s, also accepts None (sets C char* to NULL).         |
| z#    | char*+int       | Like s#, also accepts None (sets C char* to NULL).        |
| (...) | as per ...      | A Python sequence is treated as one argument per item.    |
|       |                 | The following arguments are optional.                     |
| :     |                 | Format end, followed by function name for error messages. |
| ;     |                 | Format end, followed by entire error message text.        |

## Returning Values

**Py\_BuildValue** takes in a format string much like **PyArg\_ParseTuple** does. Instead of passing in the addresses of the values you are building, you pass in the actual values. Here is an example showing how to implement an add function.

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("i", a + b);
}
```

This is what it would look like if implemented in Python –

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

You can return two values from your function as follows. This would be captured using a list in Python.

```
static PyObject *foo_add_subtract(PyObject *self, PyObject *args) {
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("ii", a + b, a - b);
}
```

This is what it would look like if implemented in Python –

```
def add_subtract(a, b):
    return (a + b, a - b)
```

## The **Py\_BuildValue** Function

Here is the standard signature for **Py\_BuildValue** function –

```
PyObject* Py_BuildValue(char* format,...)
```

Here format is a C string that describes the Python object to build. The following arguments of Py\_BuildValue are C values from which the result is built. The PyObject\* result is a new reference.

The following table lists the commonly used code strings, of which zero or more are joined into a string format.

| Code  | C type          | Meaning                                                              |
|-------|-----------------|----------------------------------------------------------------------|
| c     | char            | A C char becomes a Python string of length 1.                        |
| d     | double          | A C double becomes a Python float.                                   |
| f     | float           | A C float becomes a Python float.                                    |
| i     | int             | C int becomes a Python int                                           |
| l     | long            | A C long becomes a Python int                                        |
| N     | PyObject*       | Passes a Python object and steals a reference.                       |
| O     | PyObject*       | Passes a Python object and INCREFs it as normal.                     |
| O&    | convert+void*   | Arbitrary conversion                                                 |
| s     | char*           | C 0-terminated char* to Python string, or NULL to None.              |
| s#    | char*+int       | C char* and length to Python string, or NULL to None.                |
| u     | Py_UNICODE*     | C-wide, null-terminated string to Python Unicode, or NULL to None.   |
| u#    | Py_UNICODE*+int | C-wide string and length to Python Unicode, or NULL to None.         |
| w#    | char*+int       | Read/write single-segment buffer to C address and length.            |
| z     | char*           | Like s, also accepts None (sets C char* to NULL).                    |
| z#    | char*+int       | Like s#, also accepts None (sets C char* to NULL).                   |
| (...) | as per ...      | Builds Python tuple from C values.                                   |
| [...] | as per ...      | Builds Python list from C values.                                    |
| {...} | as per ...      | Builds Python dictionary from C values, alternately keys and values. |

Code {...} builds dictionaries from an even number of C values, alternately keys and values. For example, Py\_BuildValue("{issi}",23,"zig","zag",42) returns a dictionary like

```
Python's {23:'zig','zag':42}
```

# Python - Tools/Utilities

The standard library comes with a number of modules that can be used both as modules and as command-line utilities.

## The dis Module

The **dis** module is the Python disassembler. It converts byte codes to a format that is slightly more appropriate for human consumption.

### Example

```
import dis

def sum():
    vara = 10
    varb = 20

    sum = vara + varb
    print ("vara + varb = %d" % sum)

# Call dis function for the function.
dis.dis(sum)
```

This would produce the following result –

|   |                  |                        |
|---|------------------|------------------------|
| 3 | 0 LOAD_CONST     | 1 (10)                 |
|   | 2 STORE_FAST     | 0 (vara)               |
| 4 | 4 LOAD_CONST     | 2 (20)                 |
|   | 6 STORE_FAST     | 1 (varb)               |
| 6 | 8 LOAD_FAST      | 0 (vara)               |
|   | 10 LOAD_FAST     | 1 (varb)               |
|   | 12 BINARY_ADD    |                        |
|   | 14 STORE_FAST    | 2 (sum)                |
| 7 | 16 LOAD_GLOBAL   | 0 (print)              |
|   | 18 LOAD_CONST    | 3 ('vara + varb = %d') |
|   | 20 LOAD_FAST     | 2 (sum)                |
|   | 22 BINARY_MODULO |                        |

```
24 CALL_FUNCTION      1
26 POP_TOP
28 LOAD_CONST         0 (None)
30 RETURN_VALUE
```

## The pdb Module

The pdb module is the standard Python debugger. It is based on the bdb debugger framework.

You can run the debugger from the command line (type n [or next] to go to the next line and help to get a list of available commands) –

### Example

Before you try to run **pdb.py**, set your path properly to Python lib directory. So let us try with above example sum.py –

