

# Python Tutorial

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

Python is a simple, easy to learn, powerful, high level and object-oriented programming language.

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

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

## Python Introduction

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

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

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

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

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

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

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

## Python Features

Python provides lots of features that are listed below.

### **1) Easy to Learn and Use**

Python is easy to learn and use. It is developer-friendly and high level programming language.

---

### **2) Expressive Language**

Python language is more expressive means that it is more understandable and readable.

---

### **3) Interpreted Language**

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

---

### **4) Cross-platform Language**

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

---

## **5) Free and Open Source**

Python language is freely available at [official web address](#). The source-code is also available. Therefore it is open source.

---

## **6) Object-Oriented Language**

Python supports object oriented language and concepts of classes and objects come into existence.

---

## **7) Extensible**

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

---

## **8) Large Standard Library**

Python has a large and broad library and provides rich set of module and functions for rapid application development.

---

## **9) GUI Programming Support**

Graphical user interfaces can be developed using Python.

---

## **10) Integrated**

It can be easily integrated with languages like C, C++, JAVA etc.

# Python History and Versions

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

## Python Version List

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

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

Python Version	Released Date
1.0	December 3, 1994
1.1	February 1995
1.2	September 1995
1.3	January 1996
1.4	July 1996
1.5	January 1997
1.6	July 1997
1.7	January 1998
1.8	July 1998
1.9	January 1999
2.0	December 1999
2.1	February 2000
2.2	February 2001
2.3	February 2002
2.4	February 2003
2.5	February 2004
2.6	February 2005
2.7	February 2008
2.8	December 2009
2.9	December 2010
3.0	December 2008
3.1	December 2009
3.2	December 2010
3.3	December 2011
3.4	December 2012
3.5	December 2013
3.6	December 2014
3.7	December 2015
3.8	December 2016
3.9	December 2017
3.10	December 2018
3.11	December 2019
3.12	December 2020
3.13	December 2021
3.14	December 2022
3.15	December 2023

Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010

Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018

## Python Applications

Python is known for its general purpose nature that makes it applicable in almost each domain of software development. Python as a whole can be used in any sphere of development.

Here, we are specifying applications areas where python can be applied.

## **1) Web Applications**

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser etc. It also provides Frameworks such as Django, Pyramid, Flask etc to design and develop web based applications. Some important developments are: PythonWikiEngines, Pocoo, PythonBlogSoftware etc.

## **2) Desktop GUI Applications**

Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

## **3) Software Development**

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

## **4) Scientific and Numeric**

Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

## **5) Business Applications**

Python is used to build Business applications like ERP and e-commerce systems. Tryton is a high level application platform.

## **6) Console Based Application**

We can use Python to develop console based applications. For example: **IPython**.

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

Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

## **8) 3D CAD Applications**

To create CAD

application Fandango is a real application which provides full features of CAD.

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

In this section of the tutorial, we will discuss the installation of python on various operating systems.

## **Installation on Windows**

Visit the link <https://www.python.org/downloads/> to download the latest release of Python. In this process, we will install Python 3.6.7 on our Windows operating system.



Secure | <https://www.python.org/downloads/>



JavaFX Tutorial -



Hindi News | San



Flowchart Maker

Help the PSF raise \$30,000 USD by November 21st!

Participate in our Recurring Giving Campaign

## Looking for a specific release?

Python releases by version number:

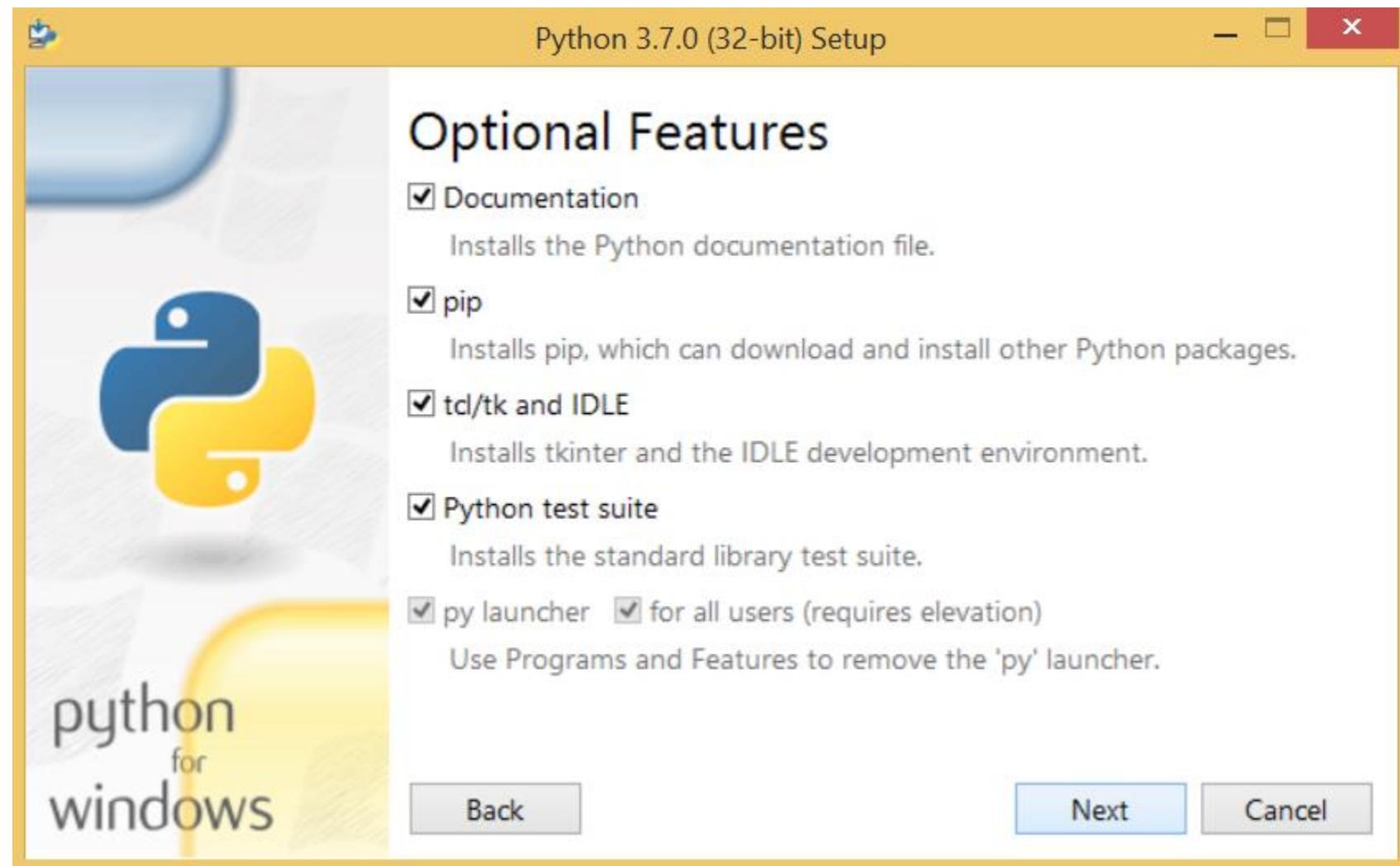
Release version	Release date	Click
<a href="#">Python 3.6.7</a>	2018-10-20	Download <a href="#">Release notes</a>
<a href="#">Python 3.5.6</a>	2018-08-02	Download <a href="#">Release notes</a>
<a href="#">Python 3.4.9</a>	2018-08-02	Download <a href="#">Release notes</a>
<a href="#">Python 3.7.0</a>	2018-06-27	Download <a href="#">Release notes</a>
<a href="#">Python 3.6.6</a>	2018-06-27	Download <a href="#">Release notes</a>
<a href="#">Python 2.7.15</a>	2018-05-01	Download <a href="#">Release notes</a>
<a href="#">Python 3.6.5</a>	2018-03-28	Download <a href="#">Release notes</a>
<a href="#">Python 3.4.8</a>	2018-02-05	Download <a href="#">Release notes</a>

[View older releases](#)

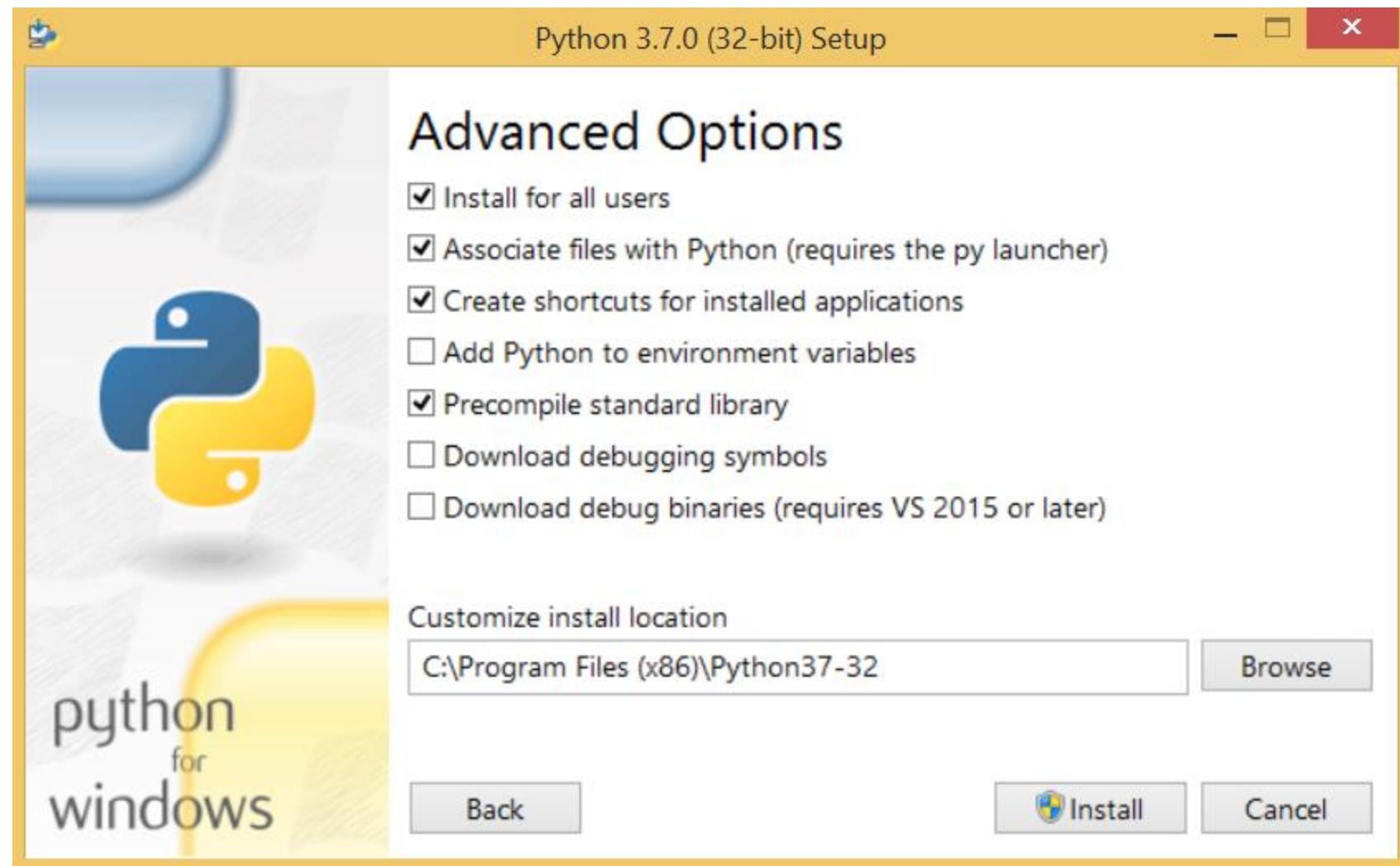
Double-click the executable file which is downloaded; the following window will open. Select Customize installation and proceed.



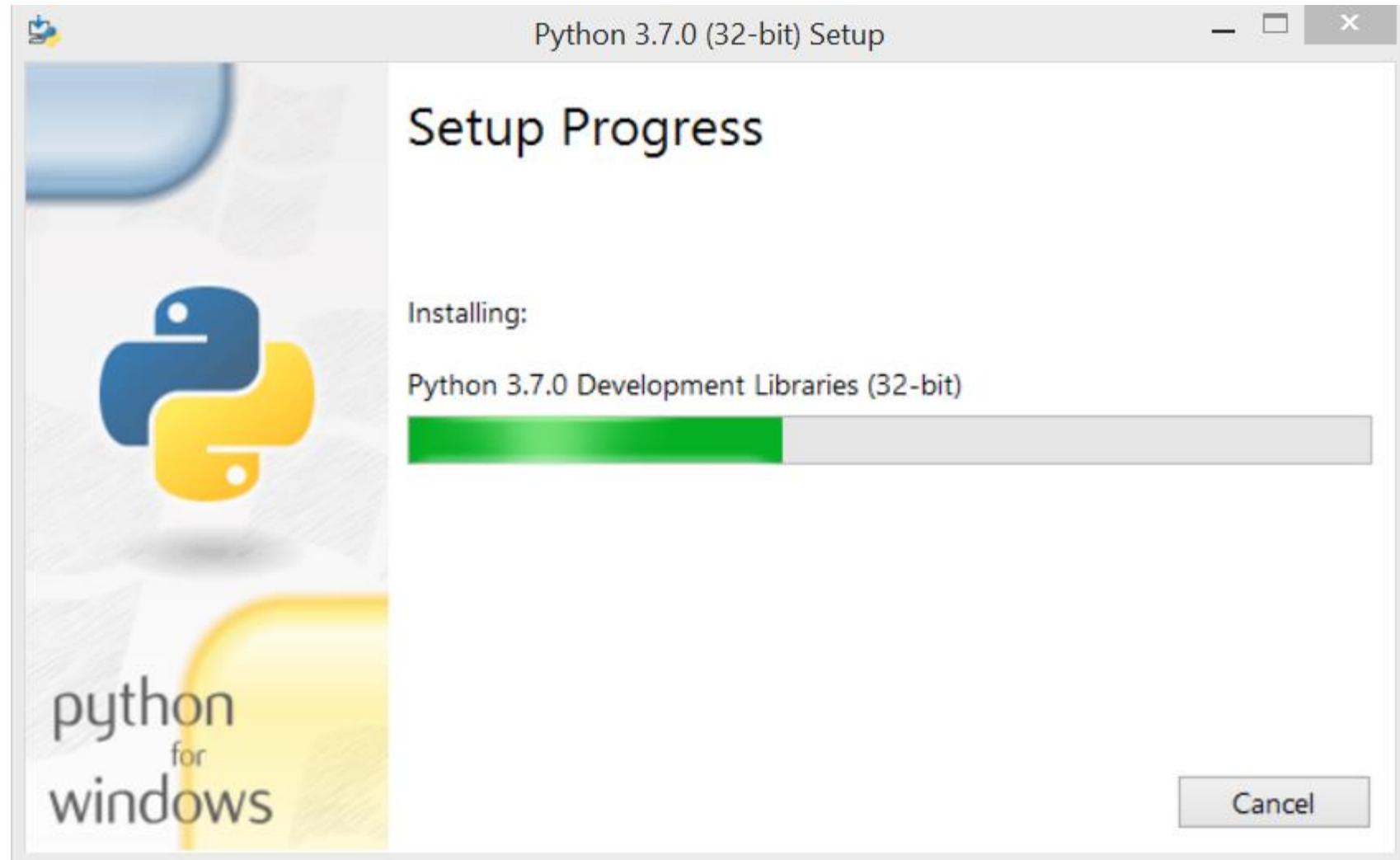
The following window shows all the optional features. All the features need to be installed and are checked by default; we need to click next to continue.



The following window shows a list of advanced options. Check all the options which you want to install and click next. Here, we must notice that the first check-box (install for all users) must be checked.



Now, we are ready to install python-3.6.7. Let's install it.



Now, try to run python on the command prompt. Type the command **python** in case of python2 or python3 in case of **python3**. It will show an error as given in the below image. It is because we haven't set the path.

0%

C:\Windows\system32\cmd.exe

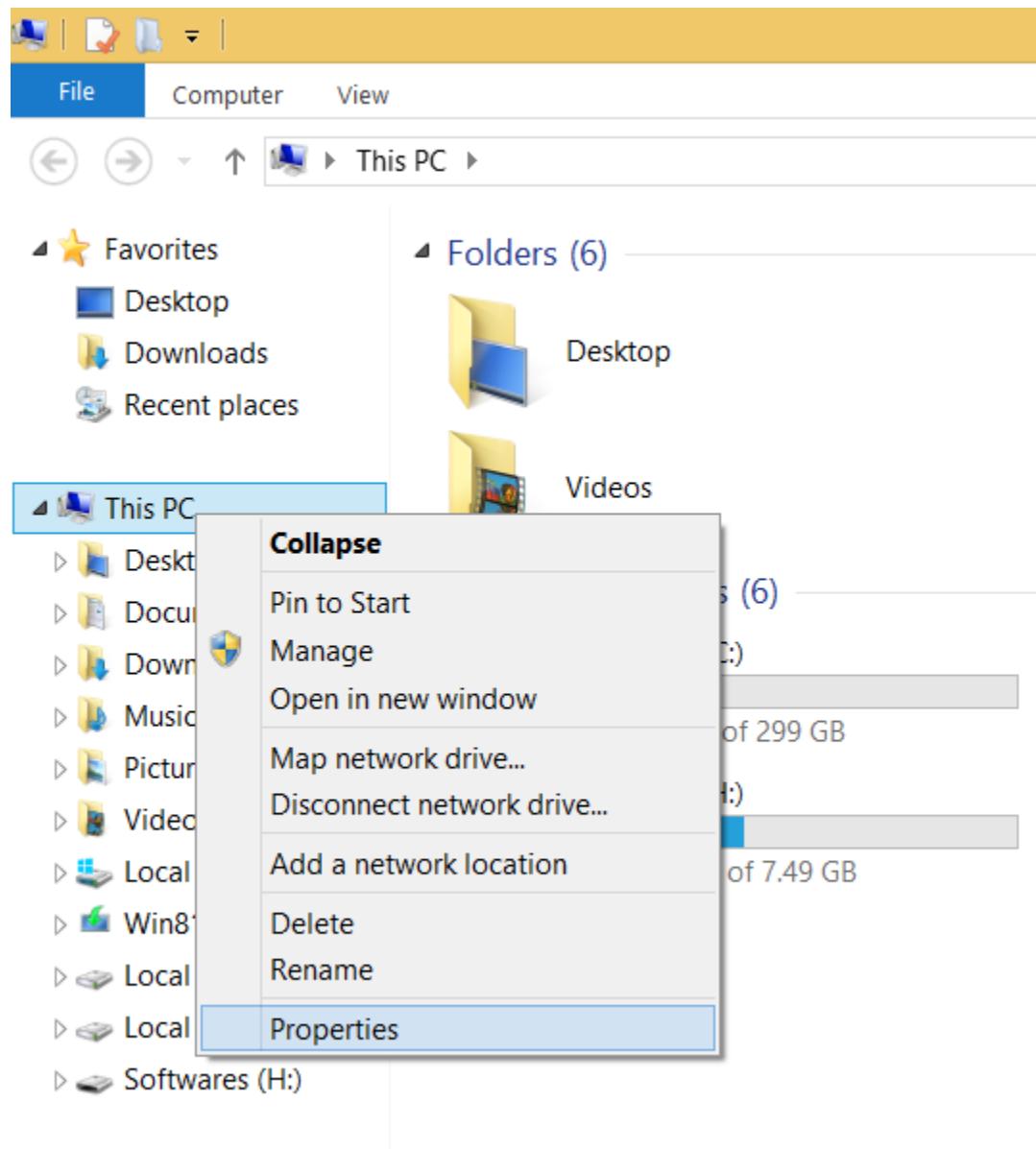


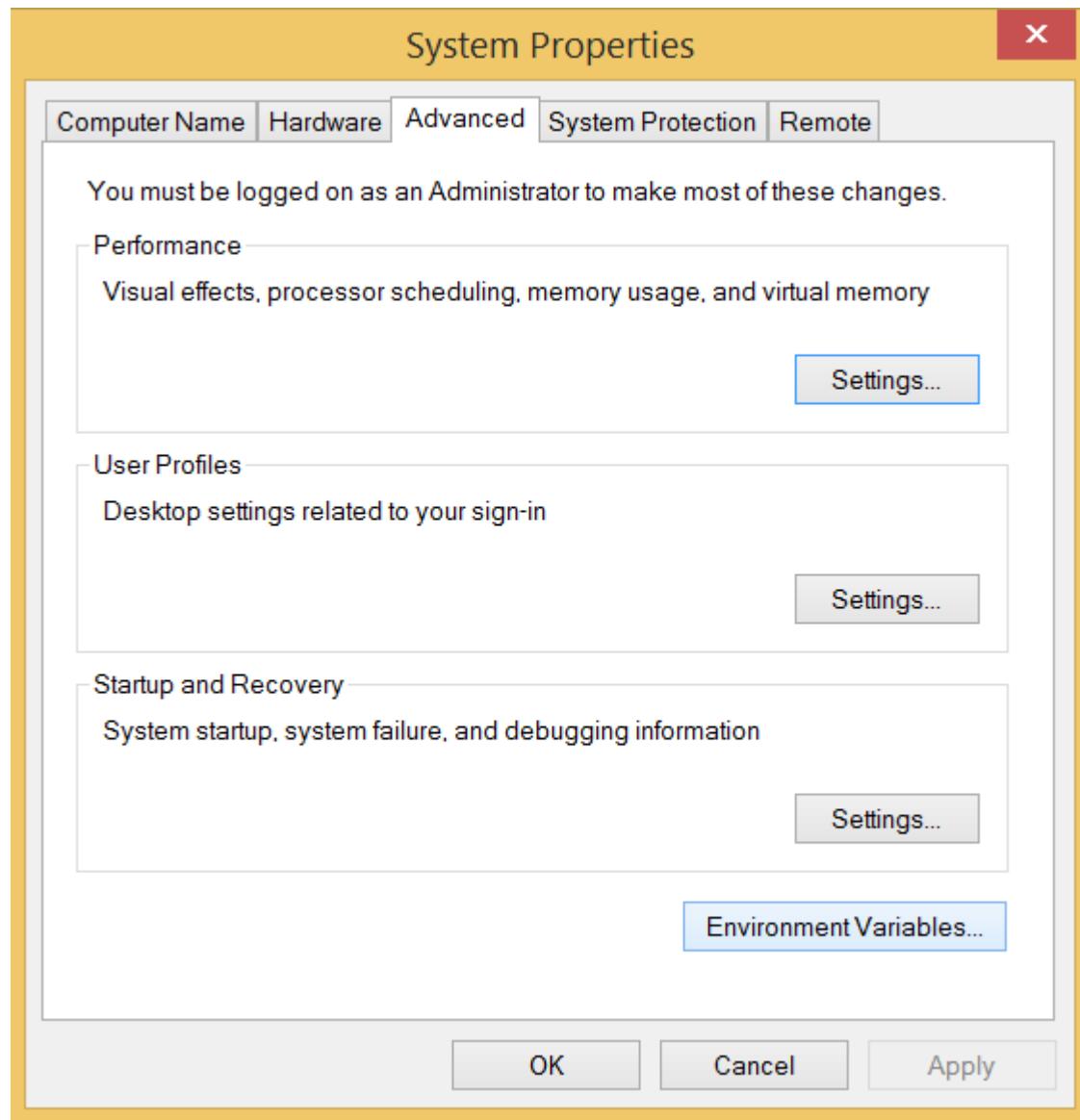
Microsoft Windows [Version 6.3.9600]  
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\developer>python  
'python' is not recognized as an internal or external command,  
operable program or batch file.

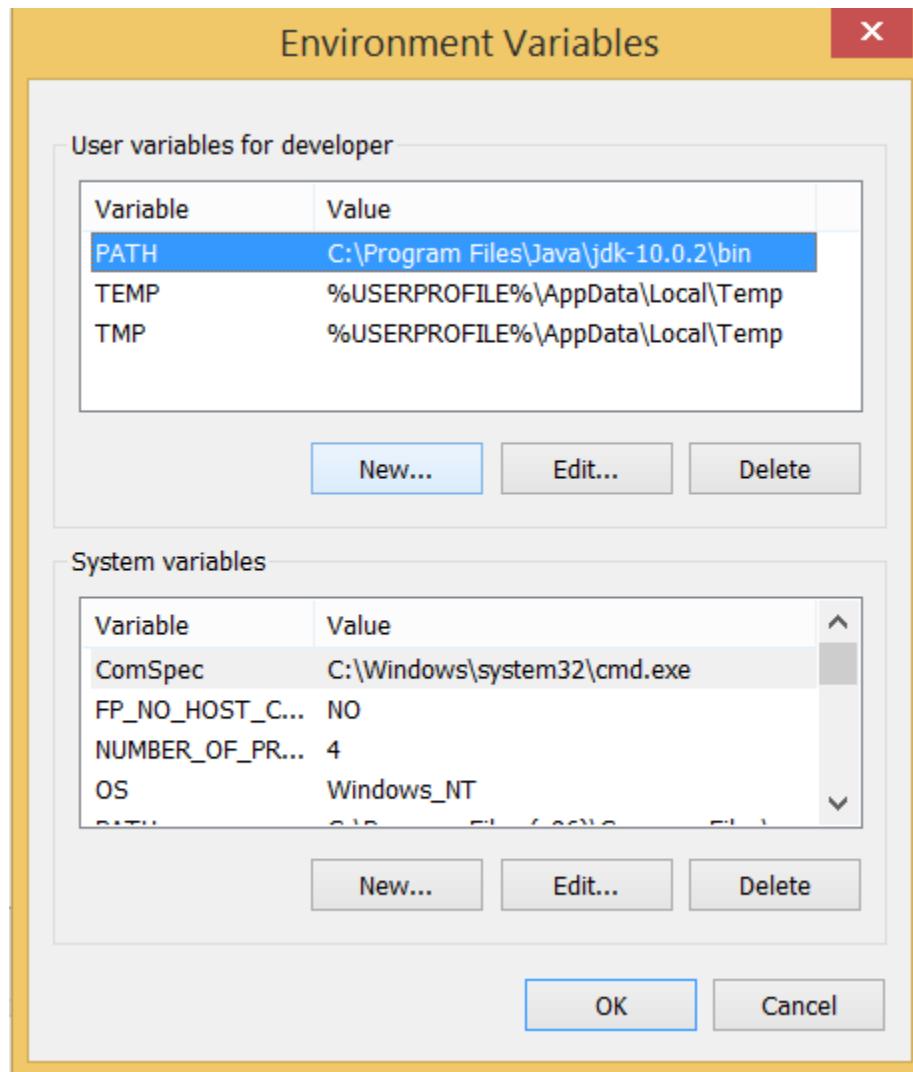
C:\Users\developer>\_

To set the path of python, we need to right click on "my computer" and go to Properties → Advanced → Environment Variables.

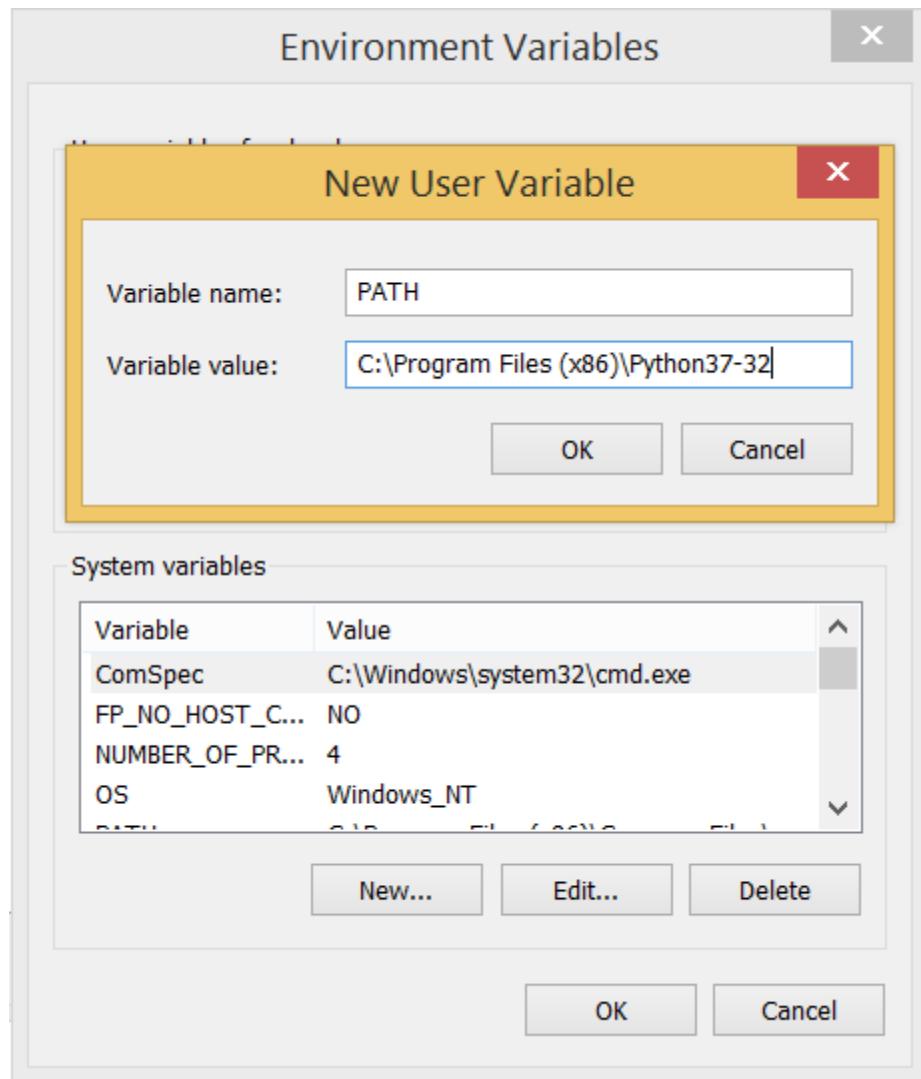




Add the new path variable in the user variable section.



Type **PATH** as the variable name and set the path to the installation directory of the python shown in the below image.



Now, the path is set, we are ready to run python on our local system. Restart CMD, and type **python** again. It will open the python interpreter shell where we can execute the python statements.

---

## Installation on Mac

To install python3 on MacOS, visit the link <https://www.javatpoint.com/how-to-install-python-on-mac> and follow the instructions given in the tutorial.

## SETTING PATH IN PYTHON

Before starting working with Python, a specific path is to set.

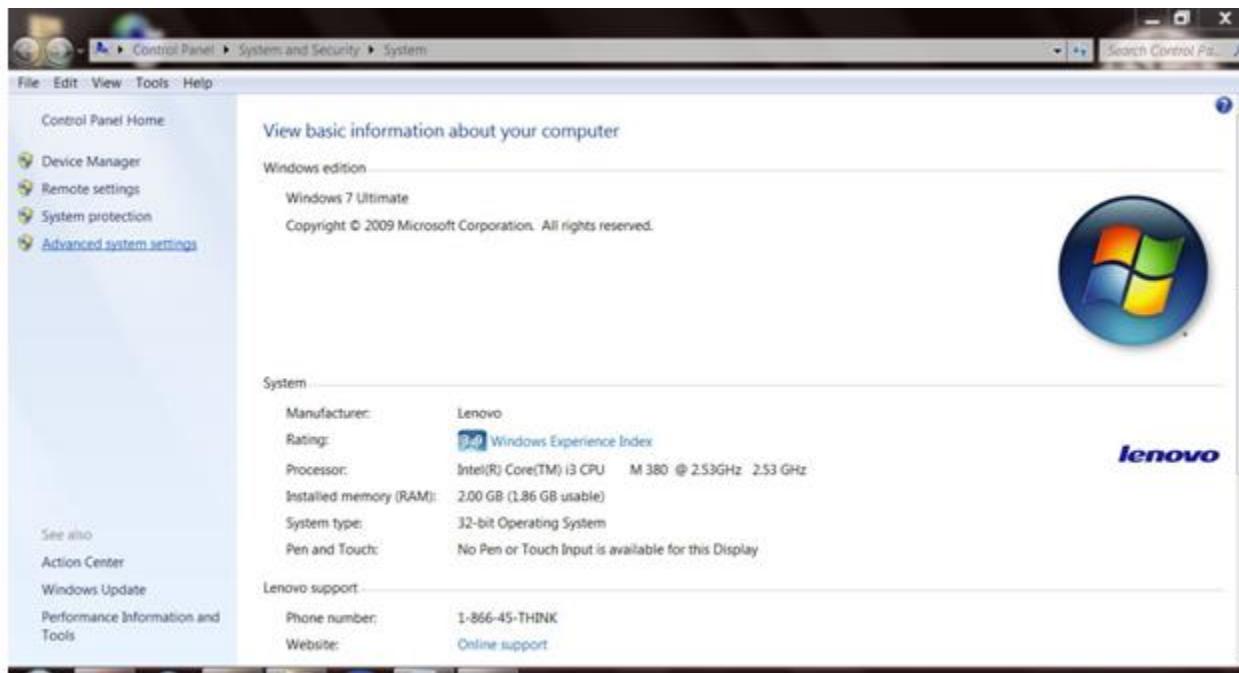
- Your Python program and executable code can reside in any directory of your system, therefore Operating System provides a specific search path that index the directories Operating System should search for executable code.
- The Path is set in the Environment Variable of My Computer properties:
- To set path follow the steps:

Right click on My Computer ->Properties ->Advanced System setting ->Environment Variable ->New

In Variable name write path and in Variable value copy path up to C://Python(i.e., path where Python is installed). Click Ok ->Ok.