```
$pdb.py sum.py
> /test/sum.py(3)<module>()
-> import dis
(Pdb) n
> /test/sum.py(5)<module>()
-> def sum():
(Pdb) n
>/test/sum.py(14)<module>()
-> dis.dis(sum)
(Pdb) n
   6           0 LOAD_CONST      1 (10)
   3           3 STORE_FAST       0 (vara)

    7           6 LOAD_CONST      2 (20)
   9           9 STORE_FAST       1 (varb)

   9          12 LOAD_FAST        0 (vara)
  15          15 LOAD_FAST        1 (varb)
  18          18 BINARY_ADD
  19          19 STORE_FAST       2 (sum)

  10          22 LOAD_CONST      3 ('vara + varb = %d')
  25          25 LOAD_FAST        2 (sum)
  28          28 BINARY_MODULO
```

```

29 PRINT_ITEM
30 PRINT_NEWLINE
31 LOAD_CONST          0 (None)
34 RETURN_VALUE

--Return--
> /test/sum.py(14)<module>()->None
-v dis.dis(sum)
(Pdb) n
--Return--
> <string>(1)<module>()->None
(Pdb)

```

## The profile Module

The profile module is the standard Python profiler. You can run the profiler from the command line –

### Example

Let us try to profile the following program –

```

vara = 10
varb = 20
sum = vara + varb
print "vara + varb = %d" % sum

```

Now, try running **cProfile.py** over this file sum.py as follow –

```

$cProfile.py sum.py
vara + varb = 30
  4 function calls in 0.000 CPU seconds

  Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno
    1      0.000    0.000    0.000    0.000 <string>:1(<module>)
    1      0.000    0.000    0.000    0.000 sum.py:3(<module>)
    1      0.000    0.000    0.000    0.000 {execfile}
    1      0.000    0.000    0.000    0.000 {method ....}

```

## The tabnanny Module

The tabnanny module checks Python source files for ambiguous indentation. If a file mixes tabs and spaces in a way that throws off indentation, no matter what tab size you're

using, the nanny complains.

## Example

Let us try to profile the following program –

```
vara = 10
varb = 20

sum = vara + varb
print "vara + varb = %d" % sum
```

If you would try a correct file with tabnanny.py, then it won't complain as follows –

```
$tabnanny.py -v sum.py
'sum.py': Clean bill of health.
```

# Python - GUIs

In this chapter, you will learn about some popular Python IDEs (**Integrated Development Environment**), and how to use IDE for program development.

To use the scripted mode of Python, you need to save the sequence of Python instructions in a text file and save it with **.py** extension. You can use any text editor available on the operating system. Whenever the interpreter encounters errors, the source code needs to be edited and run again and again. To avoid this tedious method, IDE is used. An IDE is a one stop solution for typing, editing the source code, detecting the errors and executing the program.

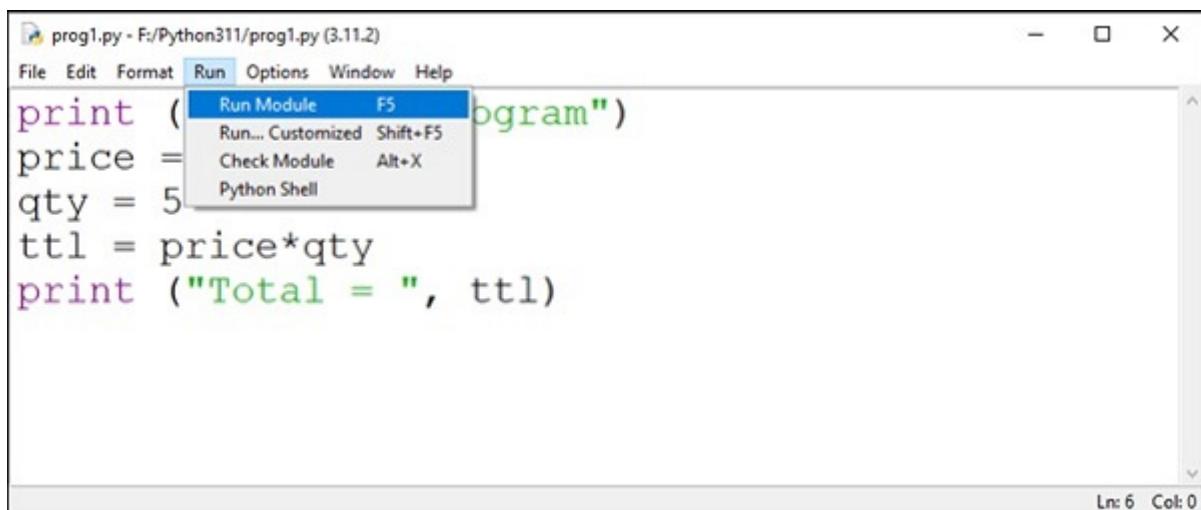
## IDLE

Python's standard library contains the **IDLE** module. IDLE stands for **Integrated Development and Learning Environment**. As the name suggests, it is useful when one is in the learning stage. It includes a Python interactive shell and a code editor, customized to the needs of Python language structure. Some of its important features include syntax highlighting, auto-completion, customizable interface etc.

To write a Python script, open a new text editor window from the File menu.



A new editor window opens in which you can enter the Python code. Save it and run it with Run menu.



## Jupyter Notebook

Initially developed as a web interface for IPython, Jupyter Notebook supports multiple languages. The name itself derives from the alphabets from the names of the supported languages – **J**ulia, **P**YTHON and **R**. Jupyter notebook is a client server application. The server is launched at the localhost, and the browser acts as its client.

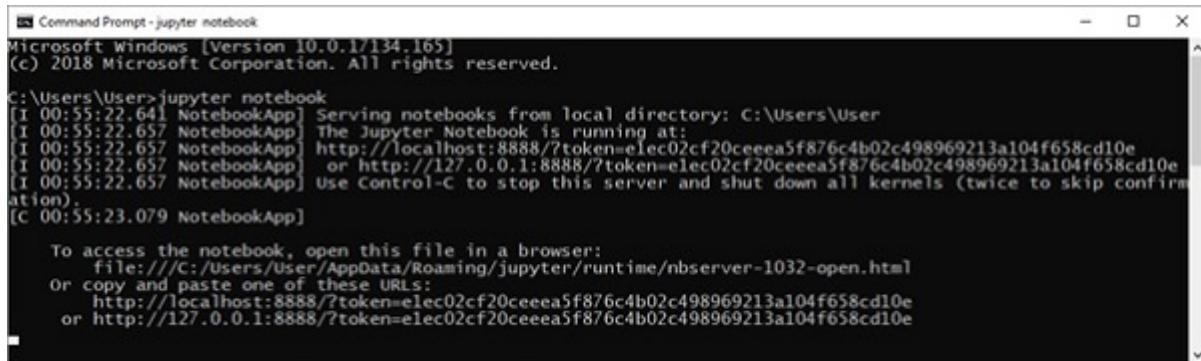
Install Jupyter notebook with PIP –

```
pip3 install jupyter
```

Invoke from the command line.

```
C:\Users\Acer>jupyter notebook
```

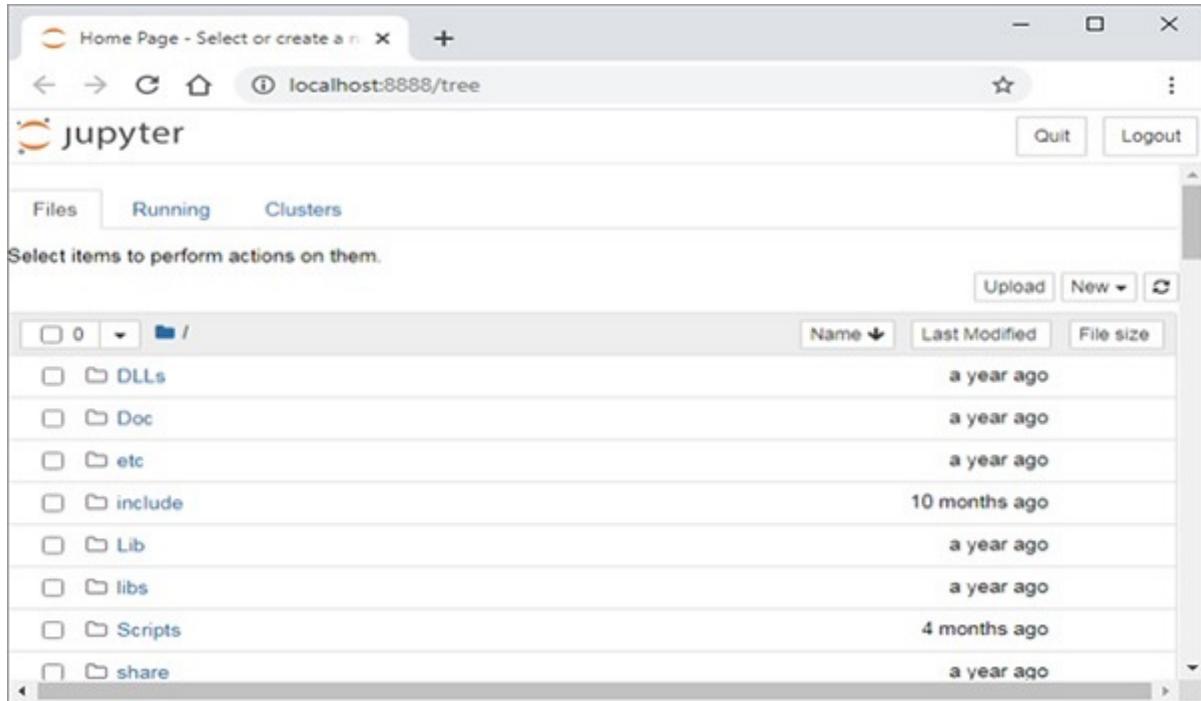
The server is launched at localhost's 8888 port number.