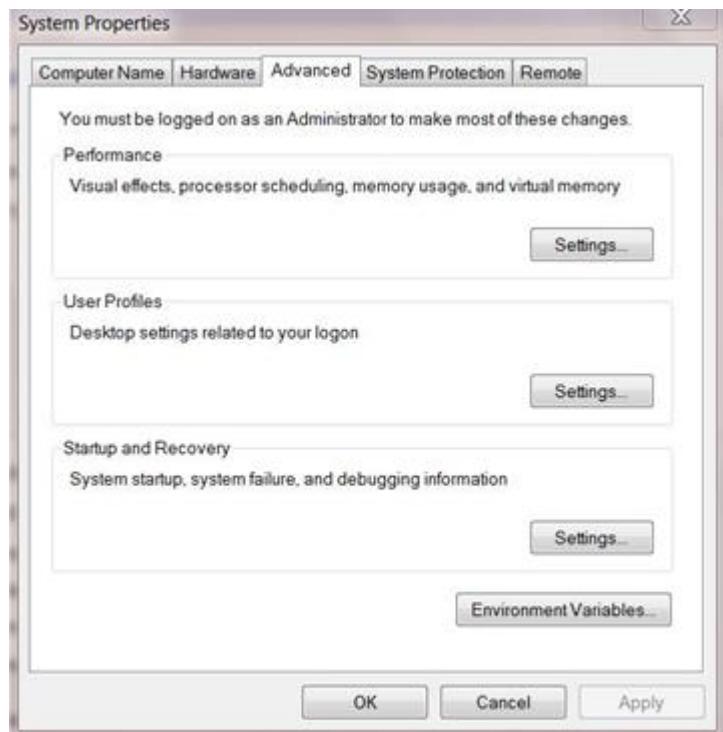
Path will be set for executing Python programs.

1. Right click on My Computer and click on properties.
2. Click on Advanced System settings



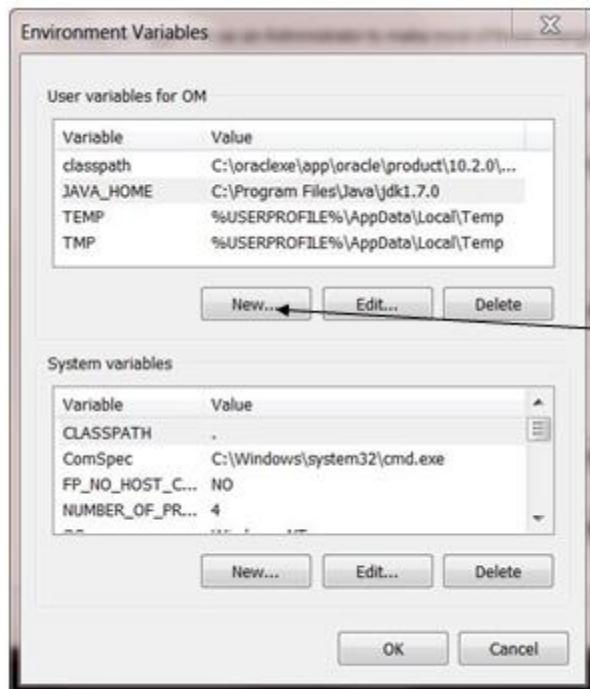
[javatpoint.com](http://javatpoint.com)

3. Click on Environment Variable tab.



[javatpoint.com](http://javatpoint.com)

4. Click on new tab of user variables.



*Click here*

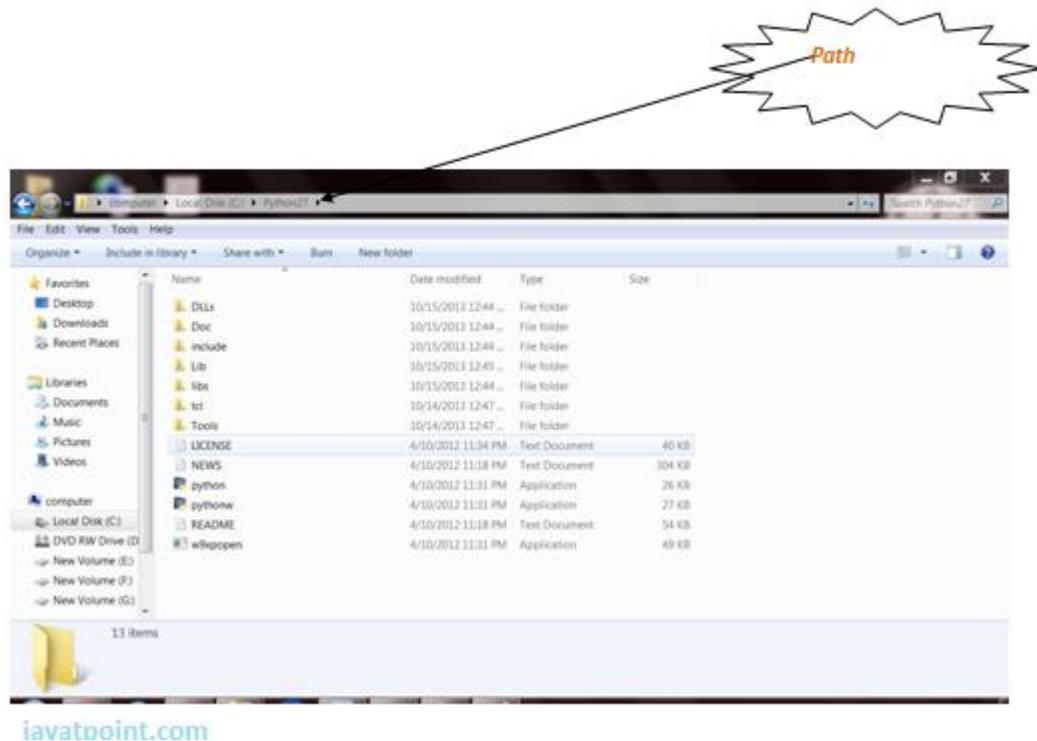
[javatpoint.com](http://javatpoint.com)

## 5. Write path in variable name



[javatpoint.com](http://javatpoint.com)

6. Copy the path of Python folder

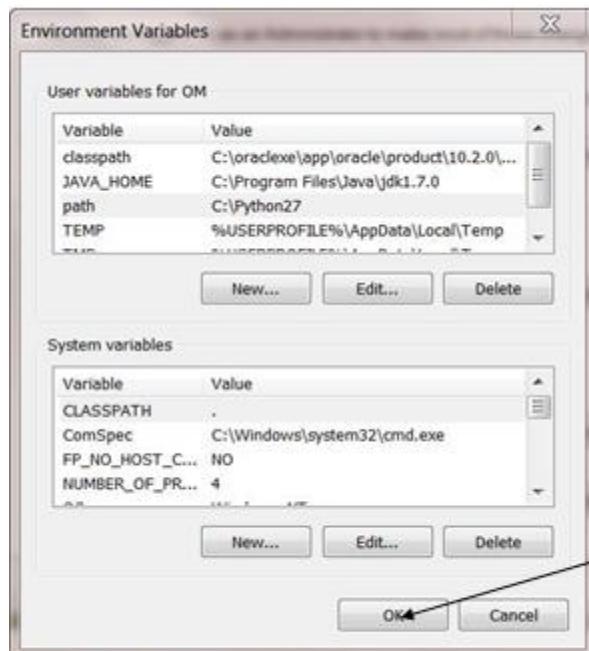


7. Paste path of Python in variable value.



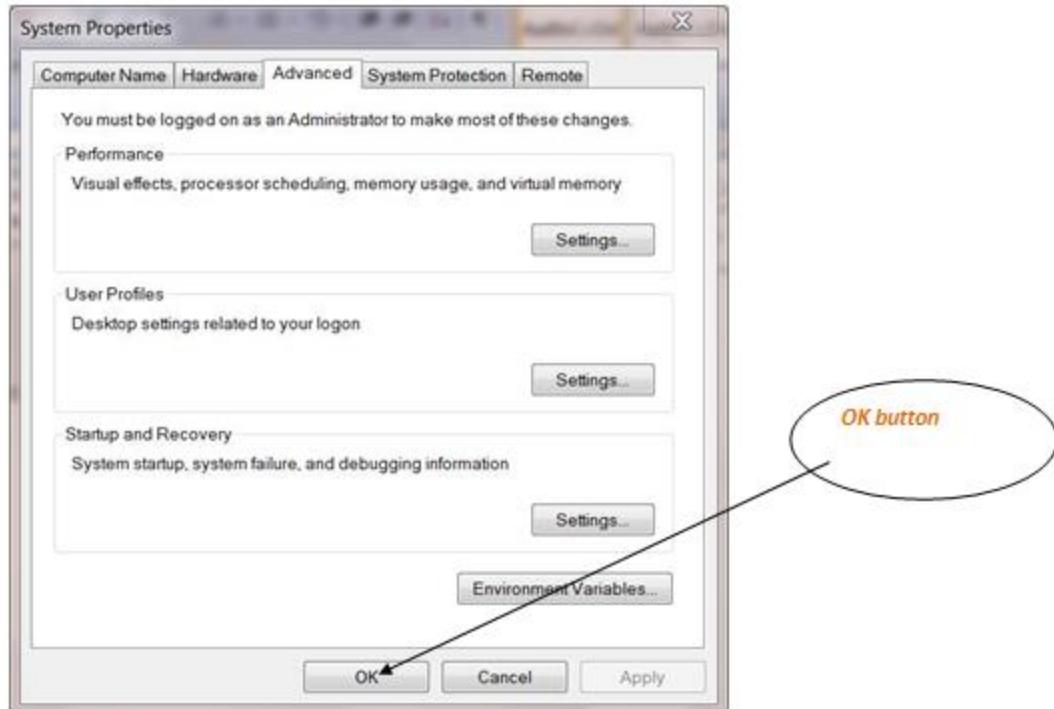
[javatpoint.com](http://javatpoint.com)

8. Click on Ok button:



[javatpoint.com](http://javatpoint.com)

9. Click on Ok button:



iavatpoint.com

G g

## First Python Program

In this Section, we will discuss the basic syntax of python by using which, we will run a simple program to print hello world on the console.

Python provides us the two ways to run a program:

- Using Interactive interpreter prompt
- Using a script file

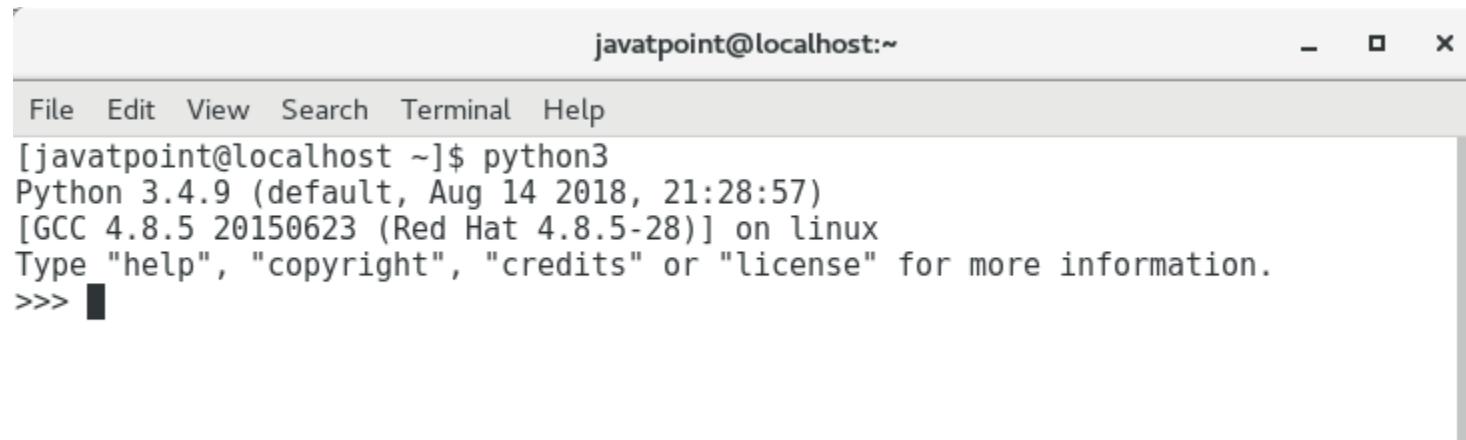
Let's discuss each one of them in detail.

## Interactive interpreter prompt

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

To open the interactive mode, open the terminal (or command prompt) and type python (python3 in case if you have python2 and python3 both installed on your system).

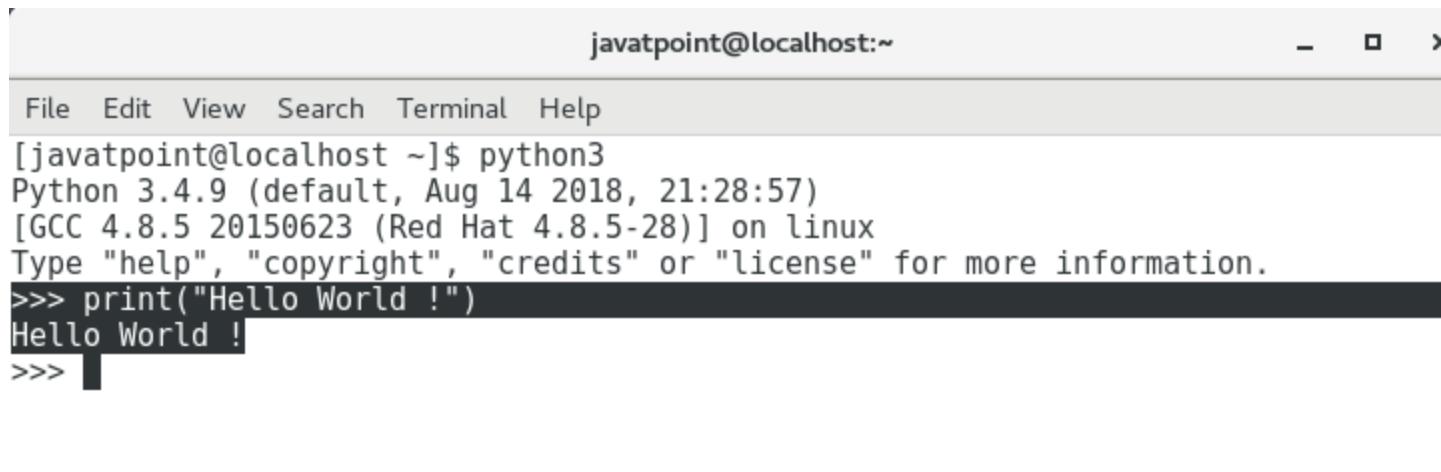
It will open the following prompt where we can execute the python statement and check their impact on the console.



A screenshot of a terminal window titled "javatpoint@localhost:~". The window has standard Linux-style window controls (minimize, maximize, close). The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The main pane displays the Python 3 interactive shell. The output shows:

```
[javatpoint@localhost ~]$ python3
Python 3.4.9 (default, Aug 14 2018, 21:28:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Let's run a python statement to print the traditional hello world on the console. Python3 provides `print()` function to print some message on the console. We can pass the message as a string into this function. Consider the following image.



```
javatpoint@localhost:~ - □ ×
File Edit View Search Terminal Help
[javatpoint@localhost ~]$ python3
Python 3.4.9 (default, Aug 14 2018, 21:28:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World !")
Hello World !
>>>
```

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

---

## Using a script file

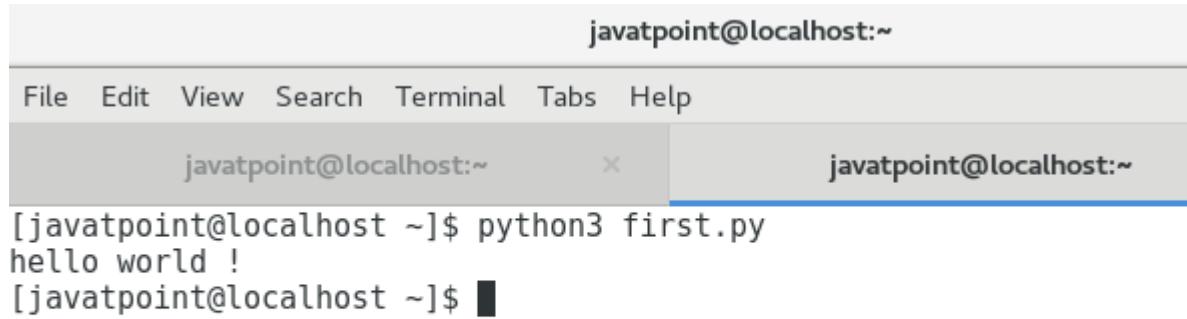
Interpreter prompt is good to run the individual statements of the code. However, we can not write the code every-time on the terminal.

We need to write our code into a file which can be executed later. For this purpose, open an editor like notepad, create a file named first.py (python used .py extension) and write the following code in it.

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

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

```
$ python3 first.py
```



A screenshot of a terminal window titled "javatpoint@localhost:~". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", "Tabs", and "Help". There are two tabs open: "javatpoint@localhost:~" (which is active) and another "javatpoint@localhost:~". The active tab contains the command "[javatpoint@localhost ~]\$ python3 first.py" followed by the output "hello world !".

```
[javatpoint@localhost ~]$ python3 first.py
hello world !
[javatpoint@localhost ~]$
```

Hence, we get our output as the message **Hello World !** is printed on the console.

---

## Get Started with PyCharm

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

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

## PyCharm installation

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

## Windows

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

---

## CentOS

To install PyCharm on CentOS, visit the link <https://www.javatpoint.com/how-to-install-pycharm-on-centos>. The link will guide you to install PyCharm on the CentOS.

---

## MacOS

To install PyCharm on MacOS, visit the link <https://www.javatpoint.com/how-to-install-pycharm-on-mac>. The link will guide you to install PyCharm on the MacOS.

---

## Ubuntu

To install PyCharm on Ubuntu, visit the link <https://www.javatpoint.com/how-to-install-pycharm-in-ubuntu>. The link will guide you to install PyCharm on Ubuntu.

---

In the upcoming section of the tutorial, we will use PyCharm to edit the python code.

## How to execute python

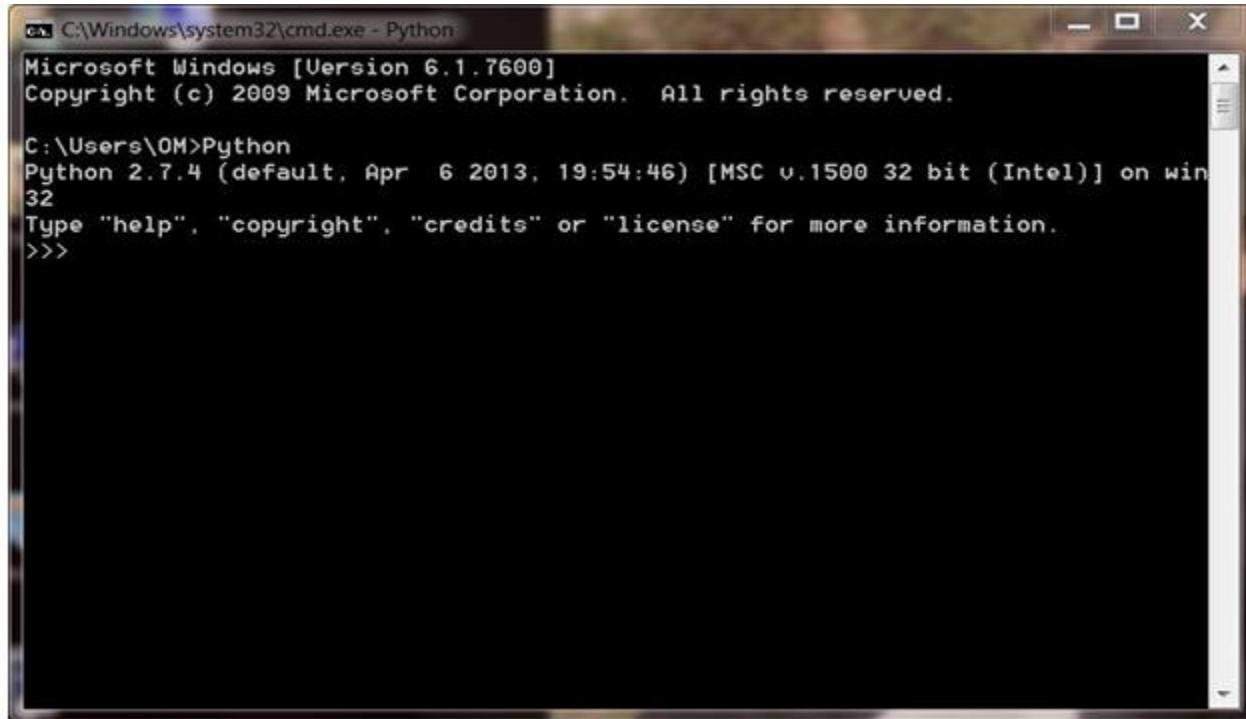
To execute Python code, we can use any approach that are given below.

### 1) Interactive Mode

Python provides Interactive Shell to execute code immediatly and produce output instantly. To get into this shell, write python in the command prompt and start working with Python.



Press Enter key and the Command Prompt will appear like:



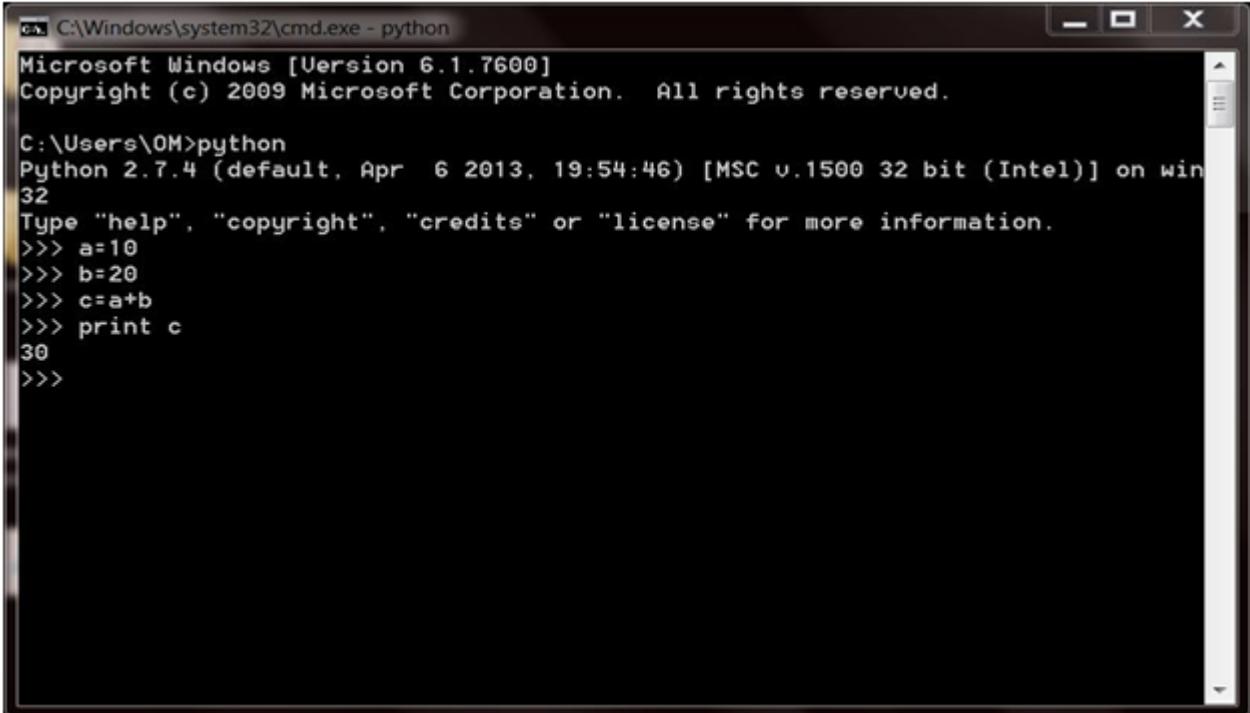
```
C:\Windows\system32\cmd.exe - Python
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>Python
Python 2.7.4 (default, Apr  6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[javatpoint.com](http://javatpoint.com)

Now we can execute our Python commands.

**Eg:**



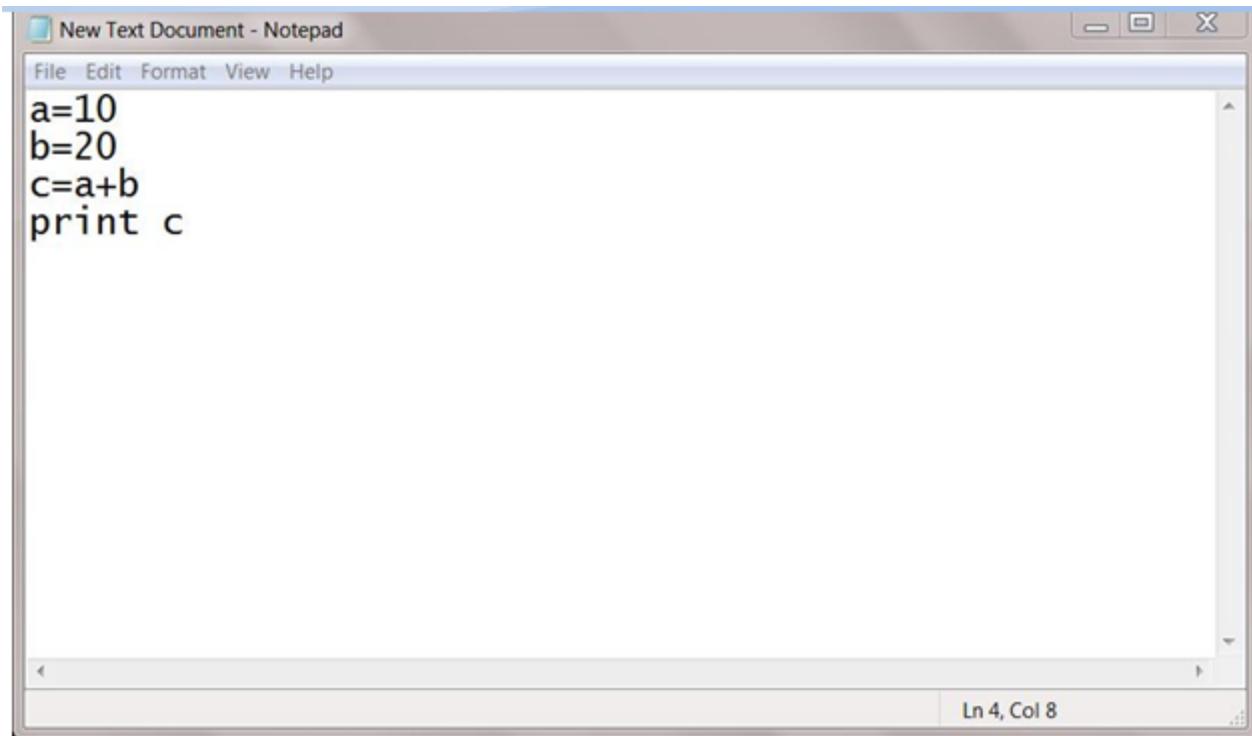
```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>python
Python 2.7.4 (default, Apr  6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=10
>>> b=20
>>> c=a+b
>>> print c
30
>>>
```

[iavatpoint.com](http://iavatpoint.com)

## 2) Script Mode

Using Script Mode, we can write our Python code in a separate file of any editor in our Operating System.



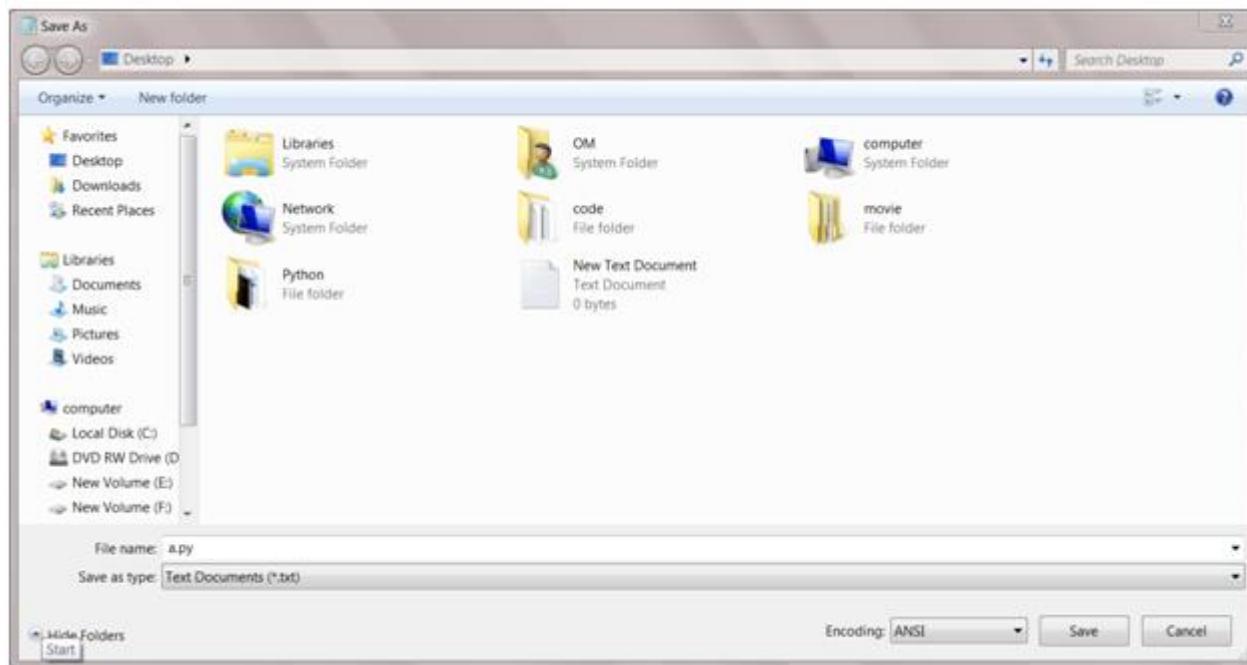
A screenshot of a Windows Notepad window titled "New Text Document - Notepad". The window contains the following Python code:

```
a=10
b=20
c=a+b
print c
```

The cursor is positioned at the end of the fourth line. The status bar at the bottom right shows "Ln 4, Col 8".

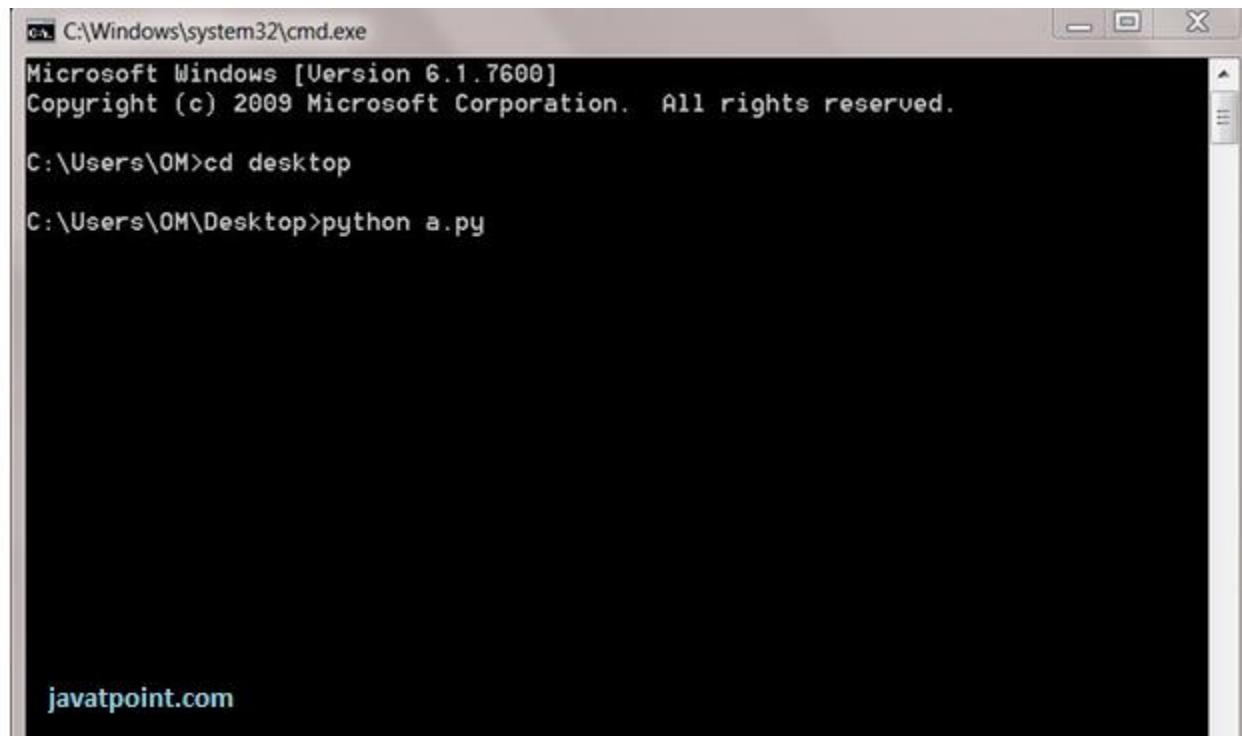
[iavatpoint.com](http://iavatpoint.com)

Save it by **.py** extension.