```
[I 00:55:22.641 NotebookApp] Serving notebooks from local directory: C:\Users\User
[I 00:55:22.657 NotebookApp] The Jupyter Notebook is running at:
[I 00:55:22.657 NotebookApp] http://localhost:8888/?token=elec02cf20ceeee5f876c4b02c498969213a104f658cd10e
[I 00:55:22.657 NotebookApp] or http://127.0.0.1:8888/?token=elec02cf20ceeee5f876c4b02c498969213a104f658cd10e
[I 00:55:22.657 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 00:55:23.079 NotebookApp]

To access the notebook, open this file in a browser:
  file:///C:/Users/User/AppData/Roaming/jupyter/runtime/nbserver-1032-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=elec02cf20ceeee5f876c4b02c498969213a104f658cd10e
  or http://127.0.0.1:8888/?token=elec02cf20ceeee5f876c4b02c498969213a104f658cd10e
```

The default browser of your system opens a link **http://localhost:8888/tree** to display the dashboard.



Open a new Python notebook. It shows IPython style input cell. Enter Python instructions and run the cell.

The screenshot shows a Jupyter Notebook window titled "Untitled". The top bar includes tabs for "Home Page - Select or create a notebook" and "Untitled - Jupyter Notebook". The toolbar has icons for file operations like New, Open, Save, and Run, along with a Python 3 kernel selector. The main area displays a code cell labeled "In [1]:" containing Python code to print the sum of two variables. The output below the cell shows the result: "a = 10 b = 20 c = 30". A new cell "In [ ]:" is visible at the bottom.

```
In [1]: a=10  
b=20  
c=a+b  
print ("a = {} b = {} c = {}".format(a,b,c))  
a = 10 b = 20 c = 30
```

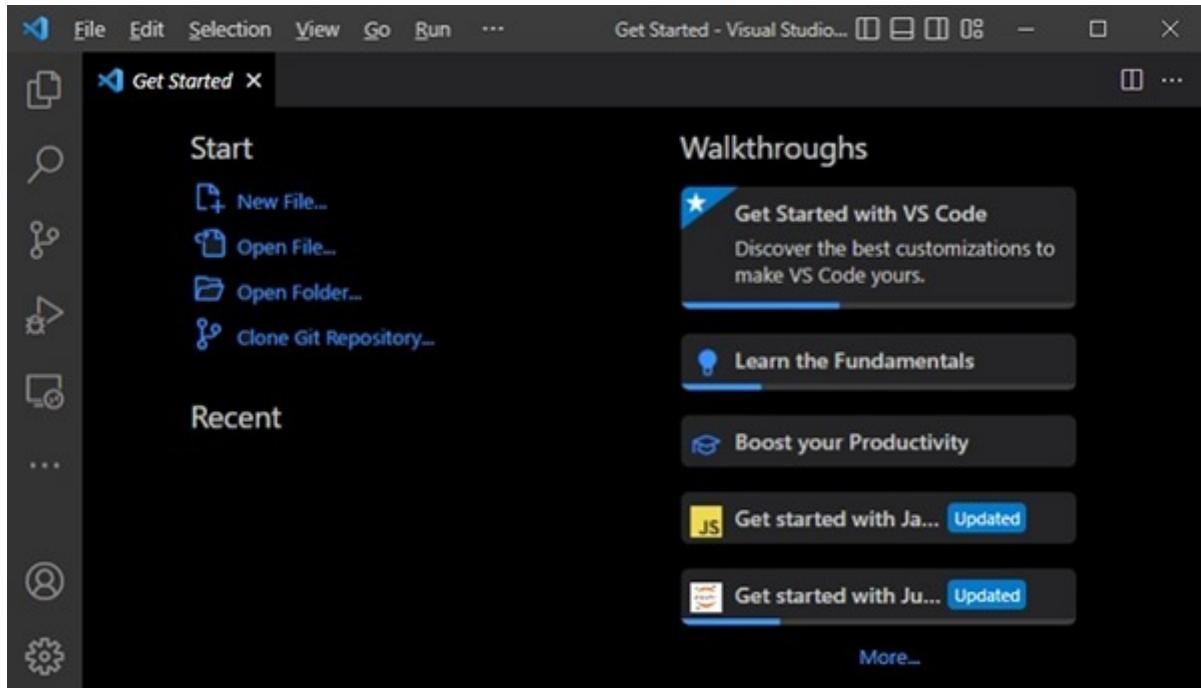
Jupyter notebook is a versatile tool, used very extensively by data scientists to display inline data visualizations. The notebook can be conveniently converted and distributed in PDF, HTML or Markdown format.

## VS Code

Microsoft has developed a source code editor called VS Code (**Visual Studio Code**) that supports multiple languages including C++, Java, Python and others. It provides features such as syntax highlighting, autocomplete, debugger and version control.

VS Code is a freeware. It is available for download and install from <https://code.visualstudio.com/>.

Launch VS Code from the start menu (in Windows).



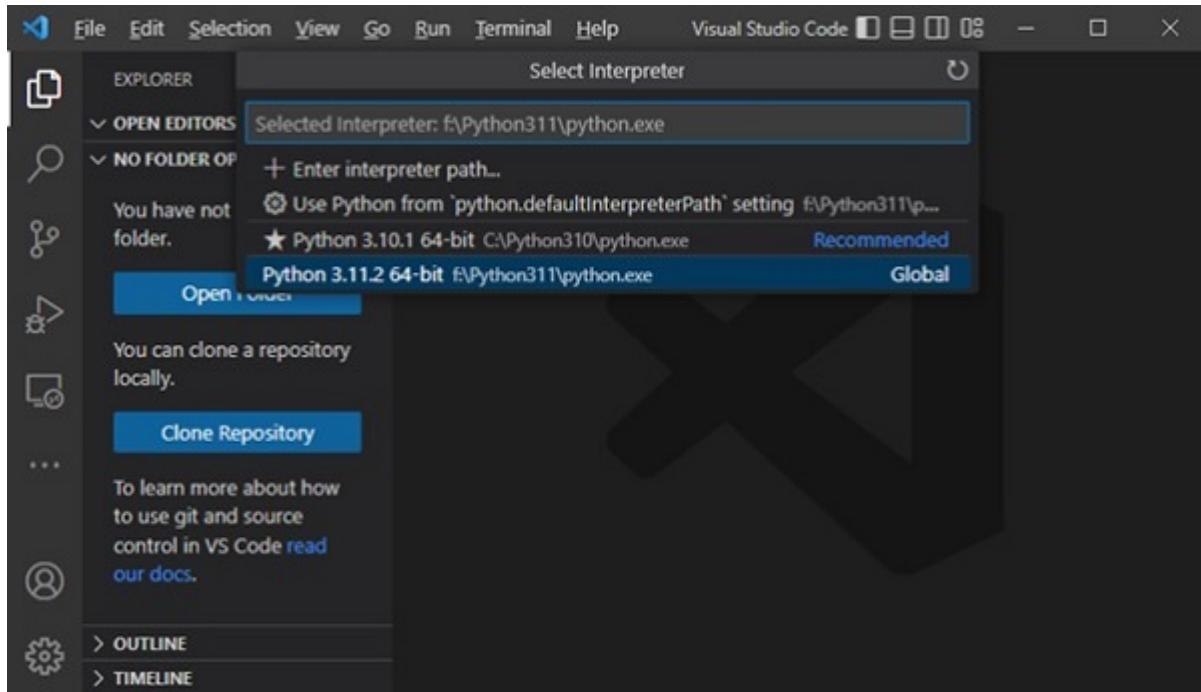
You can also launch VS Code from command line –