[javatpoint.com](http://javatpoint.com)

Now open Command prompt and execute it by :



A screenshot of a Windows Command Prompt window titled "cmd.exe". The window shows the following text:  
Microsoft Windows [Version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\Users\OM>cd desktop  
C:\Users\OM\Desktop>python a.py

javatpoint.com

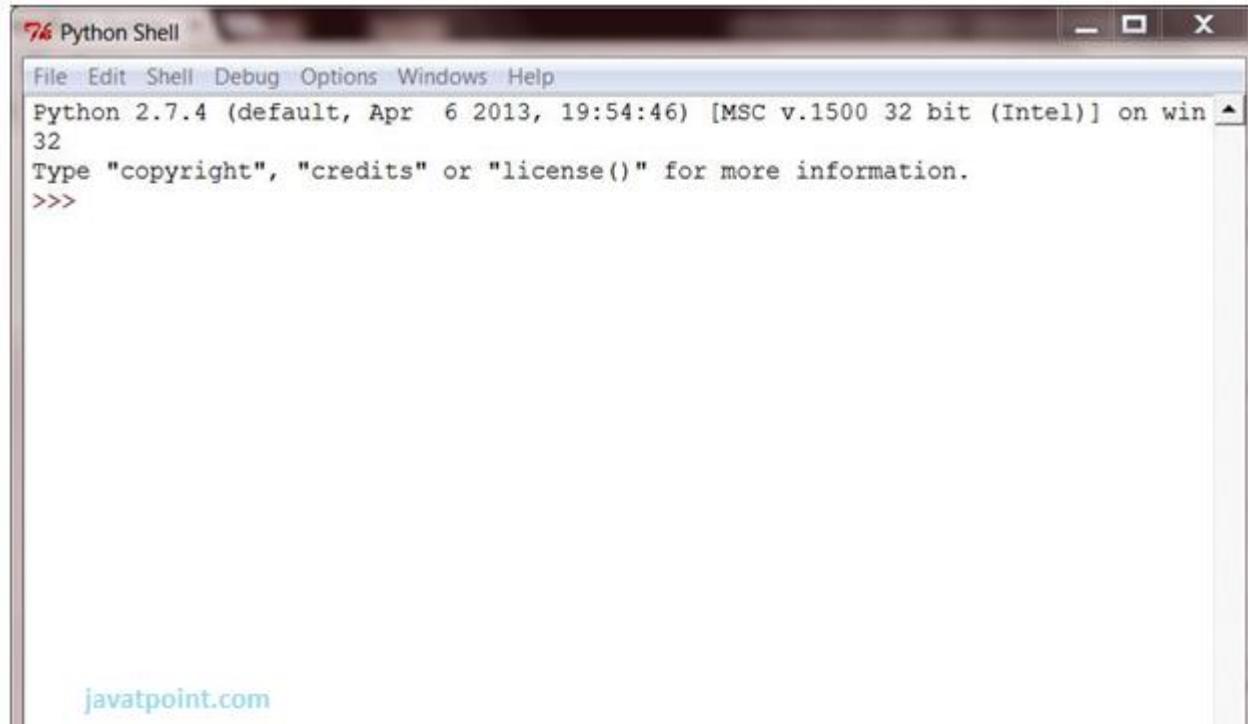
NOTE: Path in the command prompt should be location of saved file.where you have saved your file. In the above case file should be saved at desktop.

### 3) Using IDE (Integrated Development Environment)

We can execute our Python code using a Graphical User Interface (GUI).

All you need to do is:

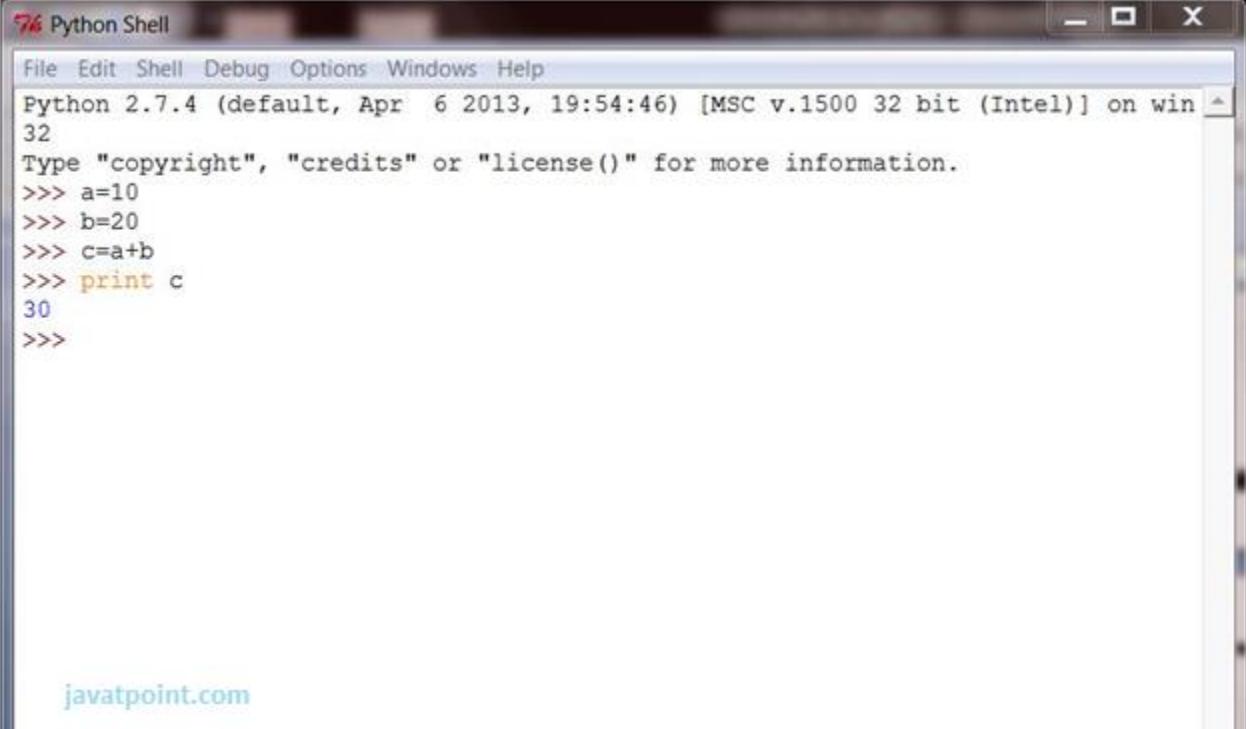
Click on Start button -> All Programs -> Python -> IDLE(Python GUI)



We can use both Interactive as well as Script mode in IDE.

### 1) Using Interactive mode:

Execute our Python code on the Python prompt and it will display result simultaneously.



The screenshot shows a Python Shell window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main area displays the following Python session:

```
File Edit Shell Debug Options Windows Help
Python 2.7.4 (default, Apr  6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=10
>>> b=20
>>> c=a+b
>>> print c
30
>>>
```

In the bottom left corner of the window, there is a watermark that reads "javatpoint.com".

## 2) Using Script Mode:

- i) Click on Start button -> All Programs -> Python -> IDLE(Python GUI)
- ii) Python Shell will be opened. Now click on File -> New Window.

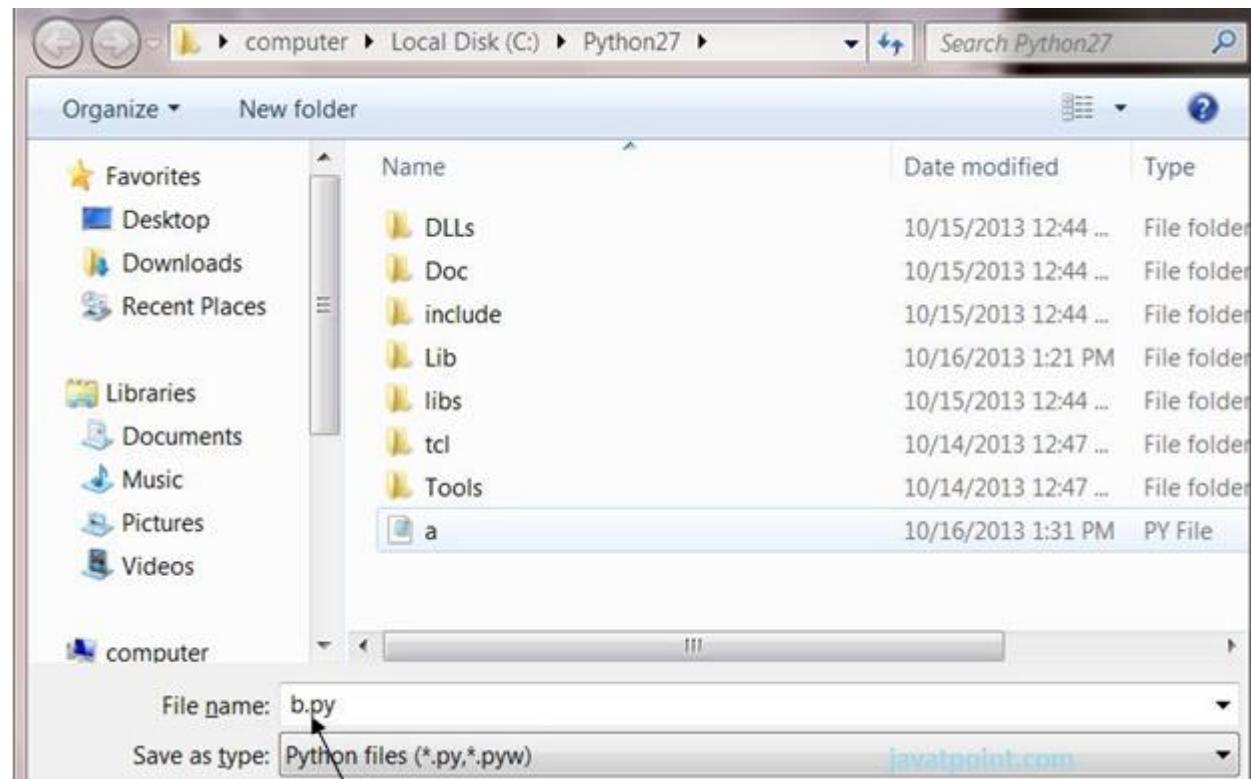
A new Editor will be opened. Write our Python code here.

The screenshot shows a Java code editor window titled '\*Untitled\*'. The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The code area contains the following Java code:

```
a=10  
b=20  
c=a+b  
print c
```

A watermark 'javatpoint.com' is visible at the bottom left of the editor window.

Click on file -> save as



Run code by clicking on Run in the Menu bar.

Run -> Run Module

Result will be displayed on a new Python shell as:

## Python Variables

Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value.

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

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

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

## Identifier Naming

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

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

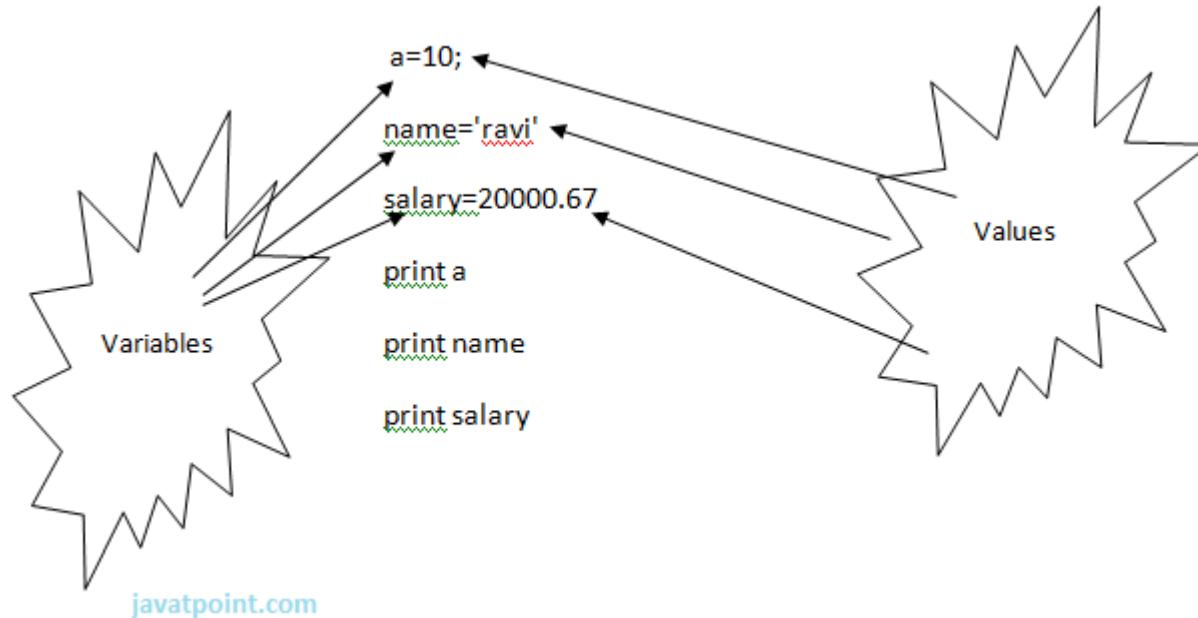
## Declaring Variable and Assigning Values

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

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

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

Eg:



Output:

1. >>>
2. 10
3. ravi
4. 20000.67
5. >>>

## Multiple Assignment

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

We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Lets see given examples.

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

1. `x=y=z=50`
2. `print x`
3. `print y`
4. `print z`

#### **Output:**

1. `>>>`
2. `50`
3. `50`
4. `50`
5. `>>>`

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

#### **Eg:**

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

#### **Output:**

1. >>>
2. 5
3. 10
4. 15
5. >>>

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

## **Basic Fundamentals:**

This section contains the basic fundamentals of Python like :

**i) Tokens and their types.**

**ii) Comments**

**a) Tokens:**

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

There are following tokens in Python:

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

## Tuples:

- Tuple is another form of collection where different type of data can be stored.
- It is similar to list where data is separated by commas. Only the difference is that list uses square bracket and tuple uses parenthesis.
- Tuples are enclosed in parenthesis and cannot be changed.

### Eg:

```
1. >>> tuple=('rahul',100,60.4,'deepak')
2. >>> tuple1=('sanjay',10)
3. >>> tuple
4. ('rahul', 100, 60.4, 'deepak')
5. >>> tuple[2:]
6. (60.4, 'deepak')
7. >>> tuple1[0]
8. 'sanjay'
9. >>> tuple+tuple1
10. ('rahul', 100, 60.4, 'deepak', 'sanjay', 10)
11.>>>
```

## Dictionary:

- Dictionary is a collection which works on a key-value pair.
- It works like an associated array where no two keys can be same.
- Dictionaries are enclosed by curly braces ({{}}) and values can be retrieved by square bracket([]).

### Eg:

```
1. >>> dictionary={'name':'charlie','id':100,'dept':'it'}
```

```
2. >>> dictionary
3. {'dept': 'it', 'name': 'charlie', 'id': 100}
4. >>> dictionary.keys()
5. ['dept', 'name', 'id']
6. >>> dictionary.values()
7. ['it', 'charlie', 100]
8. >>>
```

## Python Data Types

Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

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

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

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

### Output:

```
<type 'int'>
<type 'str'>
<type 'float'>
```

# Standard data types

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

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

1. [Numbers](#)
2. [String](#)
3. [List](#)
4. [Tuple](#)
5. [Dictionary](#)

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

---

## Numbers

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

1. `a = 3 , b = 5 #a and b are number objects`

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)

#### 4. complex (complex numbers like $2.14j$ , $2.0 + 2.3j$ , etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e.,  $x + iy$  where x and y denote the real and imaginary parts respectively).

---

## String

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

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

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

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

The following example illustrates the string handling in python.

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

### Output:

```
he
o
hello javatpointhello javatpoint
hello javatpoint how are you
```

---

## List

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

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

Consider the following example.

1. l = [1, "hi", "python", 2]
2. **print** (l[3:]);
3. **print** (l[0:2]);
4. **print** (l);
5. **print** (l + l);
6. **print** (l \* 3);

### Output:

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

---

## Tuple

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

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

Let's see a simple example of the tuple.

1. t = ("hi", "python", 2)
2. **print** (t[1:]);
3. **print** (t[0:1]);
4. **print** (t);
5. **print** (t + t);
6. **print** (t \* 3);
7. **print** (type(t))
8. t[2] = "hi";

### Output:

```
('python', 2)
('hi',)
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

## Dictionary

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

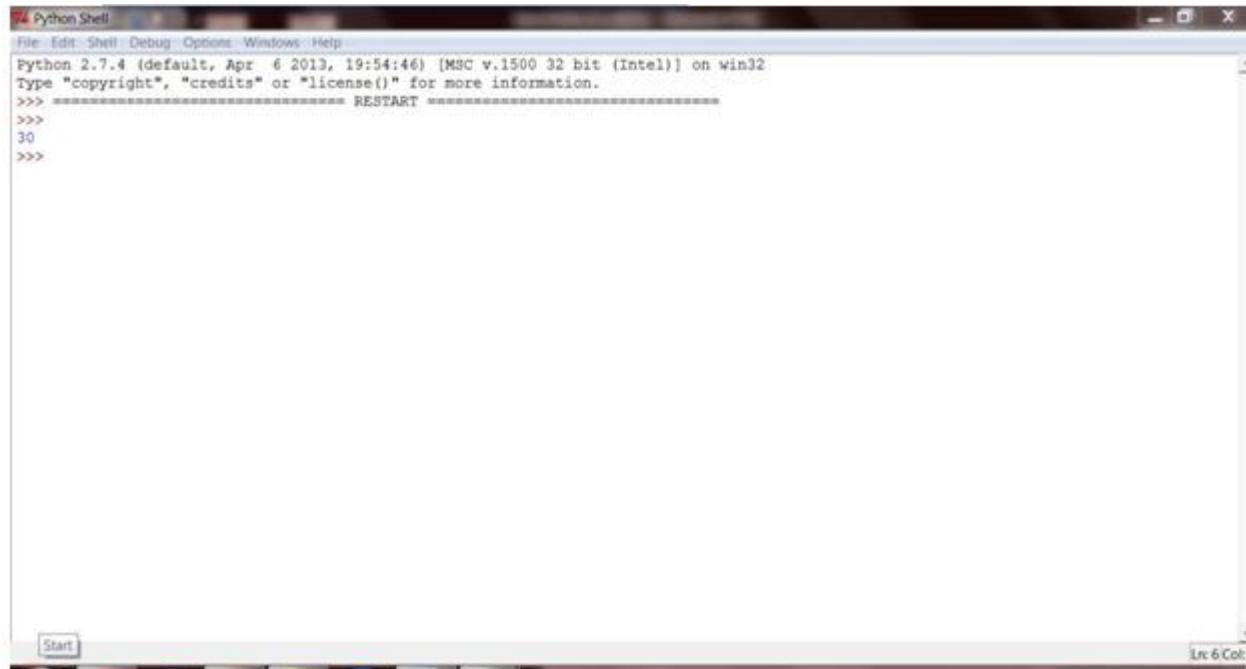
The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

1. `d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};`
2. `print("1st name is "+d[1]);`
3. `print("2nd name is "+ d[4]);`
4. `print (d);`
5. `print (d.keys());`
6. `print (d.values());`

### Output:

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

A screenshot of the Python Shell window. The title bar says "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, Help. The main window shows the Python 2.7.4 startup message: "Python 2.7.4 (default, Apr 6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win32", "Type "copyright", "credits" or "license()" for more information.", and three blank lines starting with '>>>'. The status bar at the bottom right shows "Ln: 6 Col: 4".

```
Python 2.7.4 (default, Apr 6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
30
>>>
```

[javatpoint.com](http://javatpoint.com)

## Python Keywords

Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

True	False	None	and	as
asset	def	class	continue	break

else	finally	elif	del	except
global	for	if	from	import
raise	try	or	return	pass
nonlocal	in	not	is	lambda

## Python Literals

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

Python support the following literals:

### I. String literals:

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

**Eg:**

"Aman" , '12345'

### Types of Strings:

There are two types of Strings supported in Python:

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

**Eg:**

1. >>> text1='hello'

b).Multi line String- A piece of text that is spread along multiple lines is known as Multiple line String.

There are two ways to create Multiline Strings:

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

**Eg:**

1. >>> text1='hello\  
2. user'  
3. >>> text1  
4. 'hellouser'  
5. >>>

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

**Eg:**

1. >>> str2="""welcome  
2. to  
3. SSSIT""  
4. >>> **print** str2  
5. welcome  
6. to  
7. SSSIT  
8. >>>

**II.Numeric literals:**

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

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

### **III. Boolean literals:**

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

### **IV. Special literals.**

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

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

Eg:

1. `>>> val1=10`
2. `>>> val2=None`
3. `>>> val1`
4. `10`
5. `>>> val2`
6. `>>> print val2`
7. `None`
8. `>>>`

## **V.Literal Collections.**

Collections such as tuples, lists and Dictionary are used in Python.

### **List:**

- List contain items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by commas(,) and enclosed within a square brackets([]). We can store different type of data in a List.
- Value stored in a List can be retrieved using the slice operator([] and [:]).
- The plus sign (+) is the list concatenation and asterisk(\*) is the repetition operator.

### **Eg:**

1. >>> list=['aman',678,20.4,'saurav']
2. >>> list1=[456,'rahul']
3. >>> list
4. ['aman', 678, 20.4, 'saurav']
5. >>> list[1:3]
6. [678, 20.4]
7. >>> list+list1
8. ['aman', 678, 20.4, 'saurav', 456, 'rahul']
9. >>> list1\*2
10. [456, 'rahul', 456, 'rahul']
- 11.>>>

# **Python Operators**

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

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

## Arithmetic operators

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

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

Operator	Description
+ (Addition)	It is used to add two operands. For example, if $a = 20$ , $b = 10 \Rightarrow a+b = 30$

<b>- (Subtraction)</b>	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if $a = 20$ , $b = 10 \Rightarrow a - b = 10$
<b>/ (divide)</b>	It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a/b = 2$
<b>*</b> <b>(Multiplication)</b>	It is used to multiply one operand with the other. For example, if $a = 20$ , $b = 10 \Rightarrow a * b = 200$
<b>% (reminder)</b>	It returns the remainder after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a \% b = 0$
<b>** (Exponent)</b>	It is an exponent operator represented as it calculates the first operand power to second operand.
<b>// (Floor division)</b>	It gives the floor value of the quotient produced by dividing the two operands.

## Comparison operator

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

<b>Operator</b>	<b>Description</b>
<code>==</code>	If the value of two operands is equal, then the condition becomes true.
<code>!=</code>	If the value of two operands is not equal then the condition becomes true.
<code>&lt;=</code>	If the first operand is less than or equal to the second operand, then the condition becomes true.
<code>&gt;=</code>	If the first operand is greater than or equal to the second operand, then the condition becomes true.
<code>&lt;&gt;</code>	If the value of two operands is not equal, then the condition becomes true.
<code>&gt;</code>	If the first operand is greater than the second operand, then the condition becomes true.
<code>&lt;</code>	If the first operand is less than the second operand, then the condition becomes true.

## Python assignment operators

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

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

# Bitwise operator

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

## For example,

1. if a = 7;
2. b = 6;
3. then, binary (a) = 0111
4. binary (b) = 0011
- 5.
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
(binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.

~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

## Logical Operators

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

Operator	Description
And	If both the expression are true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}$ , $b \rightarrow \text{true} \Rightarrow a \text{ and } b \rightarrow \text{true}$ .
Or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, $a \rightarrow \text{true}$ , $b \rightarrow \text{false} \Rightarrow a \text{ or } b \rightarrow \text{true}$ .
Not	If an expression <b>a</b> is true then <code>not (a)</code> will be false and vice versa.

---

## Membership Operators

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

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

---

## Identity Operators

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

---

# Operator Precedence

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

Operator	Description
**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus and minus.
* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus and minus
>> <<	Left shift and right shift
&	Binary and.
^	Binary xor and or
<= < > >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).

<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

## Python Comments

Comments in Python can be used to explain any program code. It can also be used to hide the code as well.

Comments are the most helpful stuff of any program. It enables us to understand the way, a program works. In python, any statement written along with # symbol is known as a comment. The interpreter does not interpret the comment.

Comment is not a part of the program, but it enhances the interactivity of the program and makes the program readable.

Python supports two types of comments:

### 1) Single Line Comment:

In case user wants to specify a single line comment, then comment must start with #?

**Eg:**

1. `# This is single line comment.`
2. `print "Hello Python"`

**Output:**

```
Hello Python
```

**2) Multi Line Comment:**

Multi lined comment can be given inside triple quotes.

**eg:**

1. `""" This`
2.  `Is`
3.  `Multipline comment""`

**eg:**

1. `#single line comment`
2. `print "Hello Python"`
3. `"""This is`
4. `multiline comment""`

**Output:**

```
Hello Python
```

[next](#) → ← [prev](#)

## Python If-else statements

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

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

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

## Indentation in Python

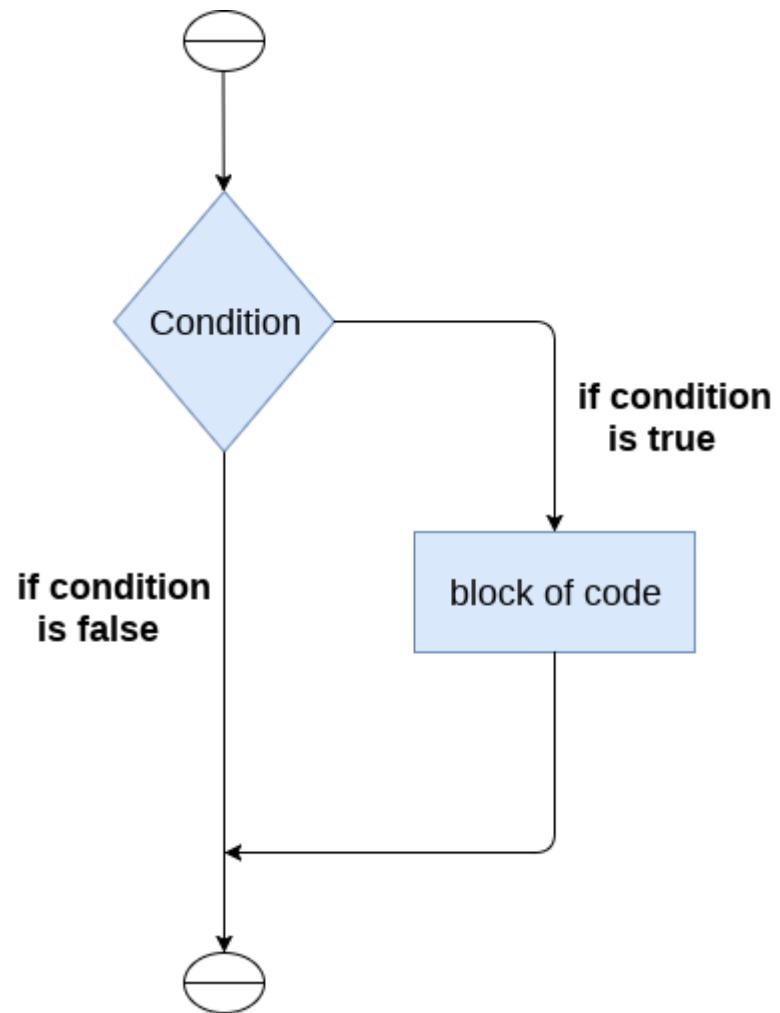
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

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

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

## The if statement

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



The syntax of the if-statement is given below.

1. **if** expression:

2. statement

### Example 1

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

#### Output:

```
enter the number?10
Number is even
```

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

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

#### Output:

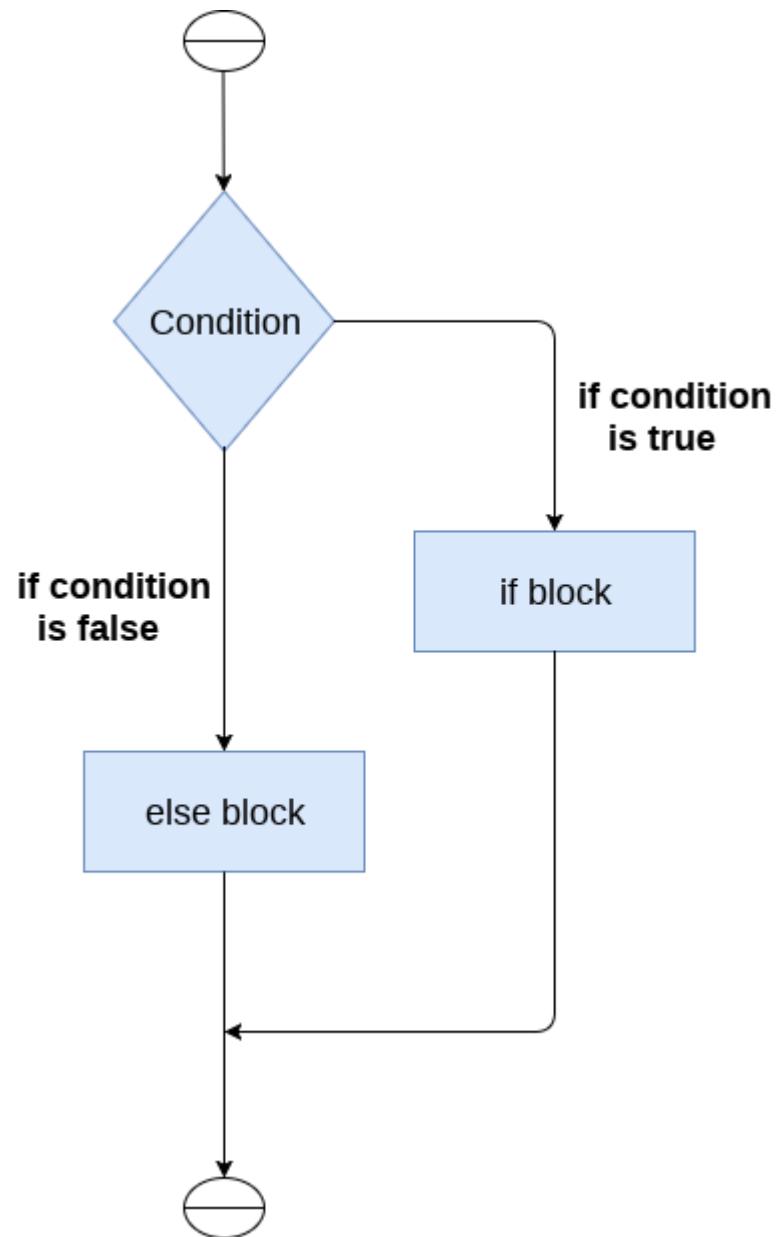
```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

---

## The if-else statement

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

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



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

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

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

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

#### **Output:**

```
Enter your age? 90
You are eligible to vote !!
```

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

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

#### **Output:**

```
enter the number?10
```

Number is even

---

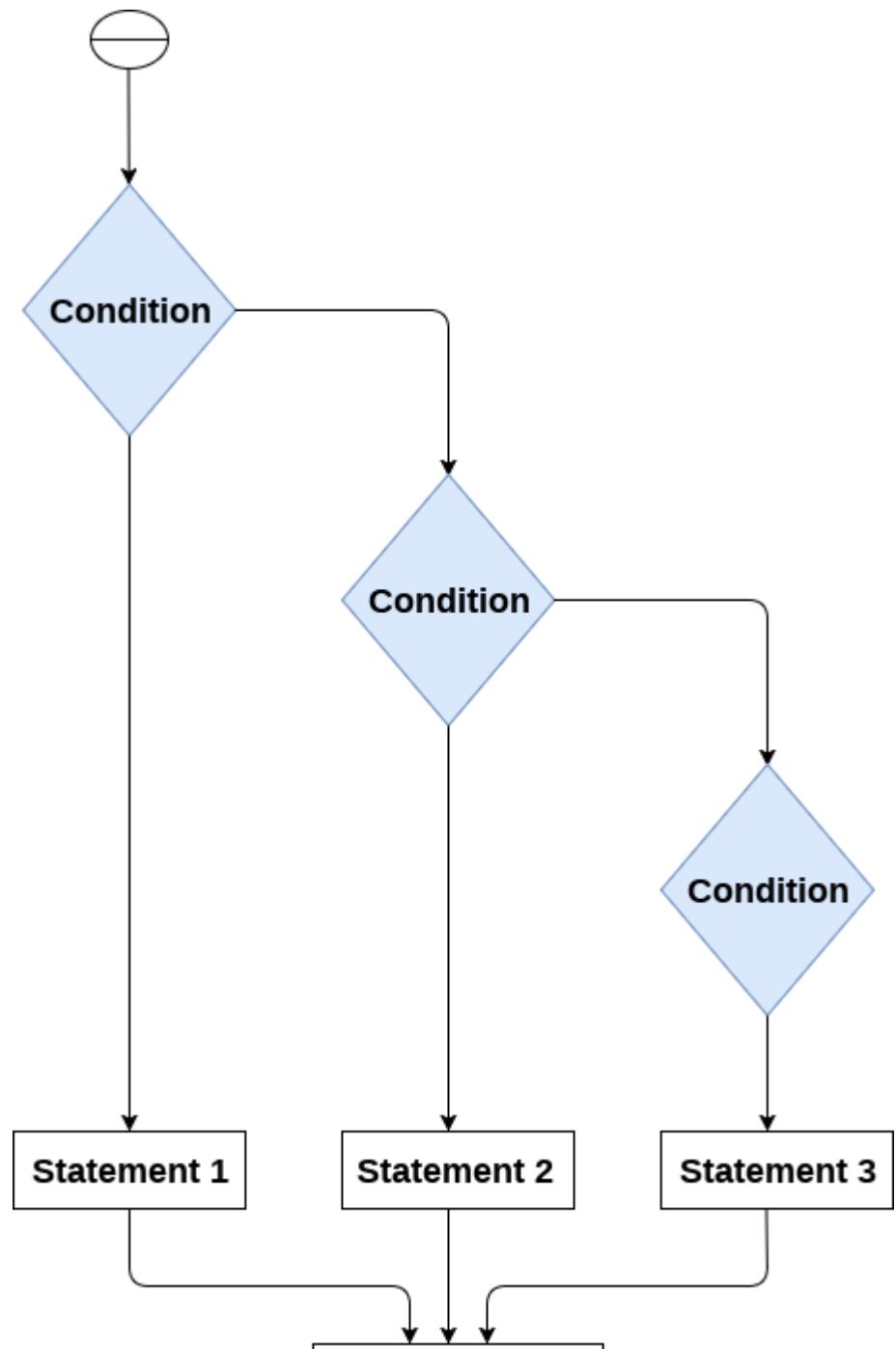
## The elif statement

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

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

The syntax of the elif statement is given below.