```
C:\test>code .
```

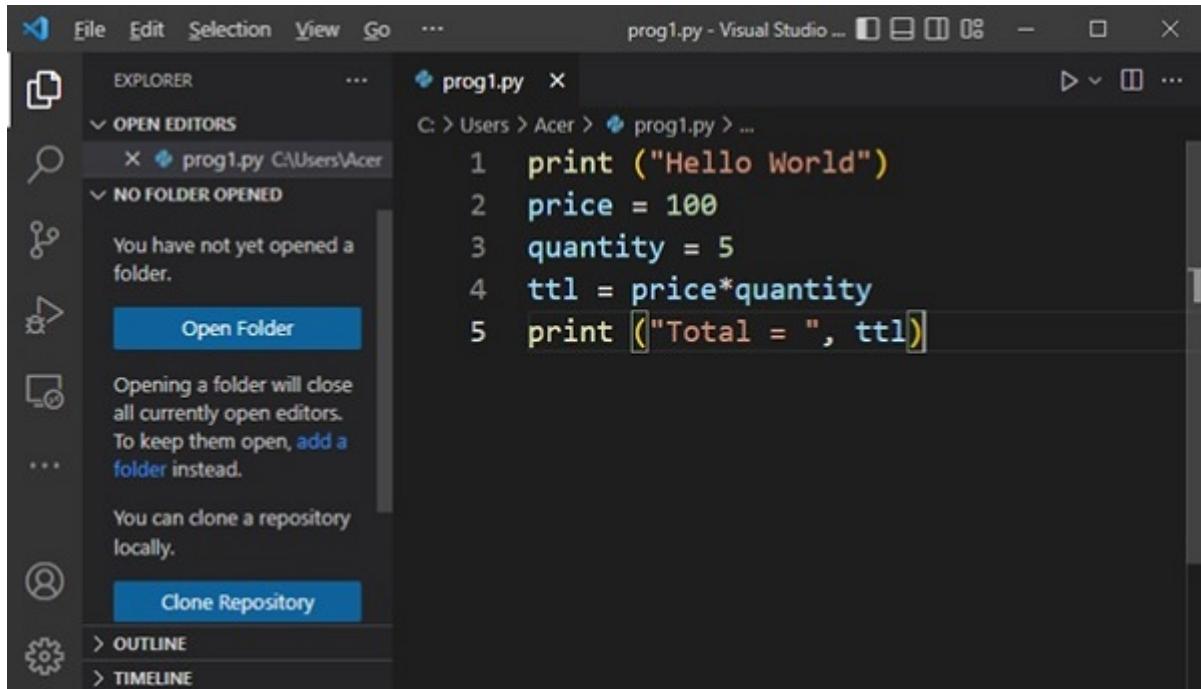
VS Code cannot be used unless respective language extension is not installed. VS Code **Extensions** marketplace has a number of extensions for language compilers and other utilities. Search for Python extension from the Extension tab (Ctrl+Shift+X) and install it.



After activating Python extension, you need to set the Python interpreter. Press Ctrl+Shift+P and select Python interpreter.



Open a new text file, enter Python code and save the file.



Open a command prompt terminal and run the program.

The screenshot shows the PyCharm IDE interface. On the left is the Explorer sidebar with options like 'Open Editors' (containing 'prog1.py'), 'No Folder Opened', 'Open Folder', and 'Clone Repository'. The main area shows a code editor with the following Python script:

```
1 print ("Hello World")
2 price = 100
3 quantity = 5
4 ttl = price*quantity
5 print ("Total = ", ttl)
```

Below the code editor is a terminal window displaying the execution of the script:

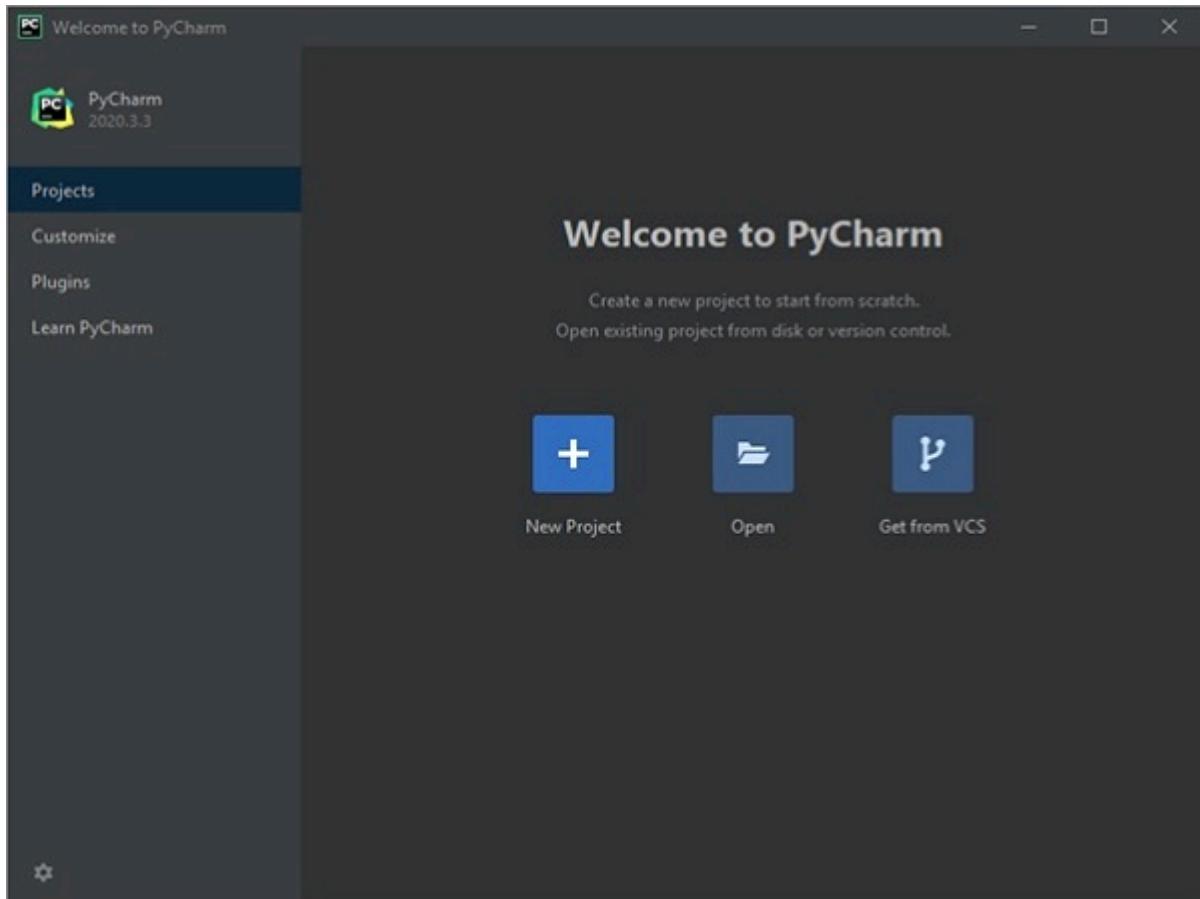
```
C:\Users\Acer>
C:\Users\Acer>python prog1.py
Hello World
Total =  500
C:\Users\Acer>
```

## PyCharm

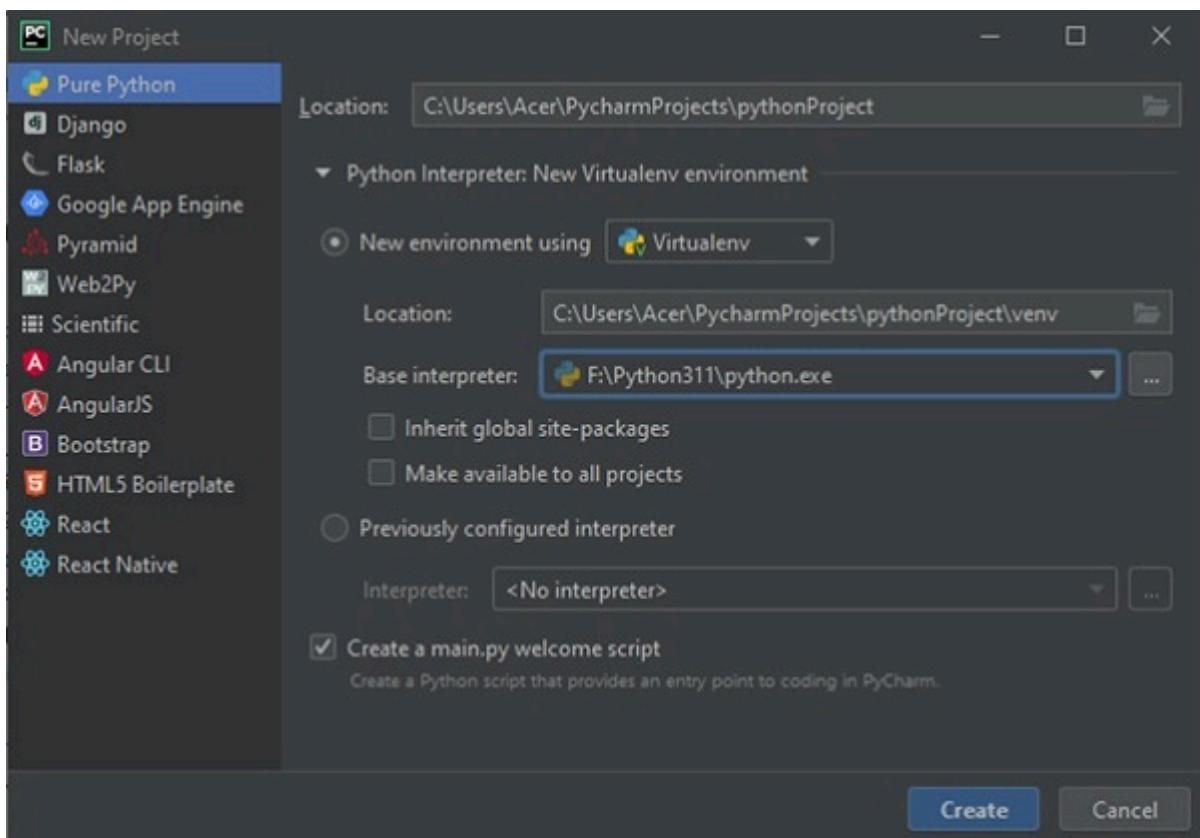
PyCharm is another popular Python IDE. It has been developed by JetBrains, a Czech software company. Its features include code analysis, a graphical debugger, integration with version control systems etc. PyCharm supports web development with Django.

The community as well as professional editions can be downloaded from <https://www.jetbrains.com/pycharm/download/>.

Download, install the latest Version: 2022.3.2 and open PyCharm. The Welcome screen appears as below –



When you start a new project, PyCharm creates a virtual environment for it based on the choice of folder location and the version of Python interpreter chosen.



You can now add one or more Python scripts required for the project. Here we add a sample Python code in `main.py` file.