1. **if** expression 1:
2.   **# block of statements**
- 3.
4. **elif** expression 2:
5.   **# block of statements**
- 5.
7. **elif** expression 3:
3.   **# block of statements**
- 9.
10. **else**:
11.   **# block of statements**



## Example 1

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

### Output:

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

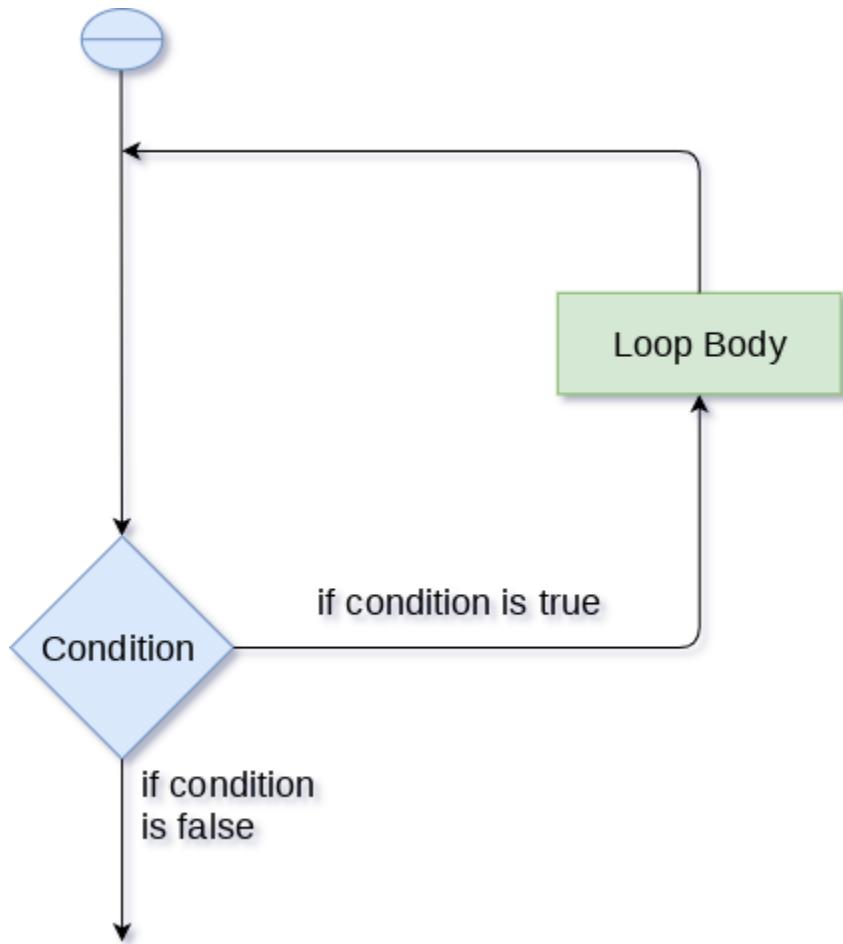
## Example 2

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

# Python Loops

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



## Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to

print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

## Advantages of loops

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

Loop Statement	Description
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

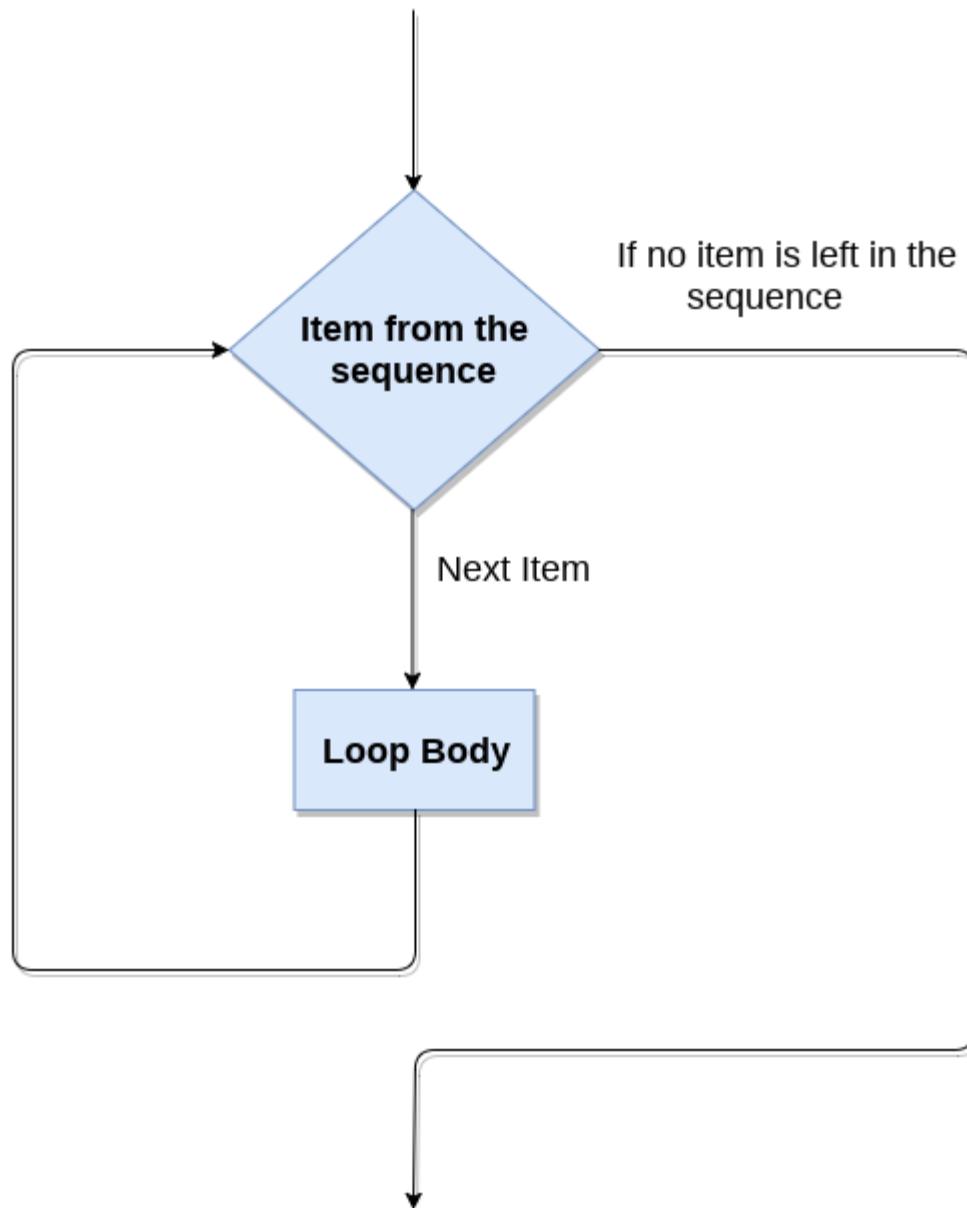
do-while loop	The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).
------------------	---

## Python for loop

The **for loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

1. **for** iterating\_var **in** sequence:
2. statement(s)



## Example

```
1. i=1
2. n=int(input("Enter the number up to which you want to print the natural numbers?"))
3. for i in range(0,10):
4.     print(i,end = ' ')
```

### Output:

```
0 1 2 3 4 5 6 7 8 9
```

## Python for loop example : printing the table of the given number

```
1. i=1;
2. num = int(input("Enter a number:"));
3. for i in range(1,11):
4.     print("%d X %d = %d"%(num,i,num*i));
```

### Output:

```
Enter a number:10
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

## Nested for loop in python

Python allows us to nest any number of for loops inside a for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax of the nested for loop in python is given below.

1. **for** iterating\_var1 **in** sequence:
2.   **for** iterating\_var2 **in** sequence:
3.     **#block of statements**
4. **#Other statements**

### Example 1

1. n = int(input("Enter the number of rows you want to print?"))
2. i,j=0,0
3. **for** i **in** range(0,n):
4.   **print()**
5.   **for** j **in** range(0,i+1):
6.     **print("\*",end="")**

#### Output:

```
Enter the number of rows you want to print?
*
**
***
****
*****
```

## Using else statement with for loop

Unlike other languages like C, C++, or Java, python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

## Example 1

1. **for** i **in** range(0,5):
2.     **print**(i)
3. **else:****print**("for loop completely exhausted, since there is no break.");

In the above example, for loop is executed completely since there is no break statement in the loop. The control comes out of the loop and hence the else block is executed.

### Output:

```
0  
1  
2  
3  
4
```

for loop completely exhausted, since there is no break.

## Example 2

1. **for** i **in** range(0,5):
2.     **print**(i)
3.     **break**;
4. **else:****print**("for loop is exhausted");
5. **print**("The loop is broken due to break statement...came out of loop")

In the above example, the loop is broken due to break statement therefore the else statement will not be executed. The statement present immediate next to else block will be executed.

#### **Output:**

```
0
```

The loop is broken due to break statement...came out of loop

## Python while loop

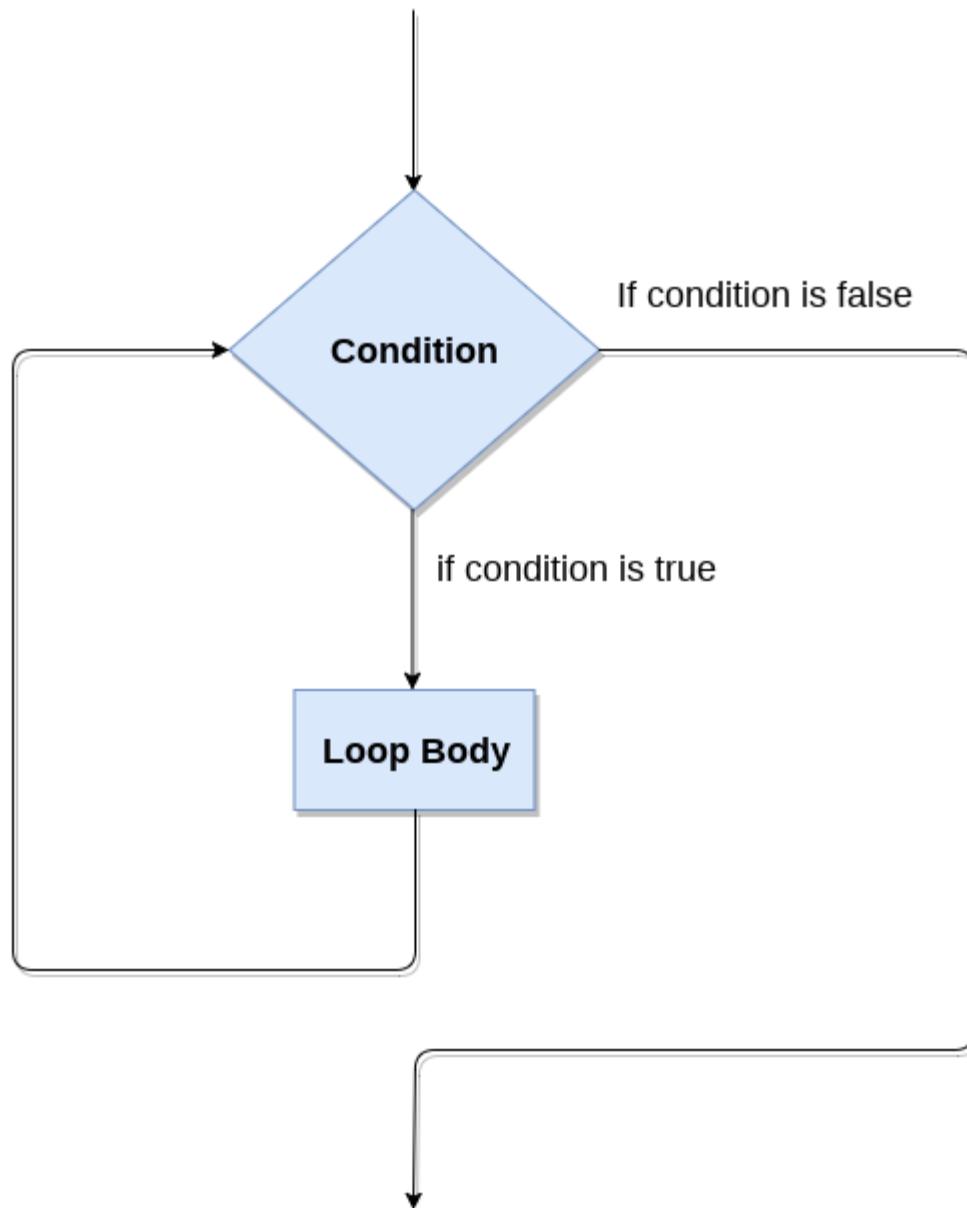
The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.

It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax is given below.

1. **while** expression:
2. statements

Here, the statements can be a single statement or the group of statements. The expression should be any valid python expression resulting into true or false. The true is any non-zero value.



## Example 1

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

### Output:

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

## Example 2

```
1. i=1
2. number=0
3. b=9
4. number = int(input("Enter the number?"))
5. while i<=10:
6.     print("%d X %d = %d \n"%(number,i,number*i));
7.     i = i+1;
```

### Output:

```
Enter the number?10
```

```
10 X 1 = 10
```

```
10 X 2 = 20
```

```
10 X 3 = 30
```

```
10 X 4 = 40
```

```
10 X 5 = 50
```

```
10 X 6 = 60
```

```
10 X 7 = 70
```

```
10 X 8 = 80
```

```
10 X 9 = 90
```

```
10 X 10 = 100
```

---

## Infinite while loop

If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

### Example 1

1. **while** (1):
2.     **print**("Hi! we are inside the infinite while loop");

### **Output:**

```
Hi! we are inside the infinite while loop  
(infinite times)
```

## Example 2

1. var = 1
2. **while** var != 2:
3.     i = int(input("Enter the number?"))
4.     **print** ("Entered value is %d"%(i))

### **Output:**

```
Enter the number?102  
Entered value is 102  
  
fdas  
Enter the number?102  
Entered value is 102  
Enter the number?103  
Entered value is 103  
Enter the number?103  
(infinite loop)
```

## Using else with Python while loop

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Consider the following example.

```
1. i=1;
2. while i<=5:
3.     print(i)
4.     i=i+1;
5. else:print("The while loop exhausted");
```

**Output:**

```
1
2
3
4
5
The while loop exhausted
```

## Example 2

```
1. i=1;
2. while i<=5:
3.     print(i)
4.     i=i+1;
5.     if(i==3):
6.         break;
7. else:print("The while loop exhausted");
```

**Output:**

```
1
2
```

## Python break statement

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

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

The syntax of the break is given below.

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

## Example 1

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

### Output:

```
item matched
found at 2 location
```

## Example 2

```
1. str = "python"
```

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

**Output:**

```
p  
y  
t  
h
```

### Example 3: break statement with while loop

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

**Output:**

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

### Example 3

```
1. n=2  
2. while 1:
```

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

**Output:**

```
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
Do you want to continue printing the table, press 0 for no?1
3 X 1 = 3
3 X 2 = 6
```

```
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
Do you want to continue printing the table, press 0 for no?0
```

## Python continue Statement

The continue statement in python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

The syntax of Python continue statement is given below.

1. #loop statements
2. **continue;**
3. #the code to be skipped

### Example 1

1. i = 0;
2. **while** i!=10:

```
3.     print("%d"%i);
4.     continue;
5.     i=i+1;
```

**Output:**

```
infinite loop
```

## Example 2

```
1. i=1; #initializing a local variable
2. #starting a loop from 1 to 10
3. for i in range(1,11):
4.     if i==5:
5.         continue;
6.     print("%d"%i);
```

**Output:**

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

---

## Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

Pass is also used where the code will be written somewhere but not yet written in the program file.

The syntax of the pass statement is given below.

## Example

```
1. list = [1,2,3,4,5]
2. flag = 0
3. for i in list:
4.     print("Current element:",i,end=" ")
5.     if i==3:
6.         pass;
7.         print("\nWe are inside pass block\n");
8.         flag = 1;
9.     if flag==1:
10.        print("\nCame out of pass\n");
11.        flag=0;
```

### Output:

```
Current element: 1 Current element: 2 Current element: 3
We are inside pass block
```

```
Came out of pass
```

```
Current element: 4 Current element: 5
```

## Python Pass

In Python, pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

### Python Pass Syntax

1. **pass**

### Python Pass Example

1. **for i in** [1,2,3,4,5]:
2.     **if** i==3:
3.         **pass**
4.         **print** "Pass when value is",i
5.     **print** i,

### Output:

1. >>>
2. 1 2 Pass when value **is** 3
3. 3 4 5
4. >>>

[next](#) → ← [prev](#)

## Python String

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

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

Consider the following example in python to create a string.

1. str = "Hi Python !"

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

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

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

## Strings indexing and splitting

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

```
str = "HELLO"
```

H	E	L	L	O
0	1	2	3	4

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

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

```
str = "HELLO"
```

H	E	L	L	O
0	1	2	3	4

str[0] = 'H'      str[:] = 'HELLO'

str[1] = 'E'      str[0:] = 'HELLO'

str[2] = 'L'      str[:5] = 'HELLO'

str[3] = 'L'      str[:3] = 'HEL'

str[4] = 'O'      str[0:2] = 'HE'

str[1:4] = 'ELL'

Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'python' is given, then str[1:3] will always include str[1] = 'p', str[2] = 'y', str[3] = 't' and nothing else.

## Reassigning strings

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

Consider the following example.

### Example 1

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

#### Output:

```
Traceback (most recent call last):
  File "12.py", line 2, in <module>
    str[0] = "h";
TypeError: 'str' object does not support item assignment
```

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

### Example 2

1. str = "HELLO"
2. **print**(str)

3. str = "hello"
4. **print**(str)

**Output:**

```
HELLO
hello
```

## String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
In	It is known as membership operator. It returns if a particular sub-string is present in the specified string.

not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

## Example

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

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

## Output:

```
HelloHelloHello
Hello world
o
11
False
False
C://python37
The string str : Hello
```

## Python Formatting operator

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

Consider the following example.

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

### Output:

```
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
```

```
Hi I am string ... My value is Ayush
```

## Built-in String functions

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

Method	Description
<a href="#"><u>capitalize()</u></a>	It capitalizes the first character of the String. This function is deprecated in python3
<a href="#"><u>casefold()</u></a>	It returns a version of s suitable for case-less comparisons.
<a href="#"><u>center(width ,fillchar)</u></a>	It returns a space padded string with the original string centred with equal number of left and right spaces.
<a href="#"><u>count(string,begin,end)</u></a>	It counts the number of occurrences of a substring in a String between begin and end index.
<code>decode(encoding = 'UTF8', errors = 'strict')</code>	Decodes the string using codec registered for encoding.

<a href="#"><u>encode()</u></a>	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
<a href="#"><u>endswith(suffix ,begin=0,end=len(string))</u></a>	It returns a Boolean value if the string terminates with given suffix between begin and end.
<a href="#"><u>expandtabs(tabsize = 8)</u></a>	It defines tabs in string to multiple spaces. The default space value is 8.
<a href="#"><u>find(substring ,beginIndex, endIndex)</u></a>	It returns the index value of the string where substring is found between begin index and end index.
<a href="#"><u>format(value)</u></a>	It returns a formatted version of S, using the passed value.
<a href="#"><u>index(subsring, beginIndex, endIndex)</u></a>	It throws an exception if string is not found. It works same as find() method.
<a href="#"><u>isalnum()</u></a>	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
<a href="#"><u>isalpha()</u></a>	It returns true if all the characters are alphabets and there is at least one character, otherwise False.

<u><a href="#">isdecimal()</a></u>	It returns true if all the characters of the string are decimals.
<u><a href="#">isdigit()</a></u>	It returns true if all the characters are digits and there is at least one character, otherwise False.
<u><a href="#">isidentifier()</a></u>	It returns true if the string is the valid identifier.
<u><a href="#">islower()</a></u>	It returns true if the characters of a string are in lower case, otherwise false.
<u><a href="#">isnumeric()</a></u>	It returns true if the string contains only numeric characters.
<u><a href="#">isprintable()</a></u>	It returns true if all the characters of s are printable or s is empty, false otherwise.
<u><a href="#">isupper()</a></u>	It returns false if characters of a string are in Upper case, otherwise False.
<u><a href="#">isspace()</a></u>	It returns true if the characters of a string are white-space, otherwise false.
<u><a href="#">istitle()</a></u>	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case.

<u><a href="#">isupper()</a></u>	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
<u><a href="#">join(seq)</a></u>	It merges the strings representation of the given sequence.
<u><a href="#">len(string)</a></u>	It returns the length of a string.
<u><a href="#">ljust(width[,fillchar])</a></u>	It returns the space padded strings with the original string left justified to the given width.
<u><a href="#">lower()</a></u>	It converts all the characters of a string to Lower case.
<u><a href="#">lstrip()</a></u>	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.
<u><a href="#">partition()</a></u>	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
<u><a href="#">maketrans()</a></u>	It returns a translation table to be used in translate function.
<u><a href="#">replace(old,new[,count])</a></u>	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.

<u><a href="#">rfind(str,beg=0,end=len(str))</a></u>	It is similar to find but it traverses the string in backward direction.
<u><a href="#">rindex(str,beg=0,end=len(str))</a></u>	It is same as index but it traverses the string in backward direction.
<u><a href="#">rjust(width,[,fillchar])</a></u>	Returns a space padded string having original string right justified to the number of characters specified.
<u><a href="#">rstrip()</a></u>	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.
<u><a href="#">rsplit(sep=None, maxsplit = -1)</a></u>	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
<u><a href="#">split(str,num=string.count(str))</a></u>	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.
<u><a href="#">splitlines(num=string.count('\n'))</a></u>	It returns the list of strings at each line with newline removed.
<u><a href="#">startswith(str,beg=0,end=len(str))</a></u>	It returns a Boolean value if the string starts with given str between begin and end.

<code>strip([chars])</code>	It is used to perform lstrip() and rstrip() on the string.
<u><a href="#">swapcase()</a></u>	It inverts case of all characters in a string.
<code>title()</code>	It is used to convert the string into the title-case i.e., The string <b>meEruT</b> will be converted to Meerut.
<u><a href="#">translate(table,deletechars = "")</a></u>	It translates the string according to the translation table passed in the function .
<u><a href="#">upper()</a></u>	It converts all the characters of a string to Upper Case.
<u><a href="#">zfill(width)</a></u>	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
<u><a href="#">rpartition()</a></u>	

## Python List

List in python is implemented to store the sequence of various type of data. However, python contains six data types that are capable to store the sequences but the most common and reliable type is list.

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

A list can be defined as follows.

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]
3. L3 = [1, "Ryan"]

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

Lets consider a proper example to define a list and printing its values.

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

#### **Output:**

```
printing employee data...
Name : John, ID: 102, Country: USA
printing departments...
Department 1:
Name: CS, ID: 11
```

```
Department 2:  
Name: IT, ID: 11  
HOD Details ....  
CS HOD Name: Mr. Holding, Id: 10  
IT HOD Name: Mr. Bewon, Id: 11  
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>
```

## List indexing and splitting

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

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

Consider the following example.

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

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

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

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

List[2] = 2                  List[2:4] = [2, 3]

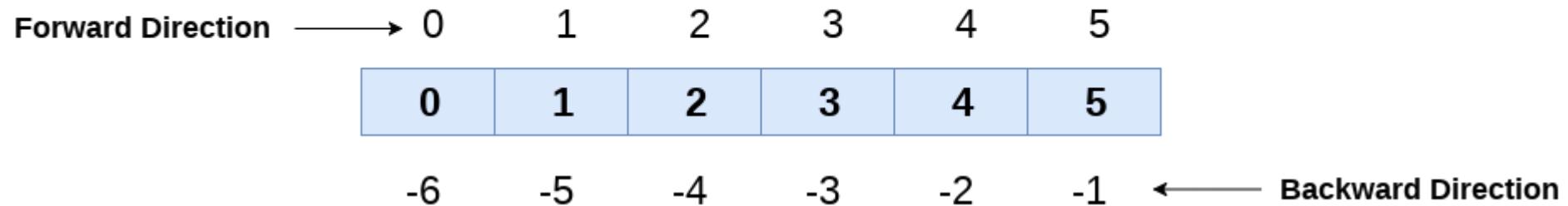
List[3] = 3                  List[1:3] = [1, 2]

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

**List[5] = 5**

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

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



## Updating List values

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

Python also provide us the `append()` method which can be used to add values to the string.

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

1. `List = [1, 2, 3, 4, 5, 6]`
2. `print(List)`
3. `List[2] = 10;`
4. `print(List)`
5. `List[1:3] = [89, 78]`
6. `print(List)`

### Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
```

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

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

Consider the following example to delete the list elements.

1. List = [0,1,2,3,4]
2. **print**(List)
3. **del** List[0]
4. **print**(List)
5. **del** List[3]
6. **print**(List)

#### Output:

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3]
```

## Python List Operations

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

Lets see how the list responds to various operators.

1. Consider a List l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8]

Operator	Description	Example

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

## Iterating a List

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

1. `List = ["John", "David", "James", "Jonathan"]`
2. `for i in List:` #i will iterate over the elements of the List and contains each element in each iteration.

```
3. print(i);
```

**Output:**

```
John  
David  
James  
Jonathan
```

## Adding elements to the list

Python provides append() function by using which we can add an element to the list. However, the append() method can only add the value to the end of the list.

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

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

**Output:**

```
Enter the number of elements in the list 5  
Enter the item?1  
Enter the item?2  
Enter the item?3  
Enter the item?4  
Enter the item?5
```

```
printing the list items....  
1 2 3 4 5
```

## Removing elements from the list

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

### Output:

```
printing original list:  
0 1 2 3 4  
printing the list after the removal of first element...  
1 2 3 4
```

## Python List Built-in functions

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

SN	Function	Description
1	cmp(list1, list2)	It compares the elements of both the lists.

2	<code>len(list)</code>	It is used to calculate the length of the list.
3	<code>max(list)</code>	It returns the maximum element of the list.
4	<code>min(list)</code>	It returns the minimum element of the list.
5	<code>list(seq)</code>	It converts any sequence to the list.

## Python List built-in methods

SN	Function	Description
1	<a href="#"><u>list.append(obj)</u></a>	The element represented by the object obj is added to the list.
2	<a href="#"><u>list.clear()</u></a>	It removes all the elements from the list.
3	<a href="#"><u>List.copy()</u></a>	It returns a shallow copy of the list.
4	<a href="#"><u>list.count(obj)</u></a>	It returns the number of occurrences of the specified object in the list.
5	<a href="#"><u>list.extend(seq)</u></a>	The sequence represented by the object seq is extended to the list.

6	<u><a href="#">list.index(obj)</a></u>	It returns the lowest index in the list that object appears.
7	<u><a href="#">list.insert(index, obj)</a></u>	The object is inserted into the list at the specified index.
8	<u><a href="#">list.pop(obj=list[-1])</a></u>	It removes and returns the last object of the list.
9	<u><a href="#">list.remove(obj)</a></u>	It removes the specified object from the list.
10	<u><a href="#">list.reverse()</a></u>	It reverses the list.
11	<u><a href="#">list.sort([func])</a></u>	It sorts the list by using the specified compare function if given.

## Python Tuple

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

A tuple can be written as the collection of comma-separated values enclosed with the small brackets. A tuple can be defined as follows.

1. T1 = (101, "Ayush", 22)
2. T2 = ("Apple", "Banana", "Orange")

## Example

```
1. tuple1 = (10, 20, 30, 40, 50, 60)
2. print(tuple1)
3. count = 0
4. for i in tuple1:
5.     print("tuple1[%d] = %d"%(count, i));
```

**Output:**

```
(10, 20, 30, 40, 50, 60)
tuple1[0] = 10
tuple1[0] = 20
tuple1[0] = 30
tuple1[0] = 40
tuple1[0] = 50
tuple1[0] = 60
```

## Example 2

```
1. tuple1 = tuple(input("Enter the tuple elements ..."))
2. print(tuple1)
3. count = 0
4. for i in tuple1:
5.     print("tuple1[%d] = %s"%(count, i));
```

**Output:**

```
Enter the tuple elements ...12345
('1', '2', '3', '4', '5')
tuple1[0] = 1
tuple1[0] = 2
tuple1[0] = 3
tuple1[0] = 4
tuple1[0] = 5
```

However, if we try to reassign the items of a tuple, we would get an error as the tuple object doesn't support the item assignment.

An empty tuple can be written as follows.

1. T3 = ()

The tuple having a single value must include a comma as given below.

1. T4 = (90,)

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

We will see all these aspects of tuple in this section of the tutorial.

## Tuple indexing and splitting

The indexing and slicing in tuple are similar to lists. The indexing in the tuple starts from 0 and goes to `length(tuple) - 1`.

The items in the tuple can be accessed by using the slice operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

**Tuple = ( 0, 1, 2, 3, 4, 5 )**

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
----------	----------	----------	----------	----------	----------

**Tuple[0] = 0**      **Tuple[0:] = (0, 1, 2, 3, 4, 5)**

**Tuple[1] = 1**      **Tuple[:] = (0, 1, 2, 3, 4, 5)**

**Tuple[2] = 2**      **Tuple[2:4] = (2, 3)**

**Tuple[3] = 3**      **Tuple[1:3] = (1, 2)**

**Tuple[4] = 4**      **Tuple[:4] = (0, 1, 2, 3)**

**Tuple[5] = 5**

Unlike lists, the tuple items can not be deleted by using the `del` keyword as tuples are immutable. To delete an entire tuple, we can use the `del` keyword with the tuple name.

Consider the following example.

1. `tuple1 = (1, 2, 3, 4, 5, 6)`
2. `print(tuple1)`
3. `del tuple1[0]`

4. **print**(tuple1)
5. **del** tuple1
6. **print**(tuple1)

**Output:**

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

Like lists, the tuple elements can be accessed in both the directions. The right most element (last) of the tuple can be accessed by using the index -1. The elements from left to right are traversed using the negative indexing.

Consider the following example.

1. tuple1 = (1, 2, 3, 4, 5)
2. **print**(tuple1[-1])
3. **print**(tuple1[-4])

**Output:**

```
5
2
```

## Basic Tuple operations

The operators like concatenation (+), repetition (\*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	<code>T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)</code>
Concatenation	It concatenates the tuple mentioned on either side of the operator.	<code>T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)</code>
Membership	It returns true if a particular item exists in the tuple otherwise false.	<code>print (2 in T1) prints True.</code>
Iteration	The for loop is used to iterate over the tuple elements.	<pre>for i in T1:     print(i)</pre> <p><b>Output</b></p> <pre>1 2 3 4 5</pre>
Length	It is used to get the length of the tuple.	<code>len(T1) = 5</code>

## Python Tuple inbuilt functions

<b>SN</b>	<b>Function</b>	<b>Description</b>
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

## Where use tuple

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
  2. Tuple can simulate dictionary without keys. Consider the following nested structure which can be used as a dictionary.
1. `[(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]`
  3. Tuple can be used as the key inside dictionary due to its immutable nature.

## List VS Tuple

SN	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ()..
2	The List is mutable.	The tuple is immutable.
3	The List has the variable length.	The tuple has the fixed length.
4	The list provides more functionality than tuple.	The tuple provides less functionality than the list.
5	The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary.

## Nesting List and tuple

We can store list inside tuple or tuple inside the list up to any number of level.

Lets see an example of how can we store the tuple inside the list.

1. Employees = [(101, "Ayush", 22), (102, "john", 29), (103, "james", 45), (104, "Ben", 34)]
2. `print("----Printing list----");`
3. `for i in Employees:`
4.     `print(i)`
5. `Employees[0] = (110, "David",22)`

6. `print();`
7. `print("----Printing list after modification----");`
8. `for i in Employees:`
9.     `print(i)`

#### **Output:**

```
----Printing list----  
(101, 'Ayush', 22)  
(102, 'john', 29)  
(103, 'james', 45)  
(104, 'Ben', 34)  
  
----Printing list after modification----  
  
(110, 'David', 22)  
(102, 'john', 29)  
(103, 'james', 45)  
(104, 'Ben', 34)
```

## Python Set

The set in python can be defined as the unordered collection of various items enclosed within the curly braces. The elements of the set can not be duplicate. The elements of the python set must be immutable.

Unlike other collections in python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together or we can get the list of elements by looping through the set.

## Creating a set

The set can be created by enclosing the comma separated items with the curly braces. Python also provides the set method which can be used to create the set by the passed sequence.

## Example 1: using curly braces

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ...")
5. **for i in** Days:
6.     **print**(i)

### Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}
<class 'set'>
looping through the set elements ...
Friday
Tuesday
Monday
Saturday
Thursday
Sunday
Wednesday
```

## Example 2: using set() method

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ...")
5. **for i in** Days:
6.     **print**(i)

### Output:

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}  
<class 'set'>  
looping through the set elements ...  
Friday  
Wednesday  
Thursday  
Saturday  
Monday  
Tuesday  
Sunday
```

---

## Python Set operations

In the previous example, we have discussed about how the set is created in python. However, we can perform various mathematical operations on python sets like union, intersection, difference, etc.

### Adding items to the set

Python provides the `add()` method which can be used to add some particular item to the set. Consider the following example.

#### Example:

1. `Months = set(["January","February", "March", "April", "May", "June"])`
2. `print("\nprinting the original set ... ")`
3. `print(Months)`
4. `print("\nAdding other months to the set...");`
5. `Months.add("July");`
6. `Months.add("August");`
7. `print("\nPrinting the modified set...");`
8. `print(Months)`
9. `print("\nlooping through the set elements ... ")`

```
10. for i in Months:
```

```
11.   print(i)
```

#### Output:

```
printing the original set ...
{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...
February
July
May
April
March
August
June
January
```

To add more than one item in the set, Python provides the **update()** method.

Consider the following example.

## Example

1. Months = set(["January", "February", "March", "April", "May", "June"])
2. **print("\nprinting the original set ... ")**
3. **print(Months)**
4. **print("\nupdating the original set ... ")**
5. Months.update(["July", "August", "September", "October"]);

6. `print("\nprinting the modified set ... ")`
7. `print(Months);`

#### **Output:**

```
printing the original set ...
{'January', 'February', 'April', 'May', 'June', 'March'}
```

```
updating the original set ...
```

```
printing the modified set ...
{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}
```

## Removing items from the set

Python provides **discard()** method which can be used to remove the items from the set.

Consider the following example.

### Example

1. `Months = set(["January","February", "March", "April", "May", "June"])`
2. `print("\nprinting the original set ... ")`
3. `print(Months)`
4. `print("\nRemoving some months from the set...");`
5. `Months.discard("January");`
6. `Months.discard("May");`
7. `print("\nPrinting the modified set...");`
8. `print(Months)`
9. `print("\nlooping through the set elements ... ")`
10. `for i in Months:`

## 11. `print(i)`

### Output:

```
printing the original set ...
{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...
{'February', 'March', 'April', 'June'}

looping through the set elements ...
February
March
April
June
```

Python also provide the `remove()` method to remove the items from the set. Consider the following example to remove the items using `remove()` method.

### Example

1. Months = set(["January","February", "March", "April", "May", "June"])
2. `print("\nprinting the original set ... ")`
3. `print(Months)`
4. `print("\nRemoving some months from the set...");`
5. `Months.remove("January");`
6. `Months.remove("May");`
7. `print("\nPrinting the modified set...");`
8. `print(Months)`

### Output:

```
printing the original set ...
{'February', 'June', 'April', 'May', 'January', 'March'}
```

Removing some months from the set...

```
Printing the modified set...
{'February', 'June', 'April', 'March'}
```

We can also use the `pop()` method to remove the item. However, this method will always remove the last item.

Consider the following example to remove the last item from the set.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving some months from the set...");
5. Months.pop();
6. Months.pop();
7. **print**("\nPrinting the modified set...");
8. **print**(Months)

#### **Output:**

```
printing the original set ...
{'June', 'January', 'May', 'April', 'February', 'March'}
```

Removing some months from the set...

```
Printing the modified set...
{'May', 'April', 'February', 'March'}
```

Python provides the `clear()` method to remove all the items from the set.

Consider the following example.

1. Months = set(["January", "February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving all the items from the set...");
5. Months.clear()
6. **print**("\nPrinting the modified set...")
7. **print**(Months)

#### **Output:**

```
printing the original set ...
{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...
set()
```

---

## Difference between discard() and remove()

Despite the fact that discard() and remove() method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the python will give the error.

Consider the following example.

### Example

```
1. Months = set(["January", "February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nRemoving items through discard() method...");
5. Months.discard("Feb"); #will not give an error although the key feb is not available in the set
6. print("\nprinting the modified set...")
7. print(Months)
8. print("\nRemoving items through remove() method...");
9. Months.remove("Jan") #will give an error as the key jan is not available in the set.
10. print("\nPrinting the modified set...")
11. print(Months)
```

#### Output:

```
printing the original set ...
{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...
{'March', 'January', 'April', 'June', 'February', 'May'}

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

## Union of two Sets

The union of two sets are calculated by using the or (|) operator. The union of the two sets contains the all the items that are present in both the sets.

Consider the following example to calculate the union of two sets.

## Example 1 : using union | operator

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

### Output:

```
{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}
```

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

## Example 2: using union() method

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

### Output:

```
{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}
```

## Intersection of two sets

The & (intersection) operator is used to calculate the intersection of the two sets in python. The intersection of the two sets are given as the set of the elements that common in both sets.

Consider the following example.

## Example 1: using & operator

1. set1 = {"Ayush", "John", "David", "Martin"}
2. set2 = {"Steve", "Milan", "David", "Martin"}
3. **print**(set1&set2) #prints the intersection of the two sets

### Output:

```
{'Martin', 'David'}
```

## Example 2: using intersection() method

1. set1 = {"Ayush", "John", "David", "Martin"}
2. set2 = {"Steve", "Milan", "David", "Martin"}
3. **print**(set1.intersection(set2)) #prints the intersection of the two sets

### Output:

```
{'Martin', 'David'}
```

## The intersection\_update() method

The intersection\_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The Intersection\_update() method is different from intersection() method since it modifies the original set by removing the unwanted items, on the other hand, intersection() method returns a new set.

Consider the following example.

1. a = {"ayush", "bob", "castle"}

2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
- 4.
5. a.intersection\_update(b, c)
- 6.
7. **print(a)**

**Output:**

```
{'castle'}
```

## Difference of two sets

The difference of two sets can be calculated by using the subtraction (-) operator. The resulting set will be obtained by removing all the elements from set 1 that are present in set 2.

Consider the following example.

### Example 1 : using subtraction ( - ) operator

1. Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print(Days1-Days2)** #{"Wednesday", "Thursday" will be printed}

**Output:**

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

### Example 2 : using difference() method

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

**Output:**

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

---

## Set comparisons

Python allows us to use the comparison operators i.e., `<`, `>`, `<=`, `>=`, `==` with the sets by using which we can check whether a set is subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

Consider the following example.

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

### **Output:**

```
True  
False  
False
```

---

## FrozenSets

The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set can not be changed and therefore it can be used as a key in dictionary.

The elements of the frozen set can not be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

1. `Frozenset = frozenset([1,2,3,4,5])`
2. `print(type(Frozenset))`
3. `print("\nprinting the content of frozen set...")`
4. `for i in Frozenset:`
5.   `print(i);`
6. `Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation`

### **Output:**

```
<class 'frozenset'>  
  
printing the content of frozen set...  
1
```

```
2
3
4
5
Traceback (most recent call last):
  File "set.py", line 6, in <module>
    Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation
AttributeError: 'frozenset' object has no attribute 'add'
```

## Frozenset for the dictionary

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Consider the following example.

1. Dictionary = {"Name":"John", "Country":"USA", "ID":101}
2. **print**(type(Dictionary))
3. Frozenset = frozenset(Dictionary); *#Frozenset will contain the keys of the dictionary*
4. **print**(type(Frozenset))
5. **for i in** Frozenset:
6.     **print**(i)

### Output:

```
<class 'dict'>
<class 'frozenset'>
Name
Country
ID
```

---

## Python Built-in set methods

Python contains the following methods to be used with the sets.

SN	Method	Description
1	<u><a href="#">add(item)</a></u>	It adds an item to the set. It has no effect if the item is already present in the set.
2	clear()	It deletes all the items from the set.
3	copy()	It returns a shallow copy of the set.
4	difference_update(....)	It modifies this set by removing all the items that are also present in the specified sets.
5	<u><a href="#">discard(item)</a></u>	It removes the specified item from the set.
6	intersection()	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).
7	intersection_update(....)	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

8	<code>Isdisjoint(...)</code>	Return True if two sets have a null intersection.
9	<code>Issubset(...)</code>	Report whether another set contains this set.
10	<code>Issuperset(...)</code>	Report whether this set contains another set.
11	<code>pop()</code>	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	<code>remove(item)</code>	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
13	<code>symmetric_difference(...)</code>	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
14	<code>symmetric_difference_update(...)</code>	Update a set with the symmetric difference of itself and another.
15	<code>union(...)</code>	Return the union of sets as a new set. (i.e. all elements that are in either set.)

16

update()

Update a set with the union of itself and others.

## Python Dictionary

Dictionary is used to implement the key-value pair in python. The dictionary is the data type in python which can simulate the real-life data arrangement where some specific value exists for some particular key.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any python object whereas the keys are the immutable python object, i.e., Numbers, string or tuple.

Dictionary simulates Java map in python.hash-

### Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:). The collections of the key-value pairs are enclosed within the curly braces {}.

The syntax to define the dictionary is given below.

1. Dict = {"Name": "Ayush", "Age": 22}

In the above dictionary **Dict**, The keys **Name**, and **Age** are the string that is an immutable object.

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

1. Employee = {"Name": "John", "Age": 29, "salary": 25000, "Company": "GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

## Output

```
<class 'dict'>
printing Employee data ....
{'Age': 29, 'salary': 25000, 'Name': 'John', 'Company': 'GOOGLE'}
```

## Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**("Name : %s" %Employee["Name"])
5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

## Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

## Updating dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys.

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. **print**("printing the new data");
11. **print**(Employee)

### Output:

```
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'salary': 25000, 'Company': 'GOOGLE', 'Age': 29}
Enter the details of the new employee....
Name: David
Age: 19
Salary: 8900
Company:JTP
printing the new data
```

```
{'Name': 'David', 'salary': 8900, 'Company': 'JTP', 'Age': 19}
```

## Deleting elements using del keyword

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. `print(type(Employee))`
3. `print("printing Employee data .... ")`
4. `print(Employee)`
5. `print("Deleting some of the employee data")`
6. `del Employee["Name"]`
7. `del Employee["Company"]`
8. `print("printing the modified information ")`
9. `print(Employee)`
10. `print("Deleting the dictionary: Employee");`
11. `del Employee`
12. `print("Lets try to print it again ");`
13. `print(Employee)`

### Output:

```
<class 'dict'>
printing Employee data ....
{'Age': 29, 'Company': 'GOOGLE', 'Name': 'John', 'salary': 25000}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
Traceback (most recent call last):
  File "list.py", line 13, in <module>
```

```
    print(Employee)
NameError: name 'Employee' is not defined
```

## Iterating Dictionary

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

### Example 1

# for loop to print all the keys of a dictionary

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3.     **print**(x);

**Output:**

```
Name
Company
salary
Age
```

### Example 2

#for loop to print all the values of the dictionary

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3.     **print**(Employee[x]);

**Output:**

```
29
GOOGLE
John
25000
```

## Example 3

#for loop to print the values of the dictionary by using values() method.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.values():
3.     **print**(x);

**Output:**

```
GOOGLE
25000
John
29
```

## Example 4

#for loop to print the items of the dictionary by using items() method.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.items():
3.     **print**(x);

**Output:**

```
('Name', 'John')
('Age', 29)
```

```
('salary', 25000)
('Company', 'GOOGLE')
```

## Properties of Dictionary keys

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

Consider the following example.

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

### Output:

```
Salary 25000
Company GOOGLE
Name Johnn
Age 29
```

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuple as the key but we can not use any mutable object like the list as the key in the dictionary.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3.     **print**(x,y)

### Output:

```
Traceback (most recent call last):
  File "list.py", line 1, in 
    Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE", [100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

## Built-in Dictionary functions

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

SN	Function	Description
1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.

## Built-in Dictionary methods

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

<b>SN</b>	<b>Method</b>	<b>Description</b>
1	<u><a href="#">dic.clear()</a></u>	It is used to delete all the items of the dictionary.
2	<u><a href="#">dict.copy()</a></u>	It returns a shallow copy of the dictionary.
3	<u><a href="#">dict.fromkeys(iterable, value = None, /)</a></u>	Create a new dictionary from the iterable with the values equal to value.
4	<u><a href="#">dict.get(key, default = "None")</a></u>	It is used to get the value specified for the pass ed key.
5	<u><a href="#">dict.has_key(key)</a></u>	It returns true if the dictionary contains the specified key.
6	<u><a href="#">dict.items()</a></u>	It returns all the key-value pairs as a tuple.
7	<u><a href="#">dict.keys()</a></u>	It returns all the keys of the dictionary.
8	<u><a href="#">dict.setdefault(key,default= "None")</a></u>	It is used to set the key to the default value if the key is not specified in the dictionary
9	<u><a href="#">dict.update(dict2)</a></u>	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.

10	<a href="#"><u>dict.values()</u></a>	It returns all the values of the dictionary.
	<a href="#"><u>len()</u></a>	
	<a href="#"><u>popItem()</u></a>	
	<a href="#"><u>pop()</u></a>	
	<a href="#"><u>count()</u></a>	
	<a href="#"><u>index()</u></a>	

## Python Functions

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the python program.

In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

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

## Advantage of functions in python

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions.
- Reusability is the main achievement of python functions.
- However, Function calling is always overhead in a python program.

## Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

1. **def** my\_function():
2.     function-suite
3.     **return** <expression>

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

A function can accept any number of parameters that must be the same in the definition and function calling.

## Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

```
1. def hello_world():
2.     print("hello world")
3.
4. hello_world()
```

#### Output:

hello world

## Parameters in function

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Consider the following example which contains a function that accepts a string as the parameter and prints it.

## Example 1

```
1. #defining the function
2. def func (name):
3.     print("Hi ",name);
4.
5. #calling the function
6. func("Ayush")
```

## Example 2

```
1. #python function to calculate the sum of two variables
2. #defining the function
3. def sum (a,b):
```

```
4.     return a+b;
5.
6. #taking values from the user
7. a = int(input("Enter a: "))
8. b = int(input("Enter b: "))
9.
10.#printing the sum of a and b
11.print("Sum = ",sum(a,b))
```

#### Output:

```
Enter a: 10
Enter b: 20
Sum = 30
```

## Call by reference in Python

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

## Example 1 Passing Immutable Object (List)

```
1. #defining the function
2. Def change_list(l1):
3.     l1.add(20,30)
4.     Print(l1)
5.
```

```
6. #defining the list
7. list1 = [10,30,40,50]
8.
9. #calling the function
10.change_list(list1);
11.print("list outside function = ",list1);
```

**Output:**

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

## Example 2 Passing Mutable Object (String)

```
1. #defining the function
2. def change_string (str):
3.     str = str + " Hows you";
4.     print("printing the string inside function :",str);
5.
6. string1 = "Hi I am there"
7.
8. #calling the function
9. change_string(string1)
10.
11.print("printing the string outside function :",string1)
```

**Output:**

```
printing the string inside function : Hi I am there Hows you
printing the string outside function : Hi I am there
```

## Types of arguments

There may be several types of arguments which can be passed at the time of function calling.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

## Required Arguments

Till now, we have learned about function calling in python. However, we can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

### Example 1

1. #the argument name is the required argument to the function func
2. **def** func(name):
3.     message = "Hi "+name;
4.     **return** message;
5. name = **input**("Enter the name?")
6. **print**(func(name))

**Output:**

```
Enter the name?John  
Hi John
```

## Example 2

1. `#the function simple_interest accepts three arguments and returns the simple interest accordingly`
2. `def simple_interest(p,t,r):`
3.   `return (p*t*r)/100`
4. `p = float(input("Enter the principle amount? "))`
5. `r = float(input("Enter the rate of interest? "))`
6. `t = float(input("Enter the time in years? "))`
7. `print("Simple Interest: ",simple_interest(p,r,t))`

### Output:

```
Enter the principle amount? 10000  
Enter the rate of interest? 5  
Enter the time in years? 2  
Simple Interest: 1000.0
```

## Example 3

1. `#the function calculate returns the sum of two arguments a and b`
2. `def calculate(a,b):`
3.   `return a+b`
4. `calculate(10) # this causes an error as we are missing a required arguments b.`

### Output:

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

## Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

### Example 1

1. #function func is called with the name and message as the keyword arguments
2. **def** func(name,message):
3.     **print**("printing the message with",name,"and ",message)
4. func(name = "John",message="hello") #name and message is copied with the values John and hello respectively

#### Output:

```
printing the message with John and hello
```

### Example 2 providing the values in different order at the calling

1. #The function simple\_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case
2. **def** simple\_interest(p,t,r):
3.     **return** (p\*t\*r)/100
4. **print**("Simple Interest: ",simple\_interest(t=10,r=10,p=1900))

#### Output:

```
Simple Interest: 1900.0
```

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

### Example 3

1. #The function simple\_interest(p, t, r) is called with the keyword arguments.
2. `def simple_interest(p,t,r):`
3.   `return (p*t*r)/100`
- 4.
5. `print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900))` # doesn't find the exact match of the name of the arguments (keywords)

#### Output:

```
TypeError: simple_interest() got an unexpected keyword argument 'time'
```

The python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

### Example 4

1. `def func(name1,message,name2):`
2.   `print("printing the message with",name1,",",message,",and",name2)`
3. `func("John",message="hello",name2="David")` #the first argument is not the keyword argument

#### Output:

```
printing the message with John , hello ,and David
```

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

## Example 5

1. `def func(name1,message,name2):`
2.   `print("printing the message with",name1, " ",message, "and",name2)`
3. `func("John",message="hello","David")`

### Output:

```
SyntaxError: positional argument follows keyword argument
```

## Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

## Example 1

1. `def printme(name,age=22):`
2.   `print("My name is",name,"and age is",age)`
3. `printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function`

### Output:

```
My name is john and age is 22
```

## Example 2

1. `def printme(name,age=22):`
2.   `print("My name is",name,"and age is",age)`
3. `printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function`
4. `printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age`

### Output:

```
My name is john and age is 22
My name is David and age is 10
```

## Variable length Arguments

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with \* (star) as \*<variable - name >.

Consider the following example.

## Example

1. `def printme(*names):`
2.   `print("type of passed argument is ",type(names))`
3.   `print("printing the passed arguments...")`
4.   `for name in names:`
5.       `print(name)`
6. `printme("john","David","smith","nick")`

### **Output:**

```
type of passed argument is
printing the passed arguments...
john
David
smith
nick
```

## Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

### Example 1

1. **def** print\_message():
2.     message = "hello !! I am going to print a message." # the variable message is local to the function itself
3.     **print**(message)
4. **print\_message()**

5. `print(message) # this will cause an error since a local variable cannot be accessible here.`

**Output:**

```
hello !! I am going to print a message.  
File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in  
    print(message)  
NameError: name 'message' is not defined
```

## Example 2

1. `def calculate(*args):`
2.     `sum=0`
3.     `for arg in args:`
4.         `sum = sum +arg`
5.     `print("The sum is",sum)`
6. `sum=0`
7. `calculate(10,20,30) #60 will be printed as the sum`
8. `print("Value of sum outside the function:",sum) # 0 will be printed`

**Output:**

```
The sum is 60  
Value of sum outside the function: 0
```

## Python Lambda Functions

Python allows us to not declare the function in the standard manner, i.e., by using the `def` keyword. Rather, the anonymous functions are declared by using `lambda` keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an Anonymous function is given below.

1. **lambda** arguments : expression

## Example 1

1. `x = lambda a:a+10 # a is an argument and a+10 is an expression which got evaluated and returned.`
2. `print("sum = ",x(20))`

**Output:**

```
sum = 30
```

## Example 2

Multiple arguments to Lambda function

1. `x = lambda a,b:a+b # a and b are the arguments and a+b is the expression which gets evaluated and returned.`
2. `print("sum = ",x(20,10))`

**Output:**

```
sum = 30
```

## Why use lambda functions?

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In python, the lambda function can be used as an argument to the higher order functions as arguments. Lambda functions are also used in the scenario where we need a Consider the following example.

## Example 1

1. `#the function table(n) prints the table of n`
2. `def table(n):`
3.   `return lambda a:a*n; # a will contain the iteration variable i and a multiple of n is returned at each function call`
4. `n = int(input("Enter the number?"))`
5. `b = table(n) #the entered number is passed into the function table. b will contain a lambda function which is called again and again with the iteration variable i`
6. `for i in range(1,11):`
7.   `print(n,"X",i,"=",b(i)); #the lambda function b is called with the iteration variable i,`

### Output:

```
Enter the number?10
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

## Example 2

Use of lambda function with filter

1. `#program to filter out the list which contains odd numbers`
2. `List = {1,2,3,4,10,123,22}`
3. `Oddlist = list(filter(lambda x:(x%3 == 0),List)) # the list contains all the items of the list for which the lambda function evaluates to true`
4. `print(Oddlist)`

**Output:**

```
[3, 123]
```

## Example 3

Use of lambda function with map

1. `#program to triple each number of the list using map`
2. `List = {1,2,3,4,10,123,22}`
3. `new_list = list(map(lambda x:x*3,List)) # this will return the triple of each item of the list and add it to new_list`
4. `print(new_list)`

**Output:**

```
[3, 6, 9, 12, 30, 66, 369]
```

## Python File Handling

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling.

In this section of the tutorial, we will learn all about file handling in python including, creating a file, opening a file, closing a file, writing and appending the file, etc.

## Opening a file

Python provides the `open()` function which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The syntax to use the `open()` function is given below.

1. `file object = open(<file-name>, <access-mode>, <buffering>)`

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

<b>SN</b>	<b>Access mode</b>	<b>Description</b>
1	r	It opens the file to read-only. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.

4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.

11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

## Example

1. #opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
- 3.
4. if fileptr:
5.     print("file is opened successfully")

### Output:

```
<class '_io.TextIOWrapper'>
file is opened successfully
```

## The close() method

Once all the operations are done on the file, we must close it through our python script using the close() method. Any unwritten information gets destroyed once the close() method is called on a file object.

We can perform any operation on the file externally in the file system is the file is opened in python, hence it is good practice to close the file once all the operations are done.

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

1. fileobject.close()

Consider the following example.

## Example

1. **# opens the file file.txt in read mode**
2. fileptr = open("file.txt","r")
- 3.
4. **if** fileptr:
5.     **print**("file is opened successfully")
- 6.
7. **#closes the opened file**
8. fileptr.close()

## Reading the file

To read a file using the python script, the python provides us the read() method. The read() method reads a string from the file. It can read the data in the text as well as binary format.

The syntax of the read() method is given below.

1. fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

## Example

```
1. #open the file.txt in read mode. causes error if no such file exists.  
2. fileptr = open("file.txt","r");  
3.  
4. #stores all the data of the file into the variable content  
5. content = fileptr.read(9);  
6.  
7. # prints the type of the data stored in the file  
8. print(type(content))  
9.  
10.#prints the content of the file  
11.print(content)  
12.  
13.#closes the opened file  
14.fileptr.close()
```

## Output:

```
<class 'str'>  
Hi, I am
```

## Read Lines of the file

Python facilitates us to read the file line by line by using a function readline(). The readline() method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function readline() that reads the first line of our file "**file.txt**" containing three lines.

## Example

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file.txt","r");
- 3.
4. #stores all the data of the file into the variable content
5. content = fileptr.readline();
- 6.
7. # prints the type of the data stored in the file
8. **print**(type(content))
- 9.
- 10.#prints the content of the file
- 11.**print**(content)
- 12.
- 13.#closes the opened file
- 14.fileptr.close()

## Output:

```
<class 'str'>
Hi, I am the file and being used as
```

## Looping through the file

By looping through the lines of the file, we can read the whole file.

## Example

1. `#open the file.txt in read mode. causes an error if no such file exists.`
- 2.
- 3.
4. `fileptr = open("file.txt","r");`
- 5.
6. `#running a for loop`
7. `for i in fileptr:`
8.   `print(i) # i contains each line of the file`

## Output:

```
Hi, I am the file and being used as  
an example to read a  
file in python.
```

## Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

**w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Consider the following example.

## Example 1

```
1. #open the file.txt in append mode. Creates a new file if no such file exists.  
2. fileptr = open("file.txt","a");  
3.  
4. #appending the content to the file  
5. fileptr.write("Python is the modern day language. It makes things so simple.")  
6.  
7.  
8. #closing the opened file  
9. fileptr.close();
```

Now, we can see that the content of the file is modified.

#### File.txt:

1. Hi, I am the file **and** being used as
2. an example to read a
3. file **in** python.
4. Python **is** the modern day language. It makes things so simple.

## Example 2

```
1. #open the file.txt in write mode.  
2. fileptr = open("file.txt","w");  
3.  
4. #overwriting the content of the file  
5. fileptr.write("Python is the modern day language. It makes things so simple.")  
6.  
7.  
8. #closing the opened file
```

```
9. fileptr.close();
```

Now, we can check that all the previously written content of the file is overwritten with the new text we have passed.

#### File.txt:

1. Python **is** the modern day language. It makes things so simple.

## Creating a new file

The new file can be created by using one of the following access modes with the function open().

**x:** it creates a new file with the specified name. It causes an error if a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

## Example

1. **#open the file.txt in read mode. causes error if no such file exists.**
2. **fileptr = open("file2.txt","x");**
- 3.
4. **print(fileptr)**
- 5.
6. **if fileptr:**
7.     **print("File created successfully");**

## **Output:**

```
File created successfully
```

## Using with statement with files

The with statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. The with statement is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with statement is given below.

1. `with open(<file name>, <access mode>) as <file-pointer>:`
2.     `#statement suite`

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the with statement in the case of file s because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file. It doesn't let the file to be corrupted.

Consider the following example.

## Example

1. `with open("file.txt",'r') as f:`
2.     `content = f.read();`
3.     `print(content)`

## **Output:**

```
Python is the modern day language. It makes things so simple.
```

## File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer exists. Consider the following example.

### Example

```
1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
6.
7. #reading the content of the file
8. content = fileptr.read();
9.
10.#after the read operation file pointer modifies. tell() returns the location of the fileptr.
11.
12.print("After reading, the filepointer is at:",fileptr.tell())
```

### Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at 26
```

## Modifying file pointer position

In the real world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the python provides us the seek() method which enables us to modify the file pointer position externally.

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

1. <file-ptr>.seek(offset[, **from**])

The seek() method accepts two parameters:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

## Example

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
- 3.
4. #initially the filepointer is at 0
5. **print**("The filepointer is at byte :",fileptr.tell())
- 6.
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
- 9.
10. #tell() returns the location of the fileptr.
11. **print**("After reading, the filepointer is at:",fileptr.tell())

## **Output:**

```
The filepointer is at byte : 0  
After reading, the filepointer is at 10
```

## **Python os module**

The os module provides us the functions that are involved in file processing operations like renaming, deleting, etc.

Let's look at some of the os module functions.

### **Renaming the file**

The os module provides us the rename() method which is used to rename the specified file to a new name. The syntax to use the rename() method is given below.

1. `rename(?current-name?, ?new-name?)`

### **Example**

1. `import os;`
- 2.
3. `#rename file2.txt to file3.txt`
4. `os.rename("file2.txt","file3.txt")`

### **Removing the file**

The os module provides us the remove() method which is used to remove the specified file. The syntax to use the remove() method is given below.

1. remove(?file-name?)

### Example

1. **import** os;
- 2.
3. #deleting the file named file3.txt
4. os.remove("file3.txt")

### Creating the new directory

The mkdir() method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

1. mkdir(?directory name?)

### Example

1. **import** os;
- 2.
3. #creating a new directory with the name new
4. os.mkdir("new")

### Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

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

1. `chdir("new-directory")`

## Example

1. `import os;`
- 2.
3. `#changing the current working directory to new`
- 4.
5. `os.chdir("new")`

## The getcwd() method

This method returns the current working directory.

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

1. `os.getcwd()`

## Example

1. `import os;`
- 2.
3. `#printing the current working directory`
4. `print(os.getcwd())`

## Deleting directory

The rmdir() method is used to delete the specified directory.

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

1. os.rmdir(?directory name?)

## Example

1. **import** os;
- 2.
3. **#removing the new directory**
4. os.rmdir("new")

## Writing python output to the files

In python, there are the requirements to write the output of a python script to a file.

The **check\_call()** method of module **subprocess** is used to execute a python script and write the output of that script to a file.

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt**

**file.py:**

1. temperatures=[10,-20,-289,100]
2. **def** c\_to\_f(c):
3.     **if** c< -273.15:
4.         **return** "That temperature doesn't make sense!"
5.     **else**:
6.         f=c\*9/5+32
7.         **return** f

8. **for** t **in** temperatures:
9.     **print**(c\_to\_f(t))

**file.py:**

1. **import** subprocess
- 2.
3. **with open("output.txt", "wb") as f:**
4.     **subprocess.check\_call(["python", "file.py"], stdout=f)**

**Output:**

```
50
-4
That temperature doesn't make sense!
212
```

## The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write any more.
2	File.flush()	It flushes the internal buffer.

3	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	File.isatty()	It returns true if the file is connected to a TTY device, otherwise returns false.
5	File.next()	It returns the next line from the file.
6	File.read([size])	It reads the file for the specified size.
7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from])	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file

13	File.writelines(seq)	It writes a sequence of the strings to a file.
----	----------------------	--

## Python Modules

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

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

### Example

In this example, we will create a module named as `file.py` which contains a function `func` that contains a code to print some message on the console.

Let's create the module named as **file.py**.

1. `#displayMsg prints a message to the name being passed.`
2. `def displayMsg(name)`
3.  `print("Hi "+name);`

Here, we need to include this module into our main module to call the method `displayMsg()` defined in the module named `file`.

## Loading the module in our python code

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

1. The import statement
2. The from-import statement

## The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below.

1. **import** module1,module2,..... module n

Hence, if we need to call the function displayMsg() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

### Example:

1. **import** file;
2. name = input("Enter the name?")
3. file.displayMsg(name)

### Output:

```
Enter the name?John
Hi John
```

## The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

1. **from** < module-name> **import** <name 1>, <name 2>..,<name n>

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

#### **calculation.py:**

1. #place the code in the calculation.py
2. **def** summation(a,b):
3.     **return** a+b
4. **def** multiplication(a,b):
5.     **return** a\*b;
6. **def** divide(a,b):
7.     **return** a/b;

#### **Main.py:**

1. **from** calculation **import** summation
2. #it will import only the summation() from calculation.py
3. a = int(input("Enter the first number"))
4. b = int(input("Enter the second number"))
5. **print**("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()

#### **Output:**

```
Enter the first number10
Enter the second number20
Sum = 30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using \*.

Consider the following syntax.

1. **from** <module> **import** \*

## Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

1. **import** <module-name> as <specific-name>

## Example

1. #the module calculation of previous example is imported in this example as cal.
2. **import** calculation as cal;
3. a = int(input("Enter a?"));
4. b = int(input("Enter b?"));
5. **print**("Sum = ",cal.summation(a,b))

### Output:

```
Enter a?10
Enter b?20
Sum = 30
```

## Using dir() function

The `dir()` function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Consider the following example.

## Example

1. `import json`
- 2.
3. `List = dir(json)`
- 4.
5. `print(List)`

### Output:

```
['JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtins__', '__cached__', '__doc__',  
 '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__',  
 '__default_decoder', '__default_encoder', 'decoder', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']
```

## The `reload()` function

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the `reload()` function. The syntax to use the `reload()` function is given below.

1. `reload(<module-name>)`

for example, to reload the module `calculation` defined in the previous example, we must use the following line of code.

1. `reload(calculation)`

## Scope of variables

In Python, variables are associated with two types of scopes. All the variables defined in a module contain the global scope unless or until it is defined within a function.

All the variables defined inside a function contain a local scope that is limited to this function itself. We can not access a local variable globally.

If two variables are defined with the same name with the two different scopes, i.e., local and global, then the priority will always be given to the local variable.

Consider the following example.

### Example

1. name = "john"
2. **def** print\_name(name):
3.     **print**("Hi",name) #prints the name that is local to this function only.
4. name = **input**("Enter the name?")
5. print\_name(name)

#### Output:

```
Hi David
```

---

## Python packages

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path **/home**.
2. Create a python source file with name ITEmployees.py on the path **/home/Employees**.

### **I<sup>T</sup>Employees.py**

1. **def** getITNames():
  2.   List = ["John", "David", "Nick", "Martin"]
  3.   **return** List;
3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().
4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is **\_\_init\_\_.py** which contains the import statements of the modules defined in this directory.

### **\_\_init\_\_.py**

1. **from** ITEmployees **import** getITNames
  2. **from** BPOEmployees **import** getBPONames
5. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create **\_\_init\_\_.py** inside a directory to convert this directory to a package.
6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

### **Test.py**

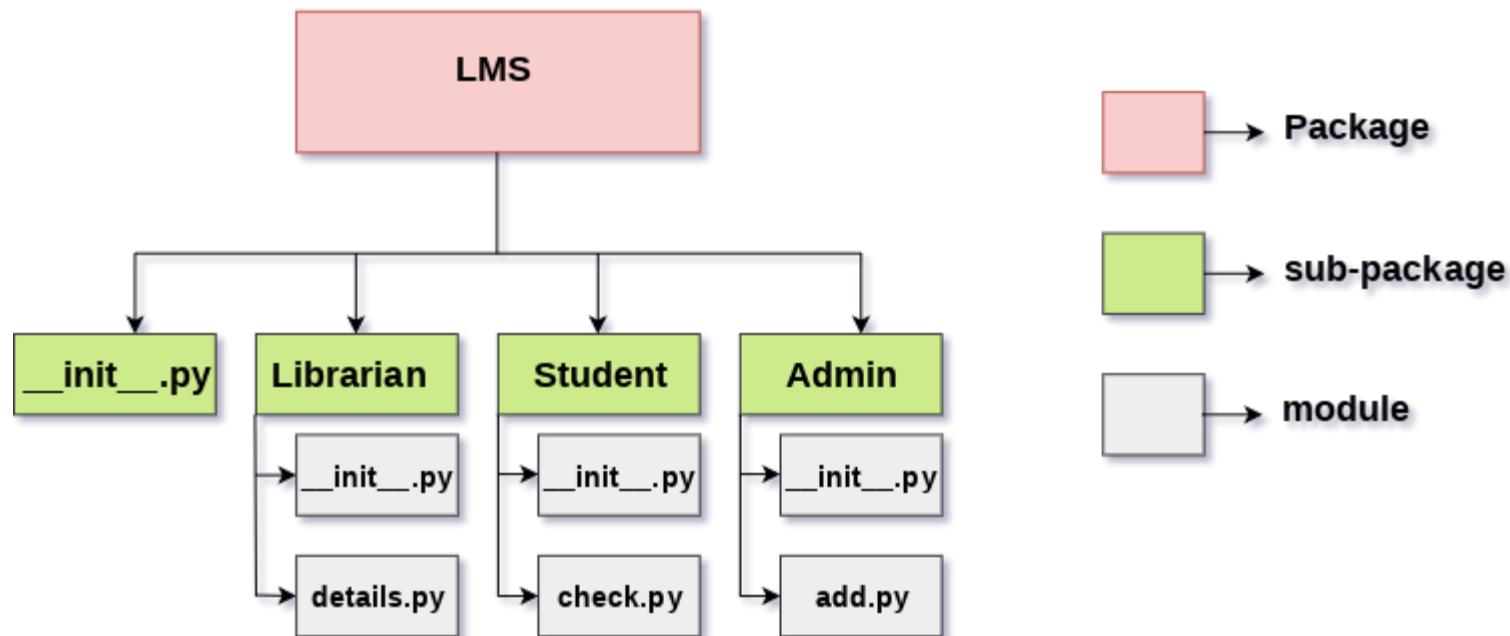
1. **import** Employees
2. **print**(Employees.getNames())

## Output:

```
['John', 'David', 'Nick', 'Martin']
```

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.



## Python Exceptions

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

Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after the that.