```

# Press Shift+F10 to execute it or replace it with your code. Indexing...
# Press Double Shift to search everywhere for classes, files, tool windows,
# libraries and snippets.

print ("My first program")
price = 100
qty = 5
ttl = price*qty
print ("Total = ", ttl)

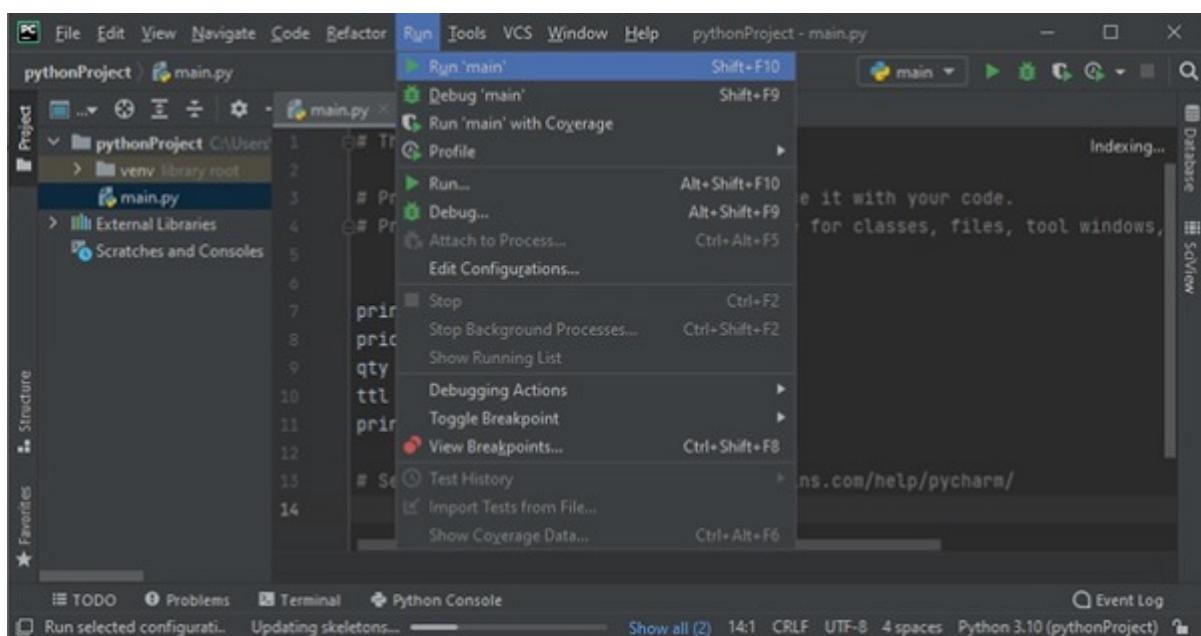
# See PyCharm help at https://www.jetbrains.com/help/pycharm/

```

Run    TODO    Problems    Terminal    Python Console    Event Log

PyCharm 2020.3.5 available // U... (10 minutes ago)    Updating skeletons...    Show all (2)    14:1    Python 3.10 (pythonProject)    ?

To execute the program, choose from Run menu or use Shift+F10 shortcut.



Output will be displayed in the console window as shown below –

```
print ("Total = ", ttl)
# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

Run: main

My first program  
Total = 500

Process finished with exit code 0