## Common Exceptions

A list of common exceptions that can be thrown from a normal python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

---

## Problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program. Consider the following example.

### Example

1. `a = int(input("Enter a:"))`
2. `b = int(input("Enter b:"))`
3. `c = a/b;`

4. `print("a/b = %d"%c)`
- 5.
6. `#other code:`
7. `print("Hi I am other part of the program")`

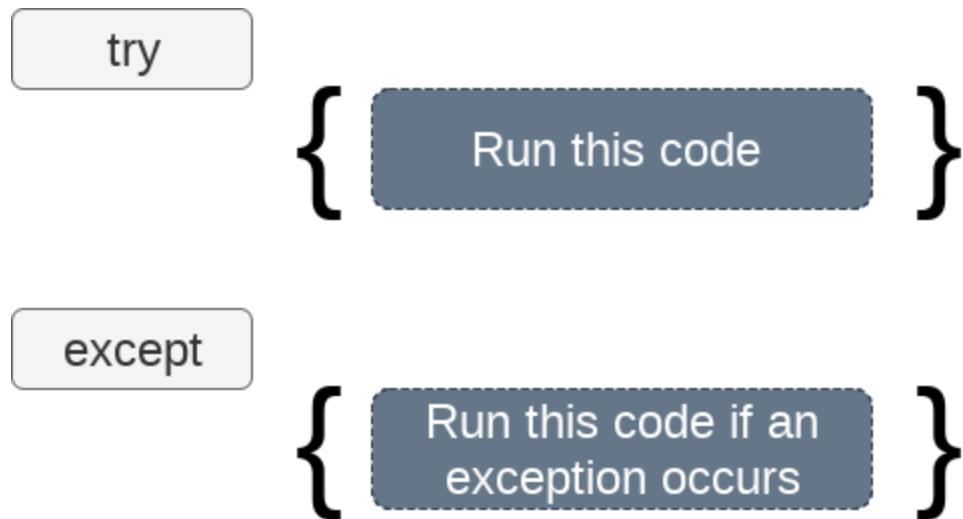
**Output:**

```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

---

## Exception handling in python

If the python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.



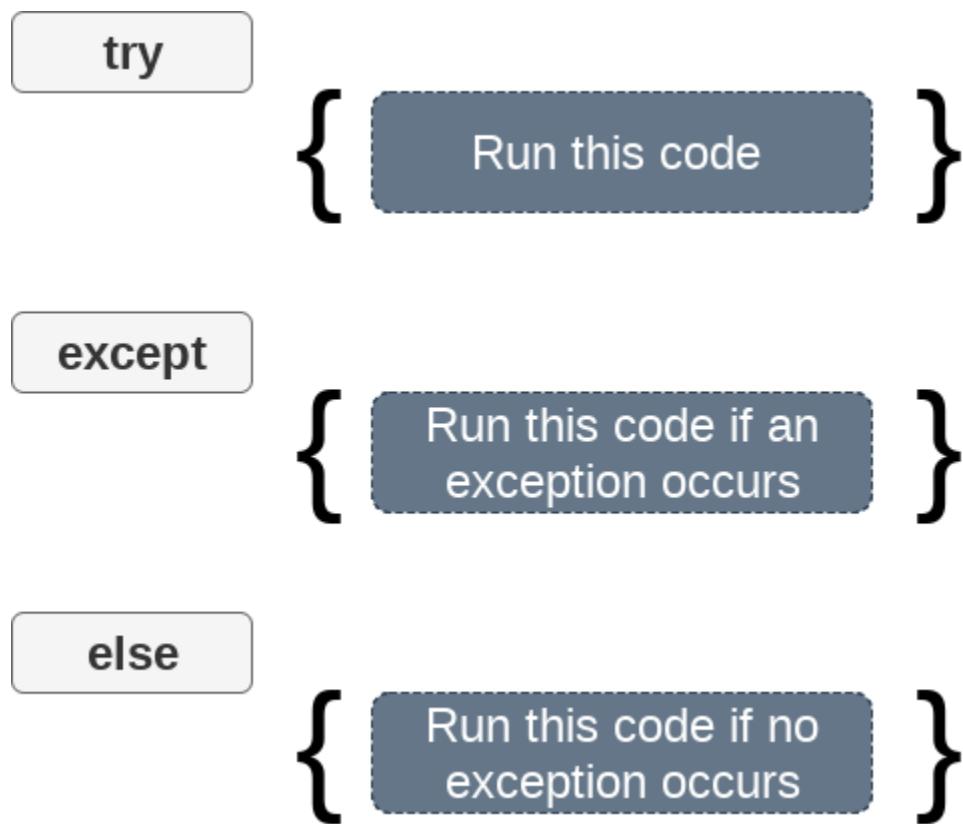
## Syntax

1. **try**:
2.    #block of code
- 3.
4. **except** Exception1:
5.    #block of code
- 6.
7. **except** Exception2:
8.    #block of code
- 9.
10. #other code

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.

1. **try**:
2.    #block of code
- 3.
4. **except** Exception1:
5.    #block of code
- 6.
7. **else**:
8.    #this code executes if no except block is executed



## Example

1. **try**:
2.   `a = int(input("Enter a:"))`
3.   `b = int(input("Enter b:"))`
4.   `c = a/b;`
5.   `print("a/b = %d"%c)`
6. **except** Exception:
7.   `print("can't divide by zero")`

```
8. else:  
9.     print("Hi I am else block")
```

**Output:**

```
Enter a:10  
Enter b:2  
a/b = 5  
Hi I am else block
```

## The except statement with no exception

Python provides the flexibility not to specify the name of exception with the except statement.

Consider the following example.

### Example

```
1. try:  
2.     a = int(input("Enter a:"))  
3.     b = int(input("Enter b:"))  
4.     c = a/b;  
5.     print("a/b = %d"%c)  
6. except:  
7.     print("can't divide by zero")  
8. else:  
9.     print("Hi I am else block")
```

**Output:**

```
Enter a:10
```

```
Enter b:0  
can't divide by zero
```

## Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

## Example

```
1. try:  
2.     #this will throw an exception if the file doesn't exist.  
3.     fileptr = open("file.txt","r")  
4. except IOError:  
5.     print("File not found")  
6. else:  
7.     print("The file opened successfully")  
8.     fileptr.close()
```

### Output:

```
File not found
```

---

## Declaring multiple exceptions

The python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

## Syntax

1. **try**:
2.    #block of code
- 3.
4. **except** (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
5.    #block of code
- 6.
7. **else**:
8.    #block of code

## Example

1. **try**:
2.    a=10/0;
3. **except** ArithmeticError:
4.    **print** "Arithmetic Exception"
5. **else**:
6.    **print** "Successfully Done"

### Output:

```
Arithmetic Exception
```

---

## The finally block

We can use the finally block with the try block in which, we can place the important code which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

## syntax

1. **try**:
2.    # block of code
3.    # this may throw an exception
4. **finally**:
5.    # block of code
6.    # this will always be executed

**try**

{ Run this code }

**except**

{ Run this code if an exception occurs }

**else**

{ Run this code if no exception occurs }

**finally**

{ Always run this code }

## Example

```
1. try:  
2.     fileptr = open("file.txt","r")  
3.     try:  
4.         fileptr.write("Hi I am good")  
5.     finally:  
6.         fileptr.close()  
7.         print("file closed")  
8. except:  
9.     print("Error")
```

### Output:

```
file closed  
Error
```

---

## Raising exceptions

An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

### syntax

```
1. raise Exception_class,<value>
```

### Points to remember

1. To raise an exception, raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.

3. To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

## Example

```
1. try:  
2.     age = int(input("Enter the age?"))  
3.     if age<18:  
4.         raise ValueError;  
5.     else:  
6.         print("the age is valid")  
7. except ValueError:  
8.     print("The age is not valid")
```

### Output:

```
Enter the age?17  
The age is not valid
```

## Example

```
1. try:  
2.     a = int(input("Enter a?"))  
3.     b = int(input("Enter b?"))  
4.     if b is 0:  
5.         raise ArithmeticError;  
6.     else:  
7.         print("a/b = ",a/b)  
8. except ArithmeticError:  
9.     print("The value of b can't be 0")
```

### Output:

```
Enter a?10
Enter b?0
The value of b can't be 0
```

## Custom Exception

The python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

Consider the following example.

### Example

```
1. class ErrorInCode(Exception):
2.     def __init__(self, data):
3.         self.data = data
4.     def __str__(self):
5.         return repr(self.data)
6.
7. try:
8.     raise ErrorInCode(2000)
9. except ErrorInCode as ae:
10.    print("Received error:", ae.data)
```

#### Output:

```
Received error: 2000
```

## Python Date and time

In the real world applications, there are the scenarios where we need to work with the date and time. There are the examples in python where we have to schedule the script to run at some particular timings.

In python, the date is not a data type, but we can work with the date objects by importing the module named with datetime, time, and calendar.

In this section of the tutorial, we will discuss how to work with the date and time objects in python.

---

## Tick

In python, the time instants are counted since 12 AM, 1st January 1970. The function time() of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Consider the following example.

## Example

1. `import time;`
- 2.
3. `#prints the number of ticks spent since 12 AM, 1st January 1970`
- 4.
5. `print(time.time())`

### Output:

```
1545124460.9151757
```

---

## How to get the current time?

The localtime() functions of the time module are used to get the current time tuple. Consider the following example.

## Example

1. `import time;`
- 2.
3. `#returns a time tuple`
- 4.
5. `print(time.localtime(time.time()))`

### Output:

```
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=18, tm_hour=15, tm_min=1,  
tm_sec=32, tm_wday=1, tm_yday=352, tm_isdst=0)
```

## Time tuple

The time is treated as the tuple of 9 numbers. Let's look at the members of the time tuple.

Index	Attribute	Values
0	Year	4 digit (for example 2018)
1	Month	1 to 12
2	Day	1 to 31

3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 60
6	Day of week	0 to 6
7	Day of year	1 to 366
8	Daylight savings	-1, 0, 1 , or -1

## Getting formatted time

The time can be formatted by using the `asctime()` function of `time` module. It returns the formatted time for the time tuple being passed.

### Example

1. `import time;`
- 2.
3. `#returns the formatted time`
- 4.
5. `print(time.asctime(time.localtime(time.time())))`

### **Output:**

```
Tue Dec 18 15:31:39 2018
```

---

## Python sleep time

The sleep() method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds given as float.

Consider the following example.

### Example

1. **import** time
2. **for** i **in** range(0,5):
3.     **print**(i)
4.     **#Each element will be printed after 1 second**
5.     time.sleep(1)

### **Output:**

```
0  
1  
2  
3  
4
```

---

## The datetime Module

The datetime module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import datetime module into the python source code.

Consider the following example to get the datetime object representation for the current time.

## Example

1. `import` datetime;
- 2.
3. `#returns the current datetime object`
- 4.
5. `print(datetime.datetime.now())`

### Output:

```
2018-12-18 16:16:45.462778
```

## Creating date objects

We can create the date objects by passing the desired date in the datetime constructor for which the date objects are to be created.

Consider the following example.

## Example

1. `import` datetime;
- 2.

3. `#returns the datetime object for the specified date`
- 4.
5. `print(datetime.datetime(2018,12,10))`

**Output:**

```
2018-12-10 00:00:00
```

We can also specify the time along with the date to create the datetime object. Consider the following example.

## Example

1. `import datetime;`
- 2.
3. `#returns the datetime object for the specified time`
- 4.
5. `print(datetime.datetime(2018,12,10,14,15,10))`

**Output:**

```
2018-12-10 14:15:10
```

## Comparison of two dates

We can compare two dates by using the comparison operators like `>`, `>=`, `<`, and `<=`.

Consider the following example.

## Example

```
1. from datetime import datetime as dt
2. #Compares the time. If the time is in between 8AM and 4PM, then it prints working hours otherwise it prints fun hours
3. if dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):
4.     print("Working hours....")
5. else:
6.     print("fun hours")
```

**Output:**

```
fun hours
```

---

## The calendar module

Python provides a calendar object that contains various methods to work with the calendars.

Consider the following example to print the Calendar of the last month of 2018.

### Example

```
1. import calendar;
2. cal = calendar.month(2018,12)
3. #printing the calendar of December 2018
4. print(cal)
```

**Output:**

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python3 time2.py  
December 2018  
Mo Tu We Th Fr Sa Su  
     1  2  
 3  4  5  6  7  8  9  
10 11 12 13 14 15 16  
17 18 19 20 21 22 23  
24 25 26 27 28 29 30  
31  
[javatpoint@localhost ~]$
```

javaTpoint

## Printing the calendar of whole year

The `prcal()` method of `calendar` module is used to print the calendar of the whole year. The year of which the calendar is to be printed must be passed into this method.

### Example

1. `import calendar`
- 2.
3. `#printing the calendar of the year 2019`
4. `calendar.prCal(2019)`

**Output:**

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

February						
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			

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

April						
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					

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

June						
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

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

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

September						
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						

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

November						
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	

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

javaTpoint

# Python OOPs Concepts

Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

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

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

## Object

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

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the doc string defined in the function source code.

## Class

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

## Syntax

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

## Method

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

## Inheritance

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

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

It provides re-usability of the code.

## Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many and Morphs means form, shape. By polymorphism, we understand that one task can be performed in different ways. For example You have a class animal, and all animals speak.

But they speak differently. Here, the "speak" behavior is polymorphic in the sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

## Encapsulation

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

## Data Abstraction

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

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

---

## Object-oriented vs Procedure-oriented Programming languages

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.

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

## Python Class and Objects

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

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

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

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

## Creating classes in python

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

### Syntax

1. `class` ClassName:
2.   `#statement_suite`

In python, we must notice that each class is associated with a documentation string which can be accessed by using `<classname>.__doc__`. A class contains a statement suite including fields, constructor, function, etc. definition.

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

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

### Example

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

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

## Creating an instance of the class

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

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

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

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

### Example

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

#### Output:

```
ID: 10
Name: ayush
```

## Python Constructor

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

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

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

---

## Creating the constructor in python

In python, the method `__init__` simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon `__init__` definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the Employee class attributes.

### Example

```
1. class Employee:  
2.     def __init__(self,name,id):  
3.         self.id = id;  
4.         self.name = name;  
5.     def display (self):  
6.         print("ID: %d \nName: %s"%(self.id,self.name))  
7. emp1 = Employee("John",101)  
8. emp2 = Employee("David",102)  
9.  
10. #accessing display() method to print employee 1 information  
11.  
12.emp1.display();
```

- 13.
14. #accessing display() method to print employee 2 information
15. emp2.display();

**Output:**

```
ID: 101
Name: John
ID: 102
Name: David
```

## Example: Counting the number of objects of a class

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

**Output:**

```
The number of students: 3
```

---

## Python Non-Parameterized Constructor Example

1. **class** Student:
2.     **# Constructor - non parameterized**
3.     **def** \_\_init\_\_(self):

```
4.     print("This is non parametrized constructor")
5.     def show(self,name):
6.         print("Hello",name)
7. student = Student()
8. student.show("John")
```

**Output:**

```
This is non parametrized constructor
Hello John
```

---

## Python Parameterized Constructor Example

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

**Output:**

```
This is parametrized constructor
Hello John
```

---

## Python In-built class functions

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

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

### Example

1. `class Student:`
2.   `def __init__(self, name, id, age):`
3.     `self.name = name;`
4.     `self.id = id;`
5.     `self.age = age`
- 6.
7.   `#creates the object of the class Student`
8.   `s = Student("John", 101, 22)`
- 9.
10. `#prints the attribute name of the object s`

```
11. print(getattr(s,'name'))
12.
13. # reset the value of attribute age to 23
14. setattr(s,"age",23)
15.
16. # prints the modified value of age
17. print(getattr(s,'age'))
18.
19. # prints true if the student contains the attribute with name id
20.
21. print(hasattr(s,'id'))
22. # deletes the attribute age
23. delattr(s,'age')
24.
25. # this will give an error since the attribute age has been deleted
26. print(s.age)
```

#### Output:

```
John
23
True
AttributeError: 'Student' object has no attribute 'age'
```

## Built-in class attributes

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

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

SN	Attribute	Description
1	<code>__dict__</code>	It provides the dictionary containing the information about the class namespace.
2	<code>__doc__</code>	It contains a string which has the class documentation
3	<code>__name__</code>	It is used to access the class name.
4	<code>__module__</code>	It is used to access the module in which, this class is defined.
5	<code>__bases__</code>	It contains a tuple including all base classes.

## Example

```
1. class Student:  
2.     def __init__(self,name,id,age):  
3.         self.name = name;  
4.         self.id = id;  
5.         self.age = age  
6.     def display_details(self):  
7.         print("Name:%s, ID:%d, age:%d"%(self.name,self.id))  
8. s = Student("John",101,22)  
9. print(s.__doc__)  
10. print(s.__dict__)
```

```
11. print(s.__module__)
```

**Output:**

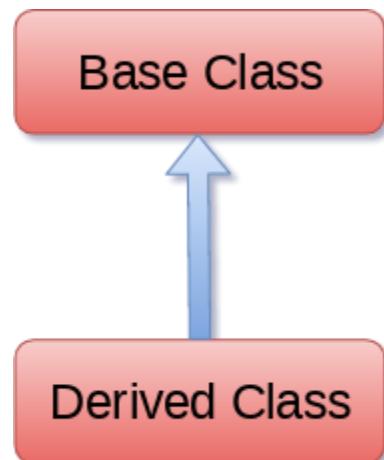
```
None
{'name': 'John', 'id': 101, 'age': 22}
__main__
```

## Python Inheritance

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

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

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



## Syntax

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

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

## Syntax

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

## Example 1

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

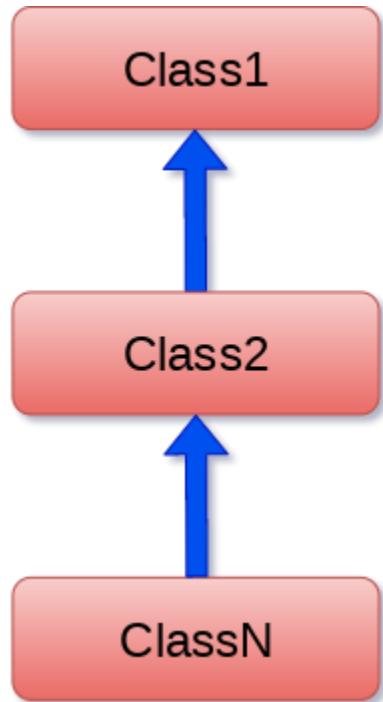
### Output:

```
dog barking  
Animal Speaking
```

---

## Python Multi-Level inheritance

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



The syntax of multi-level inheritance is given below.

## Syntax

1. `class` class1:
2.    `<class-suite>`
3. `class` class2(class1):
4.    `<class suite>`

```
5. class class3(class2):  
6.     <class suite>  
7. .  
8. .
```

## Example

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

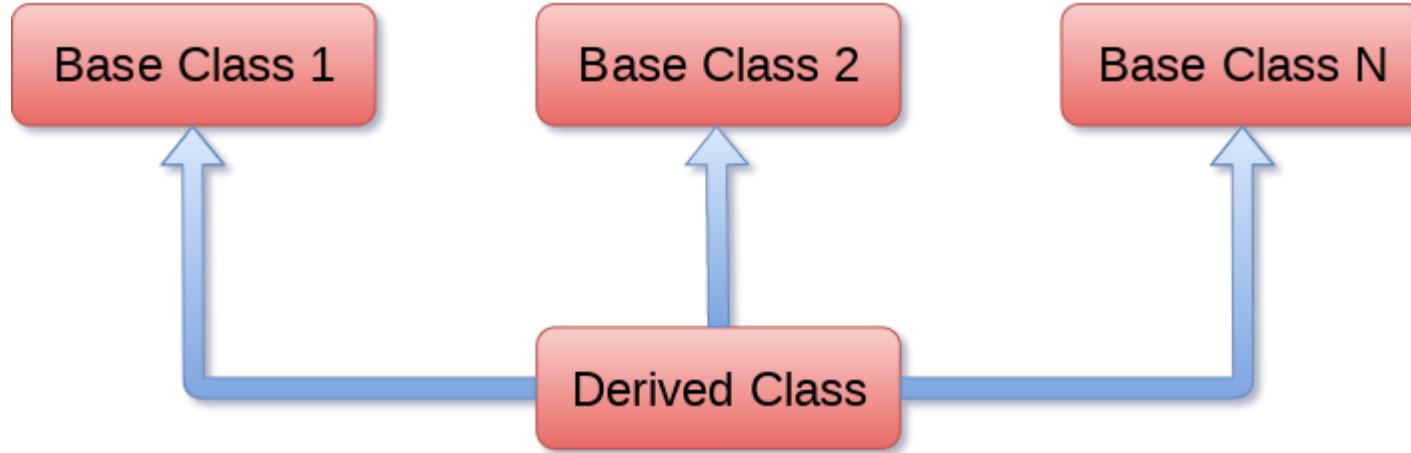
### Output:

```
dog barking  
Animal Speaking  
Eating bread...
```

---

## Python Multiple inheritance

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



The syntax to perform multiple inheritance is given below.

## Syntax

1. **class** Base1:
2.   <**class**-suite>
- 3.
4. **class** Base2:
5.   <**class**-suite>
6. .
7. .
8. .
9. **class** BaseN:
10.   <**class**-suite>
- 11.

12. **class** Derived(Base1, Base2, ..... BaseN):

13. <**class**-suite>

## Example

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

### Output:

```
30  
200  
0.5
```

## The issubclass(sub, sup) method

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

Consider the following example.

## Example

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

### Output:

```
True  
False
```

## The isinstance (obj, class) method

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

Consider the following example.

## Example

```
1. class Calculation1:
```

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

#### Output:

True

## Method Overriding

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

Consider the following example to perform method overriding in python.

## Example

```
1. class Animal:
2.     def speak(self):
3.         print("speaking")
4. class Dog(Animal):
```

```
5.     def speak(self):
6.         print("Barking")
7. d = Dog()
8. d.speak()
```

**Output:**

```
Barking
```

## Real Life Example of method overriding

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

**Output:**

```
Bank Rate of interest: 10
```

```
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

---

## Data abstraction in python

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

Consider the following example.

### Example

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

#### Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

## Python program to print "Hello Python"

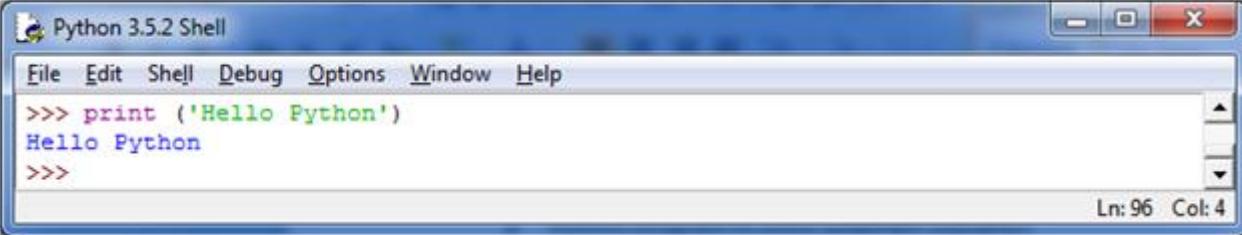
This is the most basic Python program. It specifies how to print any statement in Python.

In old python (up to Python 2.7.0), print command is not written in parenthesis but in new python software (Python 3.4.3), it is mandatory to add a parenthesis to print a statement.

**See this example:**

1. `print ('Hello Python')`

**Output:**



A screenshot of the Python 3.5.2 Shell window. The title bar says "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the following text:  
>>> print ('Hello Python')  
Hello Python  
>>>  
Ln: 96 Col: 4

## Python program to do arithmetical operations

The arithmetic operations are performed by calculator where we can perform addition, subtraction, multiplication and division. This example shows the basic arithmetic operations i.e.

- Addition
- Subtraction
- Multiplication
- Division

**See this example:**

```
1. # Store input numbers:  
2. num1 = input('Enter first number: ')  
3. num2 = input('Enter second number: ')  
4.  
5. # Add two numbers  
6. sum = float(num1) + float(num2)  
7. # Subtract two numbers  
8. min = float(num1) - float(num2)  
9. # Multiply two numbers  
10. mul = float(num1) * float(num2)  
11. # Divide two numbers  
12. div = float(num1) / float(num2)  
13. # Display the sum  
14. print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))  
15.  
16. # Display the subtraction  
17. print('The subtraction of {0} and {1} is {2}'.format(num1, num2, min))  
18. # Display the multiplication  
19. print('The multiplication of {0} and {1} is {2}'.format(num1, num2, mul))  
20. # Display the division  
21. print('The division of {0} and {1} is {2}'.format(num1, num2, div))
```

**Note:** Here input numbers are 10 and 20.

**Output:**

The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code and its execution:

```
>>> # Store input numbers
>>> num1 = input('Enter first number: ')
Enter first number: 10
>>> num2 = input('Enter second number: ')
Enter second number: 20
>>> # Add two numbers
>>> sum = float(num1) + float(num2)
>>> # Subtract two numbers
>>> min = float(num1) - float(num2)
>>> # Multiply two numbers
>>> mul = float(num1) * float(num2)
>>> # Divide two numbers
>>> div = float(num1) / float(num2)
>>> # Display the sum
>>> print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
The sum of 10 and 20 is 30.0
>>> # Display the subtraction
>>> print('The subtraction of {0} and {1} is {2}'.format(num1, num2, min))
The subtraction of 10 and 20 is -10.0
>>> # Display the multiplication
>>> print('The multiplication of {0} and {1} is {2}'.format(num1, num2, mul))
The multiplication of 10 and 20 is 200.0
>>> # Display the division
>>> print('The division of {0} and {1} is {2}'.format(num1, num2, div))
The division of 10 and 20 is 0.5
>>> |
```

The status bar at the bottom right indicates "Ln: 28 Col: 4".

## Python program to solve quadratic equation

### Quadratic equation:

Quadratic equation is made from a Latin term "quadrates" which means square. It is a special type of equation having the form of:

$$ax^2+bx+c=0$$

Here, "x" is unknown which you have to find and "a", "b", "c" specifies the numbers such that "a" is not equal to 0. If  $a = 0$  then the equation becomes liner not quadratic anymore.

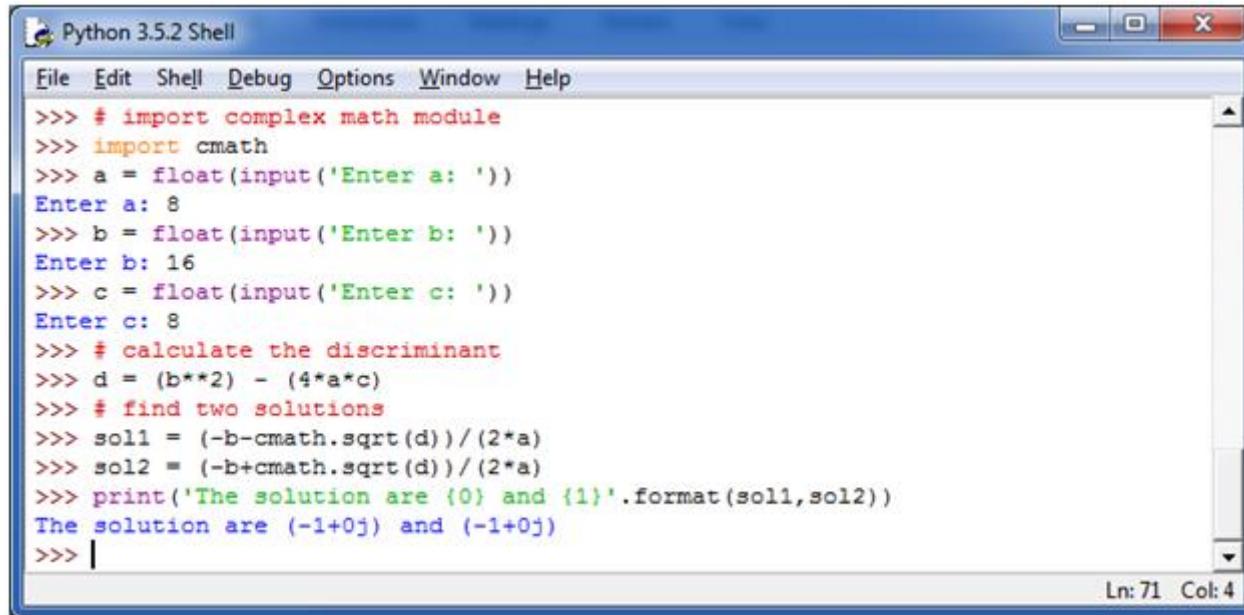
In the equation, a, b and c are called coefficients.

Let's take an example to solve the quadratic equation  $8x^2 + 16x + 8 = 0$

#### **See this example:**

```
1. # import complex math module
2. import cmath
3. a = float(input('Enter a: '))
4. b = float(input('Enter b: '))
5. c = float(input('Enter c: '))
6.
7. # calculate the discriminant
8. d = (b**2) - (4*a*c)
9.
10. # find two solutions
11. sol1 = (-b-cmath.sqrt(d))/(2*a)
12. sol2 = (-b+cmath.sqrt(d))/(2*a)
13. print('The solution are {0} and {1}'.format(sol1,sol2))
```

#### **Output:**



The screenshot shows the Python 3.5.2 Shell window. The code calculates the discriminant and finds two solutions for a quadratic equation. It uses the cmath module for complex numbers.

```
>>> # import complex math module
>>> import cmath
>>> a = float(input('Enter a: '))
Enter a: 8
>>> b = float(input('Enter b: '))
Enter b: 16
>>> c = float(input('Enter c: '))
Enter c: 8
>>> # calculate the discriminant
>>> d = (b**2) - (4*a*c)
>>> # find two solutions
>>> sol1 = (-b-cmath.sqrt(d))/(2*a)
>>> sol2 = (-b+cmath.sqrt(d))/(2*a)
>>> print('The solution are {0} and {1}'.format(sol1,sol2))
The solution are (-1+0j) and (-1+0j)
>>> |
```

## Python program to swap two variables

### Variable swapping:

In computer programming, swapping two variables specifies the mutual exchange of values of the variables. It is generally done by using a temporary variable.

For example:

1. data\_item x := 1
2. data\_item y := 0
3. swap (x, y)

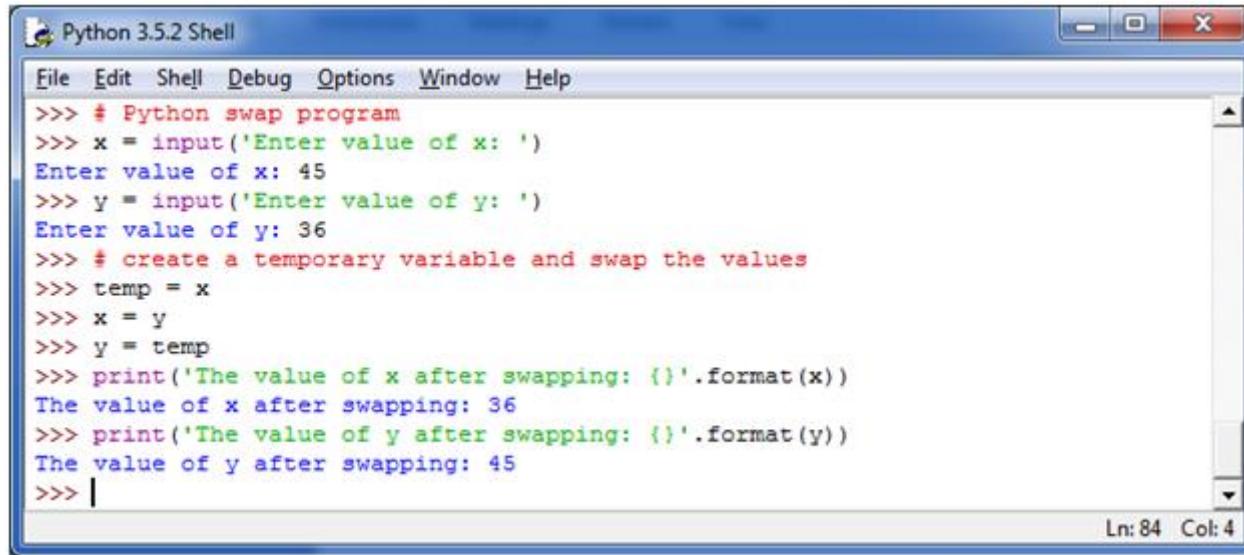
### After swapping:

1. data\_item x := 0
2. data\_item y := 1

**See this example:**

1. # Python swap program
2. x = input('Enter value of x: ')
3. y = input('Enter value of y: ')
- 4.
5. # create a temporary variable and swap the values
6. temp = x
7. x = y
8. y = temp
- 9.
10. print('The value of x after swapping: {}'.format(x))
11. print('The value of y after swapping: {}'.format(y))

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The title bar reads "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code and its execution:

```
>>> # Python swap program
>>> x = input('Enter value of x: ')
Enter value of x: 45
>>> y = input('Enter value of y: ')
Enter value of y: 36
>>> # create a temporary variable and swap the values
>>> temp = x
>>> x = y
>>> y = temp
>>> print('The value of x after swapping: {}'.format(x))
The value of x after swapping: 36
>>> print('The value of y after swapping: {}'.format(y))
The value of y after swapping: 45
>>> |
```

The status bar at the bottom right indicates "Ln: 84 Col: 4".

## Python program to generate a random number

In Python programming, you can generate a random integer, doubles, longs etc . in various ranges by importing a "random" class.

### Syntax:

First you have to import the random module and then apply the syntax:

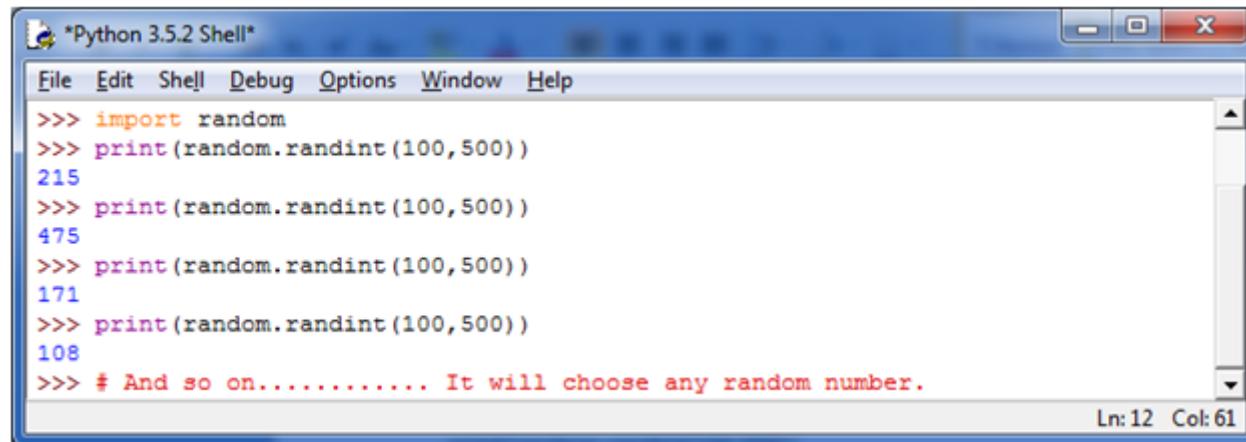
1. **import** random
2. random.randint(a,b)

### See this example:

1. **import** random

2. `print(random.randint(100,500))`

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains several print statements using the random.randint function. The output pane shows the results of these prints: 215, 475, 171, and 108, followed by a comment "# And so on..... It will choose any random number." The status bar at the bottom right indicates Ln:12 Col:61.

```
>>> import random
>>> print(random.randint(100,500))
215
>>> print(random.randint(100,500))
475
>>> print(random.randint(100,500))
171
>>> print(random.randint(100,500))
108
>>> # And so on..... It will choose any random number.
```

## Python program to convert kilometers to miles

Here, we are going to see the python program to convert kilometers to miles. Let's understand kilometers and miles first.

### Kilometer:

The kilometer is a unit of length in the metric system. It is equivalent to 1000 meters.

### Miles:

Mile is also the unit of length. It is equal to 1760 yards.

### Conversion formula:

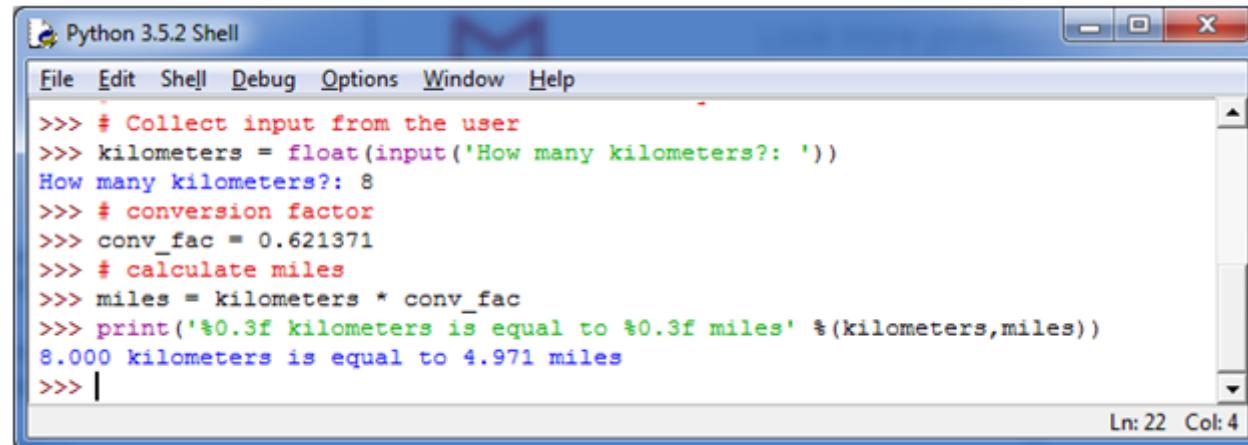
1 kilometer is equal to 0.62137 miles.

1. Miles = kilometer \* 0.62137
2. Kilometer = Miles / 0.62137

**See this example:**

```
1. # Collect input from the user
2. kilometers = float(input('How many kilometers?: '))
3. # conversion factor
4. conv_fac = 0.621371
5. # calculate miles
6. miles = kilometers * conv_fac
7. print('%0.3f kilometers is equal to %0.3f miles' %(kilometers,miles))
```

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays a Python script and its execution output. The script code is:

```
>>> # Collect input from the user
>>> kilometers = float(input('How many kilometers?: '))
How many kilometers?: 8
>>> # conversion factor
>>> conv_fac = 0.621371
>>> # calculate miles
>>> miles = kilometers * conv_fac
>>> print('%0.3f kilometers is equal to %0.3f miles' %(kilometers,miles))
8.000 kilometers is equal to 4.971 miles
>>> |
```

The status bar at the bottom right indicates "Ln: 22 Col: 4".

## Python program to convert Celsius to Fahrenheit

### **Celsius:**

Celsius is a unit of measurement for temperature. It is also known as centigrade. It is a SI derived unit used by most of the countries worldwide.

It is named after the Swedish astronomer Anders Celsius.

### **Fahrenheit:**

Fahrenheit is also a temperature scale. It is named on Polish-born German physicist Daniel Gabriel Fahrenheit. It uses degree Fahrenheit as a unit for temperature.

### **Conversion formula:**

$$T(^{\circ}\text{F}) = T(^{\circ}\text{C}) \times 9/5 + 32$$

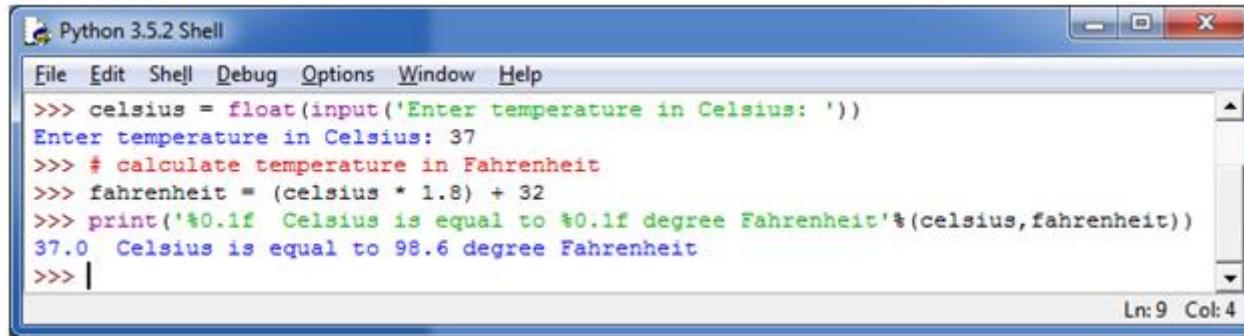
Or,

$$T(^{\circ}\text{F}) = T(^{\circ}\text{C}) \times 1.8 + 32$$

### **See this example:**

1. `# Collect input from the user`
2. `celsius = float(input('Enter temperature in Celsius: '))`
3.
4. `# calculate temperature in Fahrenheit`
5. `fahrenheit = (celsius * 1.8) + 32`
6. `print('%.1f Celsius is equal to %.1f degree Fahrenheit'%(celsius,fahrenheit))`

### **Output:**



A screenshot of the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code entered is:

```
>>> celsius = float(input('Enter temperature in Celsius: '))
Enter temperature in Celsius: 37
>>> # calculate temperature in Fahrenheit
>>> fahrenheit = (celsius * 1.8) + 32
>>> print('%0.1f Celsius is equal to %0.1f degree Fahrenheit'%(celsius,fahrenheit))
37.0 Celsius is equal to 98.6 degree Fahrenheit
>>> |
```

The output shows the conversion of 37 degrees Celsius to 98.6 degrees Fahrenheit.

## Python program to display calendar

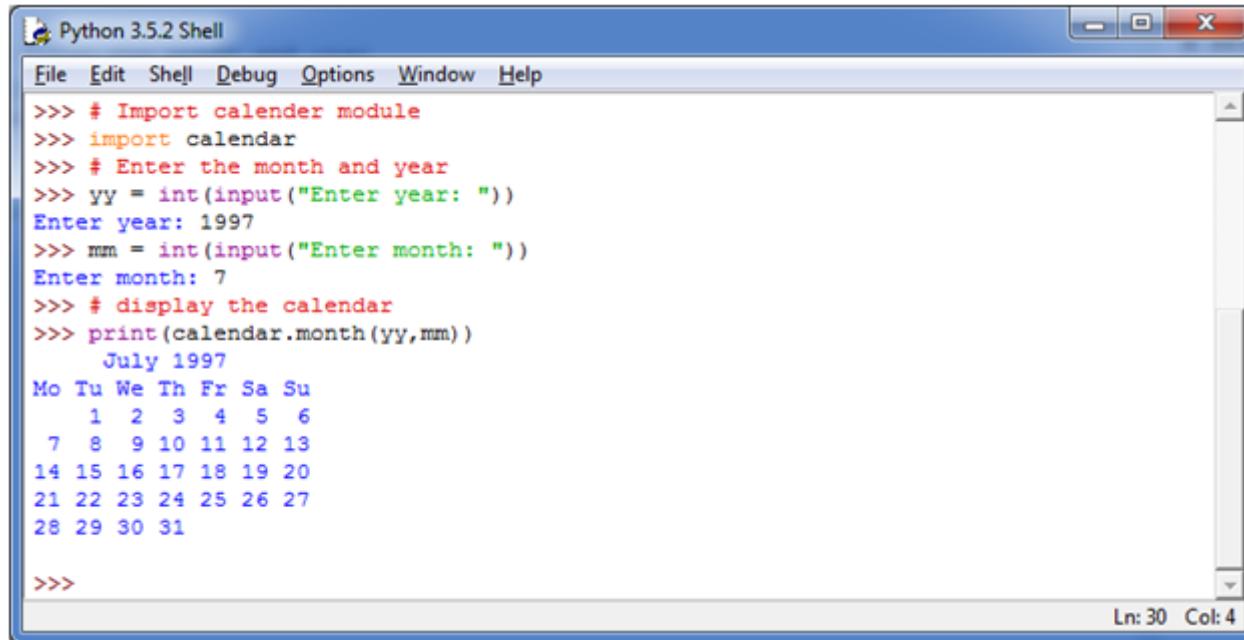
It is simple in python programming to display calendar. To do so, you need to import the calendar module which comes with Python.

1. **import** calendar
2. And then apply the syntax
3. (calendar.month(yy,mm))

### See this example:

1. **import** calendar
2. **# Enter the month and year**
3. yy = int(input("Enter year: "))
4. mm = int(input("Enter month: "))
- 5.
6. **# display the calendar**
7. **print**(calendar.month(yy,mm))

### Output:



The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code in the shell window is as follows:

```
>>> # Import calendar module
>>> import calendar
>>> # Enter the month and year
>>> yy = int(input("Enter year: "))
Enter year: 1997
>>> mm = int(input("Enter month: "))
Enter month: 7
>>> # display the calendar
>>> print(calendar.month(yy,mm))
    July 1997
Mo Tu We Th Fr Sa Su
    1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31

>>>
```

The status bar at the bottom right indicates Ln: 30 Col: 4.

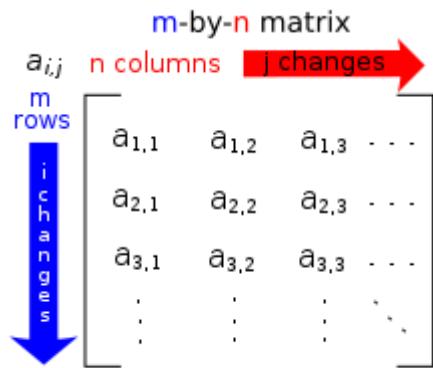
## Python Program to Add Two Matrices

### What is Matrix?

In mathematics, matrix is a rectangular array of numbers, symbols or expressions arranged in the form of rows and columns. For example: if you take a matrix A which is a 2x3 matrix then it can be shown like this:

1. 2     3     5
2. 8     12    7

### Image representation:



In Python, matrices can be implemented as nested list. Each element of the matrix is treated as a row. For example  $X = [[1, 2], [3, 4], [5, 6]]$  would represent a  $3 \times 2$  matrix. First row can be selected as  $X[0]$  and the element in first row, first column can be selected as  $X[0][0]$ .

**Let's take two matrices X and Y, having the following value:**

1.  $X = [[1, 2, 3],$
2.  $[4, 5, 6],$
3.  $[7, 8, 9]]$
- 4.
5.  $Y = [[10, 11, 12],$
6.  $[13, 14, 15],$
7.  $[16, 17, 18]]$

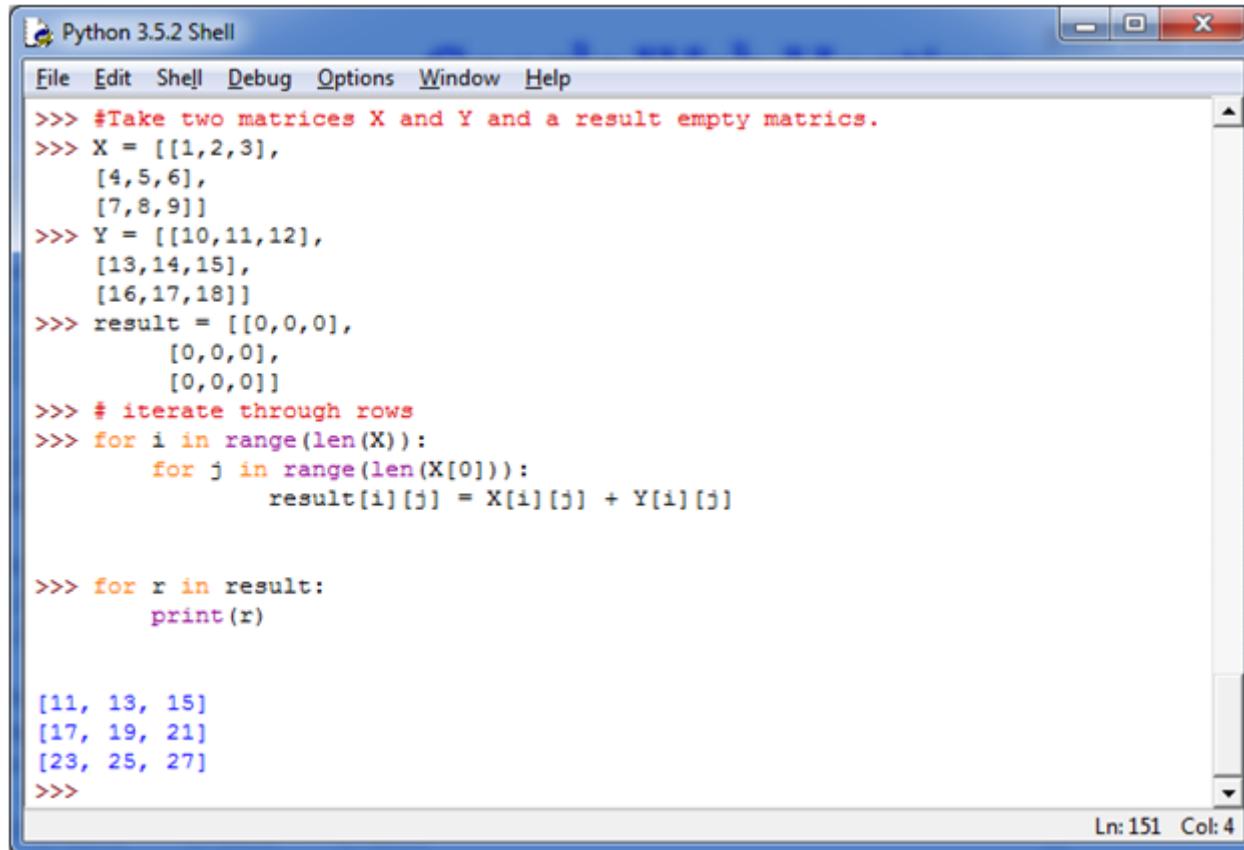
Create a new matrix result by adding them.

**See this example:**

1.  $X = [[1, 2, 3],$

```
2.      [4,5,6],  
3.      [7,8,9]]  
4.  
5. Y = [[10,11,12],  
6.      [13,14,15],  
7.      [16,17,18]]  
8.  
9. Result = [[0,0,0],  
10.        [0,0,0],  
11.        [0,0,0]]  
12. # iterate through rows  
13. for i in range(len(X)):  
14.   # iterate through columns  
15.   for j in range(len(X[0])):  
16.     result[i][j] = X[i][j] + Y[i][j]  
17. for r in result:  
18.   print(r)
```

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code in the shell window is as follows:

```
>>> #Take two matrices X and Y and a result empty matrices.  
>>> X = [[1,2,3],  
        [4,5,6],  
        [7,8,9]]  
>>> Y = [[10,11,12],  
        [13,14,15],  
        [16,17,18]]  
>>> result = [[0,0,0],  
            [0,0,0],  
            [0,0,0]]  
>>> # iterate through rows  
>>> for i in range(len(X)):  
    for j in range(len(X[0])):  
        result[i][j] = X[i][j] + Y[i][j]  
  
>>> for r in result:  
    print(r)  
  
[11, 13, 15]  
[17, 19, 21]  
[23, 25, 27]  
>>>
```

The status bar at the bottom right indicates Ln: 151 Col: 4.

## Python Program to Multiply Two Matrices

This Python program specifies how to multiply two matrices, having some certain values.

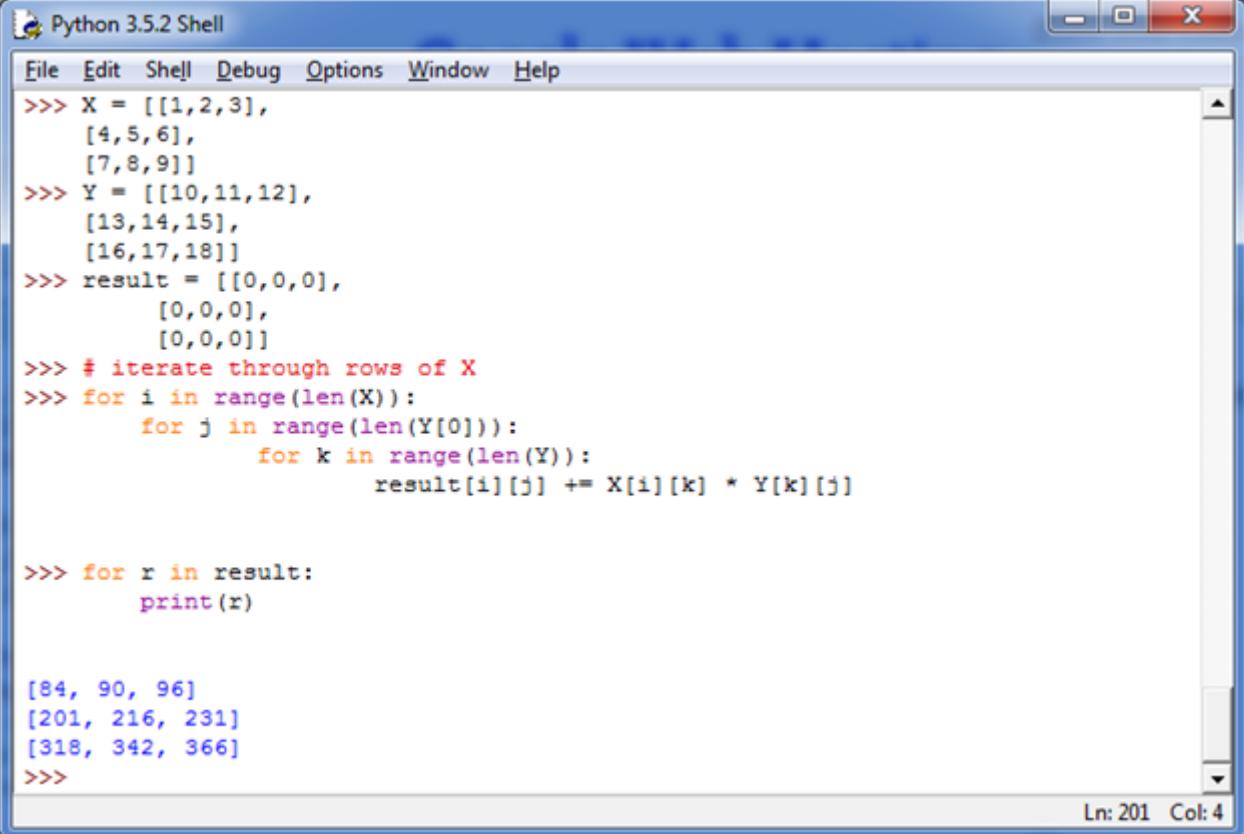
### Matrix multiplication:

Matrix multiplication is a binary operation that uses a pair of matrices to produce another matrix. The elements within the matrix are multiplied according to elementary arithmetic.

**See this example:**

```
1.  
2. X = [[1,2,3],  
3.         [4,5,6],  
4.         [7,8,9]]  
5.  
6. Y = [[10,11,12],  
7.         [13,14,15],  
8.         [16,17,18]]  
9.  
10. Result = [[0,0,0],  
11.             [0,0,0],  
12.             [0,0,0]]  
13.  
14. # iterate through rows of X  
15. for i in range(len(X)):  
16.     for j in range(len(Y[0])):  
17.         for k in range(len(Y)):  
18.             result[i][j] += X[i][k] * Y[k][j]  
19. for r in result:  
20.     print(r)
```

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code in the shell window is as follows:

```
>>> X = [[1,2,3],
   [4,5,6],
   [7,8,9]]
>>> Y = [[10,11,12],
   [13,14,15],
   [16,17,18]]
>>> result = [[0,0,0],
   [0,0,0],
   [0,0,0]]
>>> # iterate through rows of X
>>> for i in range(len(X)):
    for j in range(len(Y[0])):
        for k in range(len(Y)):
            result[i][j] += X[i][k] * Y[k][j]

>>> for r in result:
    print(r)

[84, 90, 96]
[201, 216, 231]
[318, 342, 366]
>>>
```

The status bar at the bottom right indicates Ln: 201 Col: 4.

## Python Program to Transpose a Matrix

### Transpose Matrix:

If you change the rows of a matrix with the column of the same matrix, it is known as transpose of a matrix. It is denoted as  $X'$ . **For example:** The element at  $i^{\text{th}}$  row and  $j^{\text{th}}$  column in  $X$  will be placed at  $j^{\text{th}}$  row and  $i^{\text{th}}$  column in  $X'$ .

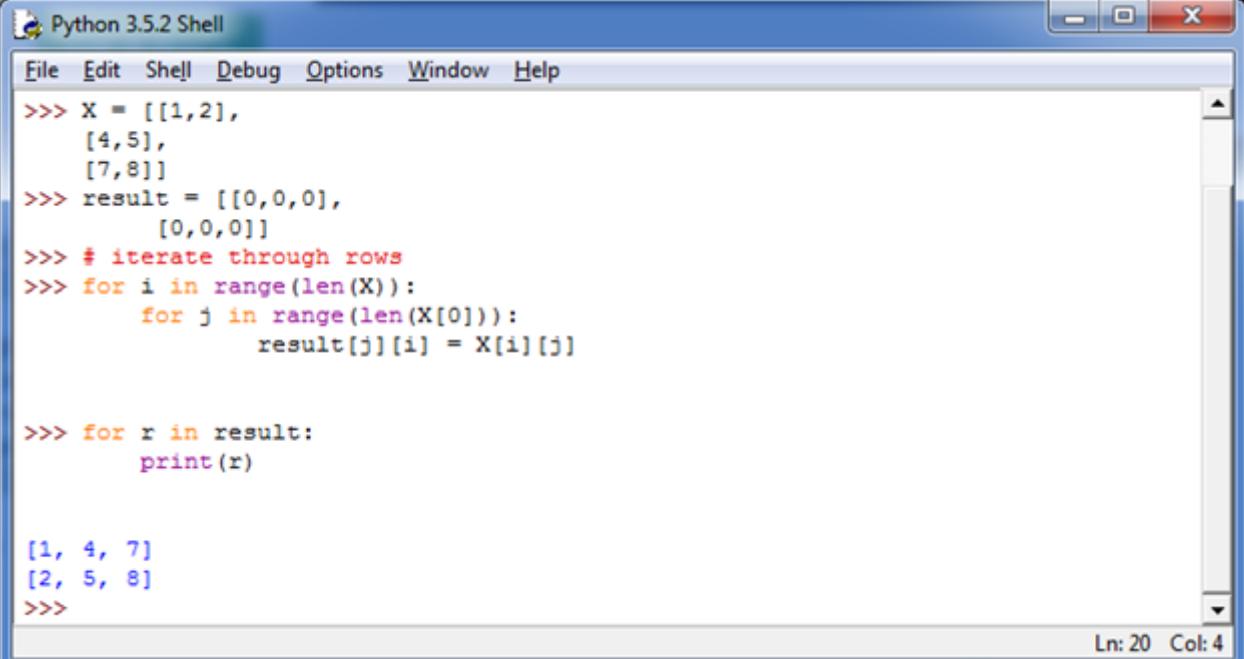
Let's take a matrix  $X$ , having the following elements:

```
1. X = [[1,2],  
2.      [4,5],  
3.      [7,8]]
```

**See this example:**

```
1. X = [[1,2],  
2.      [4,5],  
3.      [7,8]]  
4.  
5. Result = [[0,0,0],  
6.            [0,0,0]]  
7.  
8. # iterate through rows  
9. for i in range(len(X)):  
10.   for j in range(len(X[0])):  
11.     result[j][i] = X[i][j]  
12.  
13. for r in result:  
14.   print(r)
```

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python code:

```
>>> X = [[1,2],
[4,5],
[7,8]]
>>> result = [[0,0,0],
[0,0,0]]
>>> # iterate through rows
>>> for i in range(len(X)):
    for j in range(len(X[0])):
        result[j][i] = X[i][j]

>>> for r in result:
    print(r)

[1, 4, 7]
[2, 5, 8]
>>>
```

The output window shows the resulting list of lists:

```
[1, 4, 7]
[2, 5, 8]
```

In the bottom right corner of the shell window, there is a status bar with "Ln: 20 Col: 4".

## Python Program to Sort Words in Alphabetic Order

### Sorting:

Sorting is a process of arrangement. It arranges data systematically in a particular format. It follows some algorithm to sort data.

### See this example:

1. my\_str = input("Enter a string: ")
2. # breakdown the string into a list of words
3. words = my\_str.split()

```
4. # sort the list
5. words.sort()
6. # display the sorted words
7. for word in words:
8.     print(word)
```

**Output:**

The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python script:

```
>>> # collect the input from the user
>>> my_str = input("Enter a string: ")
Enter a string: My name is khan and I am not a terrorist.
>>> # breakdown the string into a list of words
>>> words = my_str.split()
>>> # sort the list
>>> words.sort()
>>> # display the sorted words
>>> for word in words:
    print(word)
```

The output window displays the sorted words:

```
I
My
a
am
and
is
khan
name
not
terrorist.
```

The status bar at the bottom right indicates Ln: 239 Col: 4.

# Python Program to Remove Punctuation from a String

## Punctuation:

The practice, action, or system of inserting points or other small marks into texts, in order to aid interpretation; division of text into sentences, clauses, etc., is called punctuation. -Wikipedia

Punctuation are very powerful. They can change the entire meaning of a sentence.

## See this example:

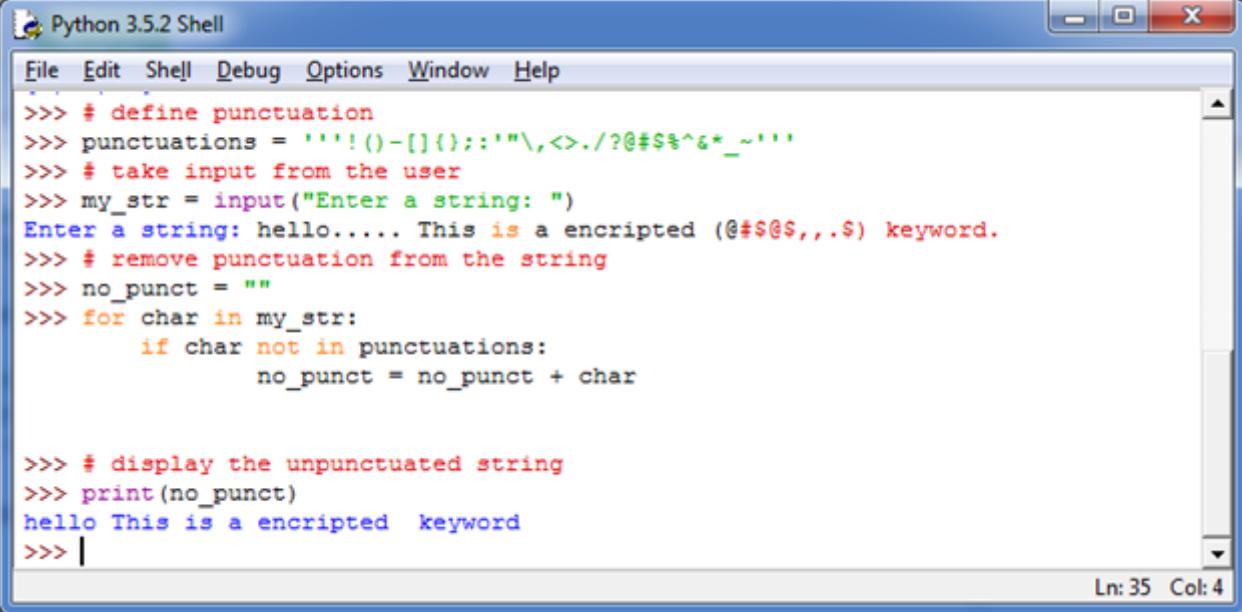
- "Woman, without her man, is nothing" (the sentence boasting about men's importance.)
- "Woman: without her, man is nothing" (the sentence boasting about women?s importance.)

This program is written to remove punctuation from a statement.

## See this example:

```
1. # define punctuation
2. punctuation = '''!()-[]{};:'"\,;<>./?@#$%^&*_~'''
3. # take input from the user
4. my_str = input("Enter a string: ")
5. # remove punctuation from the string
6. no_punct = ""
7. for char in my_str:
8.     if char not in punctuation:
9.         no_punct = no_punct + char
10. # display the unpunctuated string
11. print(no_punct)
```

## Output:



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its execution output. The code defines punctuation characters, takes user input, removes punctuation from the string, and prints the result. The output shows the original string with punctuation removed.

```
>>> # define punctuation
>>> punctuations = '''!()-[]{};:'"\,.>./?@#$%^&*_~'''
>>> # take input from the user
>>> my_str = input("Enter a string: ")
Enter a string: hello..... This is a encripted (@#$%,,.%) keyword.
>>> # remove punctuation from the string
>>> no_punct = ""
>>> for char in my_str:
    if char not in punctuations:
        no_punct = no_punct + char

>>> # display the unpunctuated string
>>> print(no_punct)
hello This is a encripted keyword
>>> |
```

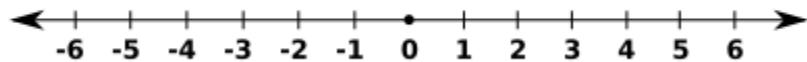
Ln: 35 Col: 4

## Python Program to check if a Number is Positive, Negative or Zero

We can use a Python program to distinguish that if a number is positive, negative or zero.

**Positive Numbers:** A number is known as a positive number if it has a greater value than zero. i.e. 1, 2, 3, 4 etc.

**Negative Numbers:** A number is known as a negative number if it has a lesser value than zero. i.e. -1, -2, -3, -4 etc.



**See this example:**

```
1. num = float(input("Enter a number: "))
2.
3. if num > 0:
4.     print("{0} is a positive number".format(num))
5. elif num == 0:
6.     print("{0} is zero".format(num))
7. else:
8.     print("{0} is negative number".format(num))
```

**Output:**

Python 3.5.2 Shell

```
>>> num = float(input("Enter a number: "))
Enter a number: 15
>>> if num > 0:
    print("{0} is a positive number".format(num))
elif num == 0:
    print("{0} is zero".format(num))
else:
    print("{0} is negative number".format(num))

15.0 is a positive number
>>> num = float(input("Enter a number: "))
Enter a number: -12
>>> if num > 0:
    print("{0} is a positive number".format(num))
elif num == 0:
    print("{0} is zero".format(num))
else:
    print("{0} is negative number".format(num))

-12.0 is negative number
>>> num = float(input("Enter a number: "))
Enter a number: 0
>>> if num > 0:
    print("{0} is a positive number".format(num))
elif num == 0:
    print("{0} is zero".format(num))
else:
    print("{0} is negative number".format(num))

0.0 is zero
>>> |
```

Ln: 35 Col: 4

**Note:**In the above example, elif statement is used. The elif statement is used to check multiple expressions for TRUE and executes a block of code when one of the conditions becomes TRUE.

It is always followed by if statement.

## Python Program to Check if a Number is Odd or Even

### Odd and Even numbers:

If you divide a number by 2 and it gives a remainder of 0 then it is known as even number, otherwise an odd number.

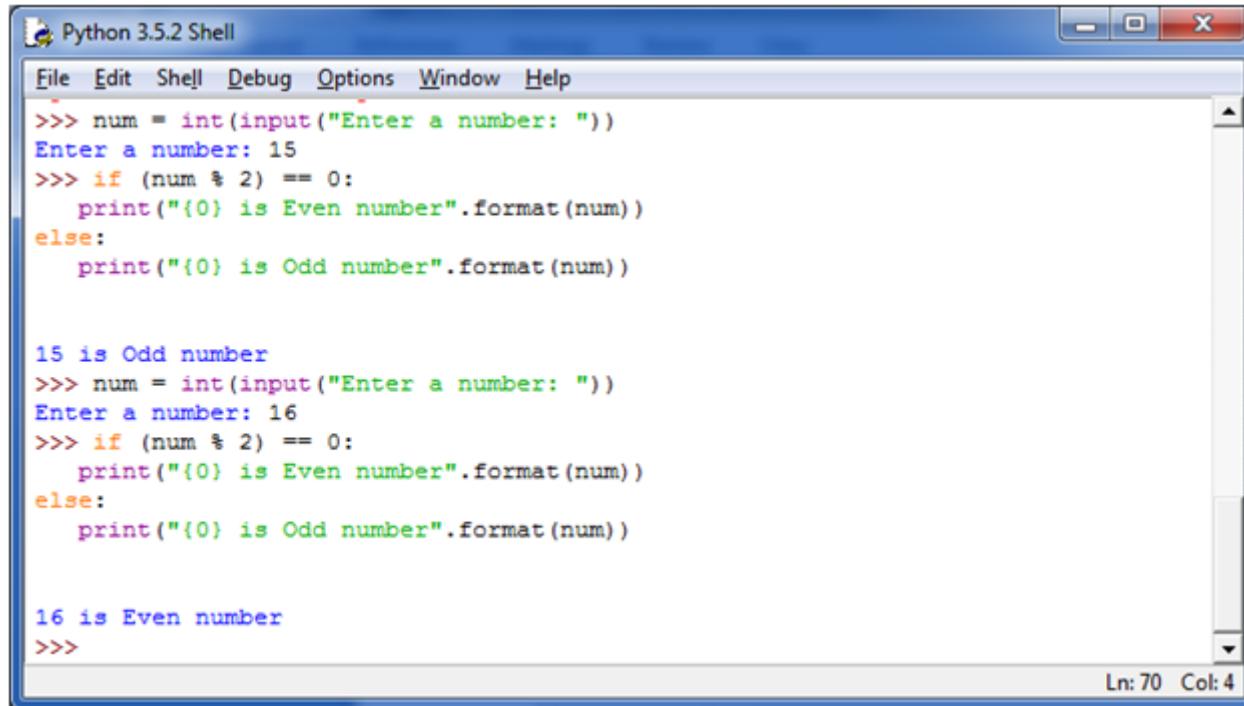
**Even number examples:** 2, 4, 6, 8, 10, etc.

**Odd number examples:**1, 3, 5, 7, 9 etc.

### See this example:

1. num = int(input("Enter a number: "))
2. **if** (num % 2) == 0:
3.     **print**("{0} is Even number".format(num))
4. **else**:
5.     **print**("{0} is Odd number".format(num))

### Output:



The screenshot shows the Python 3.5.2 Shell window. It displays two separate runs of a script. In the first run, a user enters '15' and the program outputs '15 is Odd number'. In the second run, a user enters '16' and the program outputs '16 is Even number'. The shell interface includes a menu bar with File, Edit, Shell, Debug, Options, Window, and Help, and a status bar at the bottom indicating Ln: 70 Col: 4.

```
>>> num = int(input("Enter a number: "))
Enter a number: 15
>>> if (num % 2) == 0:
    print("{0} is Even number".format(num))
else:
    print("{0} is Odd number".format(num))

15 is Odd number
>>> num = int(input("Enter a number: "))
Enter a number: 16
>>> if (num % 2) == 0:
    print("{0} is Even number".format(num))
else:
    print("{0} is Odd number".format(num))

16 is Even number
>>>
```

## Python Program to Check Leap Year

### Leap Year:

A year is called a leap year if it contains an additional day which makes the number of the days in that year is 366. This additional day is added in February which makes it 29 days long.

A leap year occurred once every 4 years.

### How to determine if a year is a leap year?

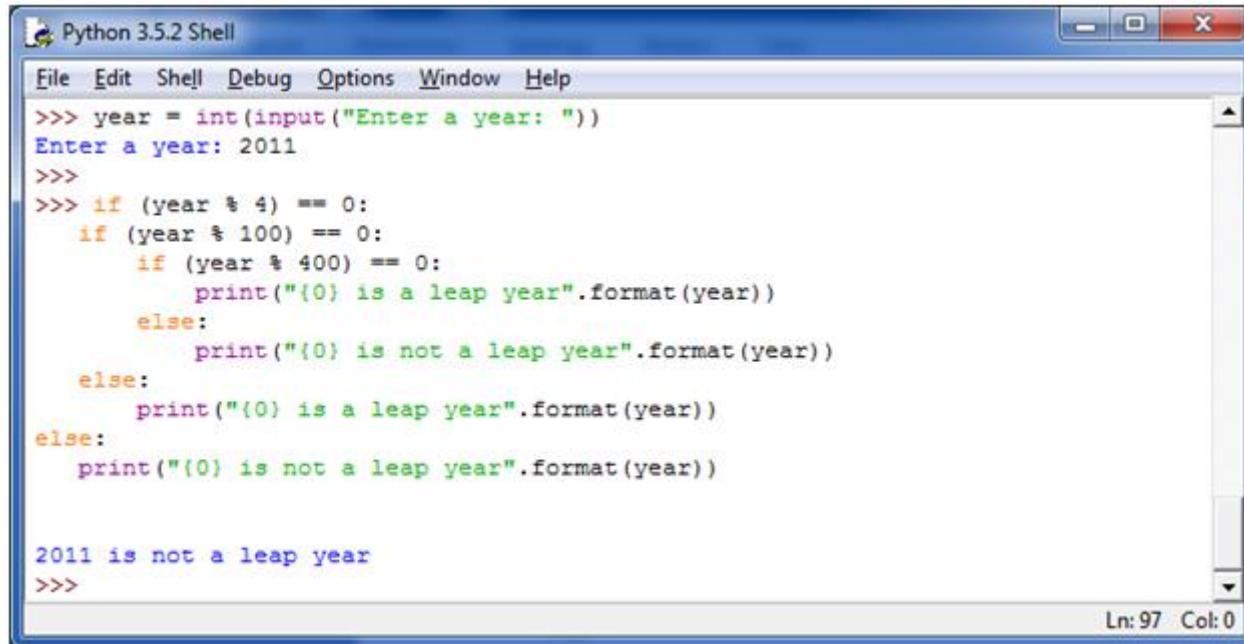
You should follow the following steps to determine whether a year is a leap year or not.

1. If a year is evenly divisible by 4 means having no remainder then go to next step. If it is not divisible by 4. It is not a leap year. For example: 1997 is not a leap year.
2. If a year is divisible by 4, but not by 100. For example: 2012, it is a leap year. If a year is divisible by both 4 and 100, go to next step.
3. If a year is divisible by 100, but not by 400. For example: 1900, then it is not a leap year. If a year is divisible by both, then it is a leap year. So 2000 is a leap year.

#### See this example:

```
1. year = int(input("Enter a year: "))
2. if (year % 4) == 0:
3.     if (year % 100) == 0:
4.         if (year % 400) == 0:
5.             print("{0} is a leap year".format(year))
6.         else:
7.             print("{0} is not a leap year".format(year))
8.     else:
9.         print("{0} is a leap year".format(year))
10. else:
11.     print("{0} is not a leap year".format(year))
```

#### Output:



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window contains the following Python code:

```
>>> year = int(input("Enter a year: "))
Enter a year: 2011
>>>
>>> if (year % 4) == 0:
    if (year % 100) == 0:
        if (year % 400) == 0:
            print("{0} is a leap year".format(year))
        else:
            print("{0} is not a leap year".format(year))
    else:
        print("{0} is a leap year".format(year))
else:
    print("{0} is not a leap year".format(year))

2011 is not a leap year
>>>
```

The status bar at the bottom right indicates "Ln: 97 Col: 0".

## Python Program to Check Prime Number

### Prime numbers:

A prime number is a natural number greater than 1 and having no positive divisor other than 1 and itself.

For example: 3, 7, 11 etc are prime numbers.

### Composite number:

Other natural numbers that are not prime numbers are called composite numbers.

For example: 4, 6, 9 etc. are composite numbers.

**See this example:**

```
1. num = int(input("Enter a number: "))
2.
3. if num > 1:
4.     for i in range(2,num):
5.         if (num % i) == 0:
6.             print(num,"is not a prime number")
7.             print(i,"times",num//i,"is",num)
8.             break
9.     else:
10.        print(num,"is a prime number")
11.
12. else:
13.     print(num,"is not a prime number")
```

**Output:**

Python 3.5.2 Shell

```
File Edit Shell Debug Options Window Help
>>> num = int(input("Enter a number: "))
Enter a number: 313
>>> if num > 1:
    for i in range(2,num):
        if (num % i) == 0:
            print(num,"is not a prime number")
            print(i,"times",num/i,"is",num)
            break
    else:
        print(num,"is a prime number")

else:
    print(num,"is not a prime number")

313 is a prime number
>>> num = int(input("Enter a number: "))
Enter a number: 515
>>> if num > 1:
    for i in range(2,num):
        if (num % i) == 0:
            print(num,"is not a prime number")
            print(i,"times",num/i,"is",num)
            break
    else:
        print(num,"is a prime number")

else:
    print(num,"is not a prime number")

515 is not a prime number
5 times 103 is 515
>>> |
```

Ln: 62 Col: 4

# Python Program to Print all Prime Numbers between an Interval

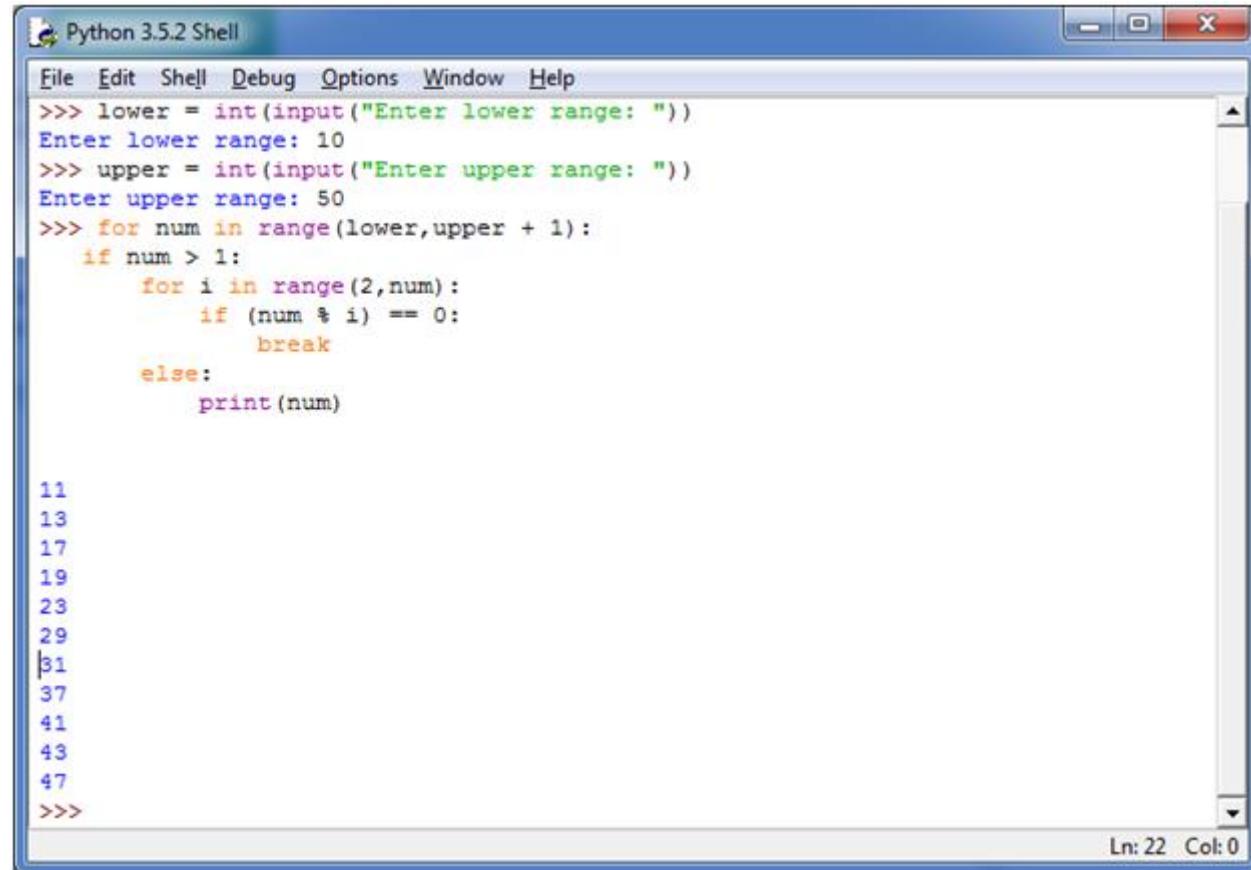
We have already read the concept of prime numbers in the previous program. Here, we are going to print the prime numbers between given interval.

## See this example:

```
1. #Take the input from the user:  
2. lower = int(input("Enter lower range: "))  
3. upper = int(input("Enter upper range: "))  
4.  
5. for num in range(lower,upper + 1):  
6.     if num > 1:  
7.         for i in range(2,num):  
8.             if (num % i) == 0:  
9.                 break  
10.            else:  
11.                print(num)
```

This example will show the prime numbers between 10 and 50.

## Output:



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code:

```
>>> lower = int(input("Enter lower range: "))
Enter lower range: 10
>>> upper = int(input("Enter upper range: "))
Enter upper range: 50
>>> for num in range(lower,upper + 1):
    if num > 1:
        for i in range(2,num):
            if (num % i) == 0:
                break
        else:
            print(num)

11
13
17
19
23
29
31
37
41
43
47
>>>
```

The status bar at the bottom right indicates "Ln: 22 Col: 0".

## Python Program to Find the Factorial of a Number

### What is factorial?

Factorial is a non-negative integer. It is the product of all positive integers less than or equal to that number for which you ask for factorial. It is denoted by exclamation sign (!).

For example:

$$1. \quad 4! = 4 \times 3 \times 2 \times 1 = 24$$

The factorial value of 4 is 24.

**Note:** The factorial value of 0 is 1 always. (Rule violation)

**See this example:**

```
1. num = int(input("Enter a number: "))
2. factorial = 1
3. if num < 0:
4.     print("Sorry, factorial does not exist for negative numbers")
5. elif num == 0:
6.     print("The factorial of 0 is 1")
7. else:
8.     for i in range(1,num + 1):
9.         factorial = factorial*i
10.    print("The factorial of",num,"is",factorial)
```

**The following example displays the factorial of 5 and -5.**

**Output:**

The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains Python code for calculating factorials. It handles negative numbers by printing an error message, handles 0 by printing "The factorial of 0 is 1", and calculates factorials for positive integers using a for loop. The code is run twice: once for num=5 and once for num=-5.

```
>>> num = int(input("Enter a number: "))
Enter a number: 5
>>> factorial = 1
>>> if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
print("The factorial of",num,"is",factorial)

The factorial of 5 is 120
>>> num = int(input("Enter a number: "))
Enter a number: -5
>>> factorial = 1
>>> if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
print("The factorial of",num,"is",factorial)

Sorry, factorial does not exist for negative numbers
>>>
```

[next](#) → ← [prev](#)

## Python Program to Display the multiplication Table

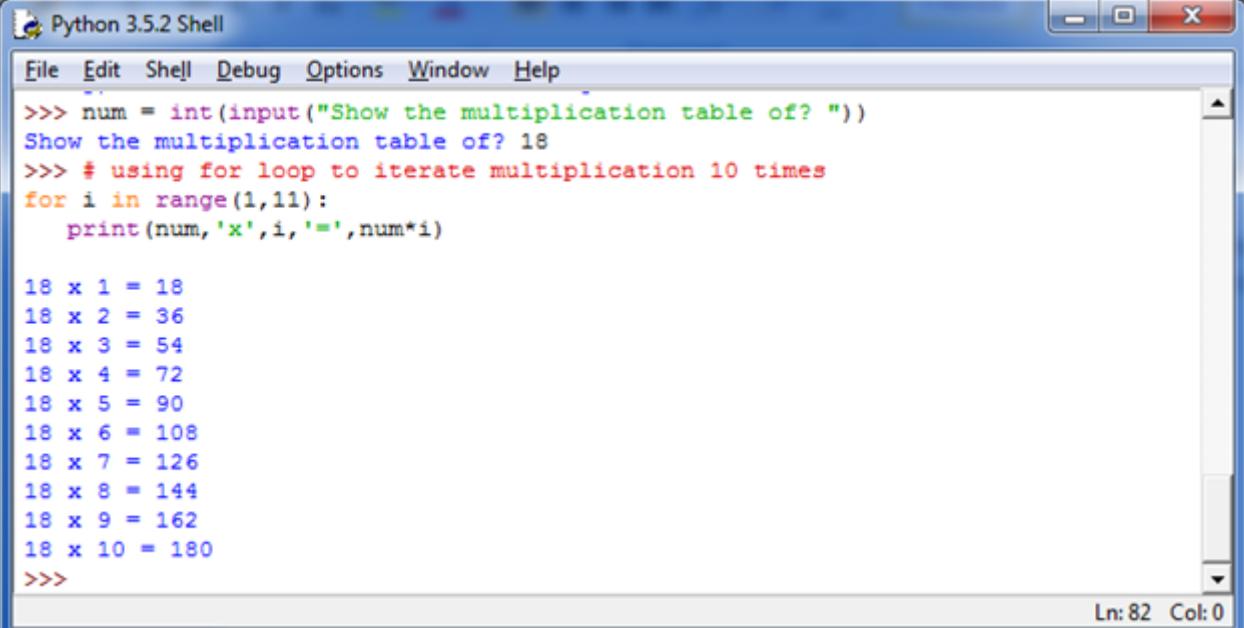
In Python, you can make a program to display the multiplication table of any number. The following program displays the multiplication table (from 1 to 10) according to the user input.

**See this example:**

1. num = int(input("Show the multiplication table of? "))
2. # using for loop to iterate multiplication 10 times
3. for i in range(1,11):
4. print(num,'x',i,'=',num\*i)

The following example shows the multiplication table of 18.

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its output. The code prompts the user for a number and then prints a multiplication table for that number using a for loop. The output shows the multiplication table for 18.

```
>>> num = int(input("Show the multiplication table of? "))
Show the multiplication table of? 18
>>> # using for loop to iterate multiplication 10 times
for i in range(1,11):
    print(num,'x',i,'=',num*i)

18 x 1 = 18
18 x 2 = 36
18 x 3 = 54
18 x 4 = 72
18 x 5 = 90
18 x 6 = 108
18 x 7 = 126
18 x 8 = 144
18 x 9 = 162
18 x 10 = 180
>>>
```

## Python Program to Print the Fibonacci sequence

### Fibonacci sequence:

The Fibonacci sequence specifies a series of numbers where the next number is found by adding up the two numbers just before it.

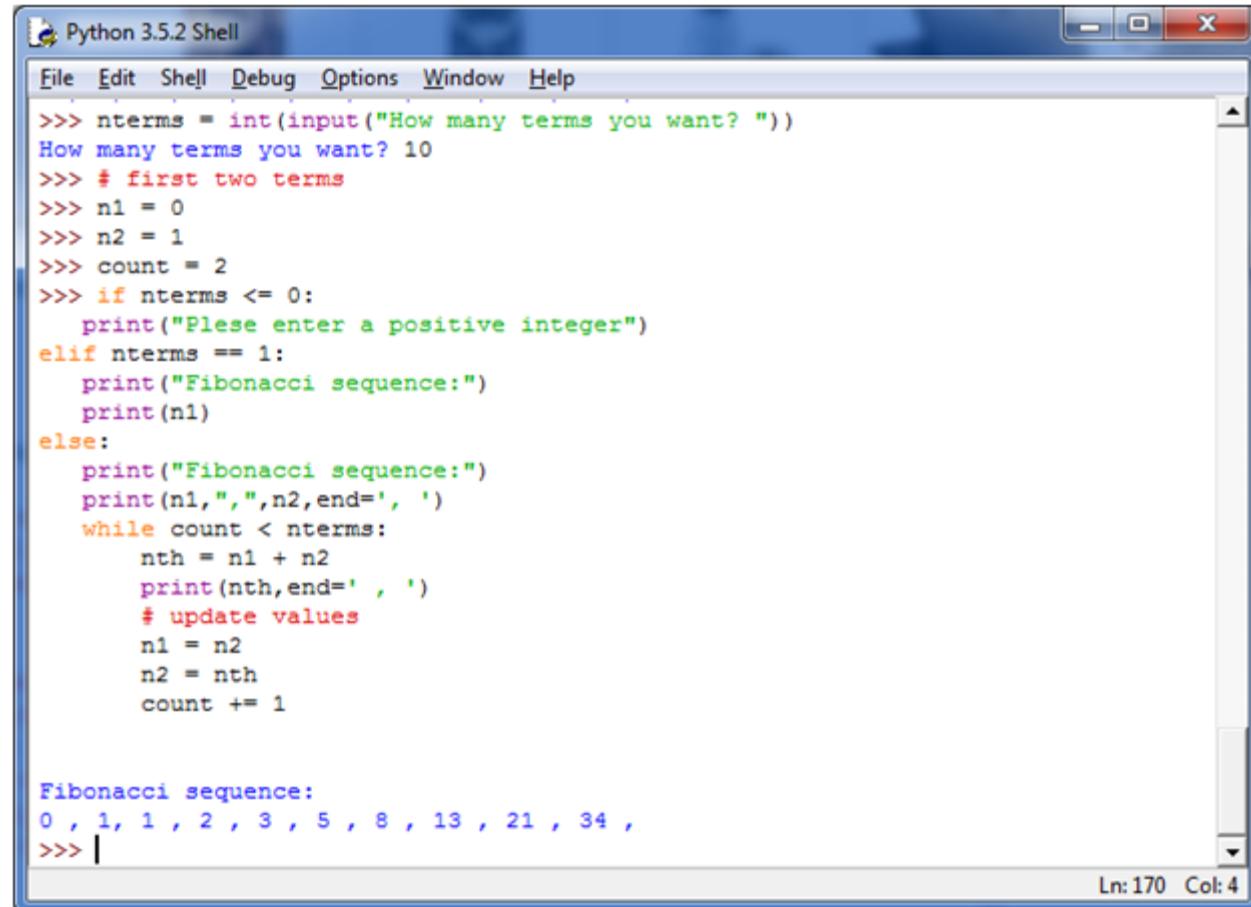
### For example:

- 1.
2. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, **and** so on....

### See this example:

```
1. nterms = int(input("How many terms you want? "))
2. # first two terms
3. n1 = 0
4. n2 = 1
5. count = 2
6. # check if the number of terms is valid
7. if nterms <= 0:
8.     print("Please enter a positive integer")
9. elif nterms == 1:
10.    print("Fibonacci sequence:")
11.    print(n1)
12. else:
13.    print("Fibonacci sequence:")
14.    print(n1,",",n2,end=', ')
15.    while count < nterms:
16.        nth = n1 + n2
17.        print(nth,end=' , ')
18.        # update values
19.        n1 = n2
20.        n2 = nth
21.        count += 1
```

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code in the shell window is as follows:

```
>>> nterms = int(input("How many terms you want? "))
How many terms you want? 10
>>> # first two terms
>>> n1 = 0
>>> n2 = 1
>>> count = 2
>>> if nterms <= 0:
    print("Please enter a positive integer")
elif nterms == 1:
    print("Fibonacci sequence:")
    print(n1)
else:
    print("Fibonacci sequence:")
    print(n1, ", ", n2, end=' , ')
    while count < nterms:
        nth = n1 + n2
        print(nth, end=' , ')
        # update values
        n1 = n2
        n2 = nth
        count += 1

Fibonacci sequence:
0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 ,
>>> |
```

The status bar at the bottom right indicates "Ln: 170 Col: 4".

## Python Program to Check Armstrong Number

### Armstrong number:

A number is called Armstrong number if it is equal to the sum of the cubes of its own digits.

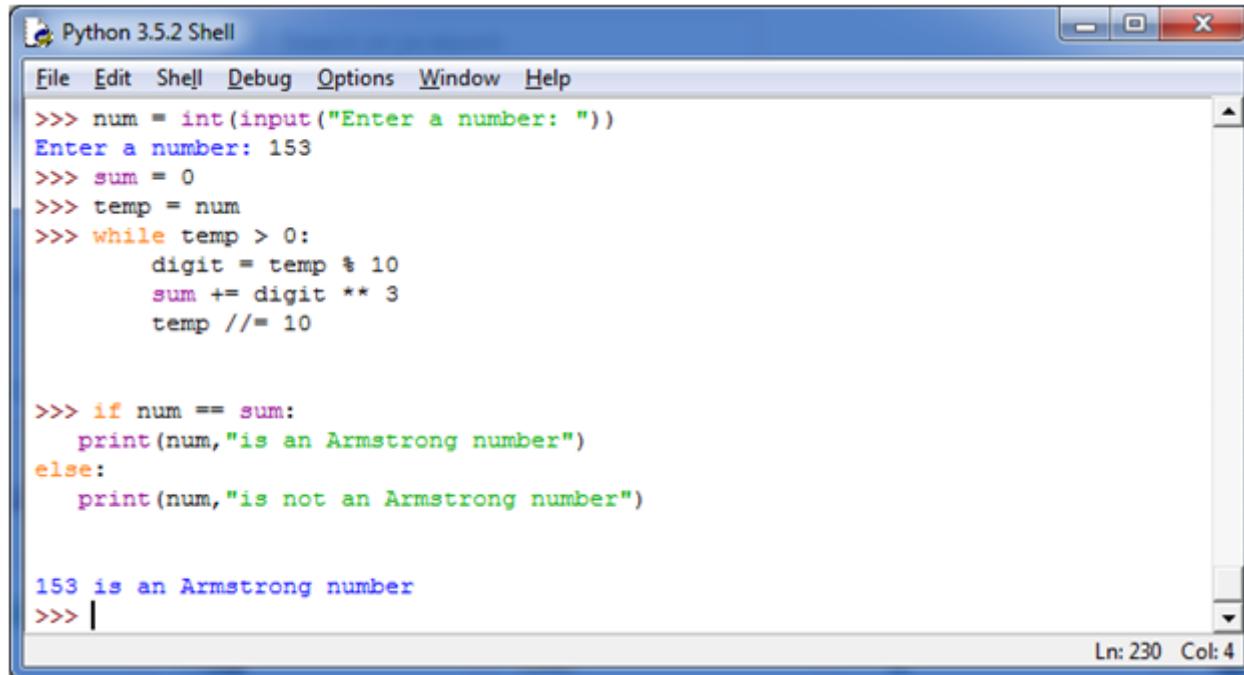
**For example:** 153 is an Armstrong number since  $153 = 1*1*1 + 5*5*5 + 3*3*3$ .

The Armstrong number is also known as **narcissistic** number.

**See this example:**

```
1. num = int(input("Enter a number: "))
2. sum = 0
3. temp = num
4.
5. while temp > 0:
6.     digit = temp % 10
7.     sum += digit ** 3
8.     temp //= 10
9.
10. if num == sum:
11.     print(num,"is an Armstrong number")
12. else:
13.     print(num,"is not an Armstrong number")
```

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The code is as follows:

```
>>> num = int(input("Enter a number: "))
Enter a number: 153
>>> sum = 0
>>> temp = num
>>> while temp > 0:
    digit = temp % 10
    sum += digit ** 3
    temp //= 10

>>> if num == sum:
    print(num, "is an Armstrong number")
else:
    print(num, "is not an Armstrong number")

153 is an Armstrong number
>>> |
```

The output shows that 153 is an Armstrong number.

## Python Program to Find Armstrong Number between an Interval

We have already read the concept of Armstrong numbers in the previous program. Here, we print the Armstrong numbers within a specific given interval.

### See this example:

1. lower = int(input("Enter lower range: "))
2. upper = int(input("Enter upper range: "))
- 3.
4. **for** num **in** range(lower,upper + 1):
5. sum = 0

```
6.     temp = num
7.     while temp > 0:
8.         digit = temp % 10
9.         sum += digit ** 3
10.        temp //= 10
11.        if num == sum:
12.            print(num)
```

This example shows all Armstrong numbers between 100 and 500.

**Output:**

The screenshot shows the Python 3.5.2 Shell window. The code prompts the user for a lower and upper range, then iterates through each number in that range. For each number, it initializes a sum to 0, then loops through its digits, cubing each and adding it to the sum. If the sum equals the original number, it prints the number. The output shows several narcissistic numbers: 125, 153, 216, 370, 371, and 407.

```
>>> lower = int(input("Enter lower range: "))
Enter lower range: 100
>>> upper = int(input("Enter upper range: "))
Enter upper range: 500
>>> for num in range(lower,upper + 1):
    sum = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        sum += digit ** 3
        temp //= 10
    if num == sum:
        print(num)

125
153
216
370
371
407
>>> |
```

## Python Program to Find the Sum of Natural Numbers

### Natural numbers:

As the name specifies, a natural number is the number that occurs commonly and obviously in the nature. It is a whole, non-negative number.

Some mathematicians think that a natural number must contain 0 and some don't believe this theory. So, a list of natural number can be defined as:

- 1.
2. N= {0, 1, 2, 3, 4, .... **and** so on}
3. N= {1, 2, 3, 4, .... **and** so on}

**See this example:**

```
1. num = int(input("Enter a number: "))
2.
3. if num < 0:
4.     print("Enter a positive number")
5. else:
6.     sum = 0
7.
8.     # use while loop to iterate un till zero
9.     while(num > 0):
10.         sum += num
11.         num -= 1
12.     print("The sum is",sum)
```

This example shows the sum of the first 100 positive numbers (0-100)

**Output:**

The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python script:

```
>>> num = int(input("Enter a number: "))
Enter a number: 100
>>> if num < 0:
    print("Enter a positive number")
else:
    sum = 0
    # use while loop to iterate un till zero
    while(num > 0):
        sum += num
        num -= 1
    print("The sum is",sum)

The sum is 5050
>>>
```

The output window shows the result "The sum is 5050". The status bar at the bottom right indicates "Ln: 31 Col: 4".

## Python Program to Find LCM

### LCM: Least Common Multiple/ Lowest Common Multiple

LCM stands for Least Common Multiple. It is a concept of arithmetic and number system. The LCM of two integers a and b is denoted by  $\text{LCM}(a,b)$ . It is the smallest positive integer that is divisible by both "a" and "b".

**For example:** We have two integers 4 and 6. Let's find LCM

**Multiples of 4 are:**

1. 4, 8, 12, 16, 20, 24, 28, 32, 36,... **and** so on...

**Multiples of 6 are:**

1. 6, 12, 18, 24, 30, 36, 42,... **and** so on....

**Common multiples of 4 and 6 are simply the numbers that are in both lists:**

1. 12, 24, 36, 48, 60, 72,... **and** so on....

LCM is the lowest common multiplier so it is 12.

**See this example:**

```
1. def lcm(x, y):  
2.     if x > y:  
3.         greater = x  
4.     else:  
5.         greater = y  
6.     while(True):  
7.         if((greater % x == 0) and (greater % y == 0)):  
8.             lcm = greater  
9.             break  
10.            greater += 1  
11.        return lcm  
12.  
13.  
14. num1 = int(input("Enter first number: "))  
15. num2 = int(input("Enter second number: "))  
16. print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
```

The following example will show the LCM of 12 and 20 (according to the user input)

**Output:**

The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code area contains a function definition for LCM and its execution:

```
>>> def lcm(x, y):
    if x > y:
        greater = x
    else:
        greater = y

    while(True):
        if((greater % x == 0) and (greater % y == 0)):
            lcm = greater
            break
        greater += 1

    return lcm

>>> num1 = int(input("Enter first number: "))
Enter first number: 12
>>> num2 = int(input("Enter second number: "))
Enter second number: 20
>>> print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
The L.C.M. of 12 and 20 is 60
>>> |
```

The status bar at the bottom right indicates Ln: 117 Col: 4.

## Python Program to Find HCF

### HCF: Highest Common Factor

Highest Common Factor or Greatest Common Divisor of two or more integers when at least one of them is not zero is the largest positive integer that evenly divides the numbers without a remainder. For example, the GCD of 8 and 12 is 4.

#### For example:

We have two integers 8 and 12. Let's find the HCF.

The divisors of 8 are:

1. 1, 2, 4, 8

The divisors of 12 are:

1. 1, 2, 3, 4, 6, 12

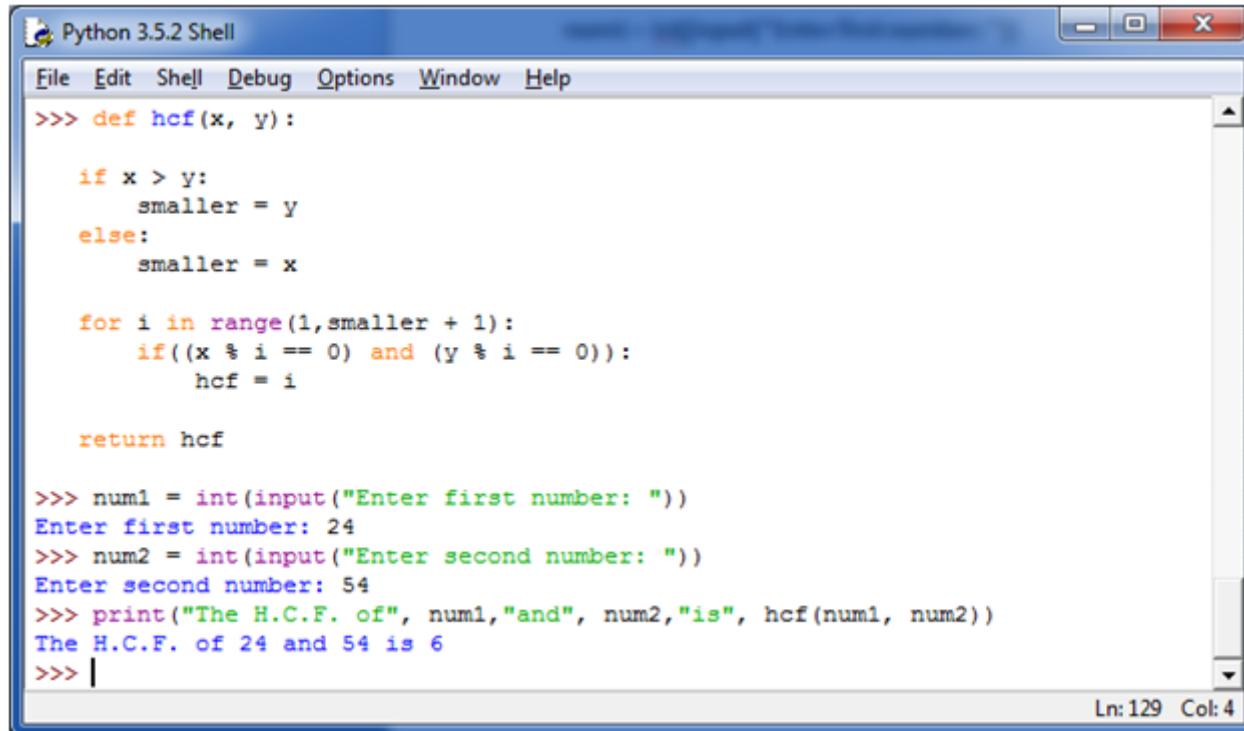
HCF /GCD is the greatest common divisor. So HCF of 8 and 12 are 4.

**See this example:**

```
1. def hcf(x, y):  
2.     if x > y:  
3.         smaller = y  
4.     else:  
5.         smaller = x  
6.     for i in range(1,smaller + 1):  
7.         if((x % i == 0) and (y % i == 0)):  
8.             hcf = i  
9.     return hcf  
10.  
11.num1 = int(input("Enter first number: "))  
12.num2 = int(input("Enter second number: "))  
13.print("The H.C.F. of", num1,"and", num2,"is", hcf(num1, num2))
```

The following example shows the HCF of 24 and 54. (according to user input)

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains Python code and its execution results:

```
>>> def hcf(x, y):
    if x > y:
        smaller = y
    else:
        smaller = x

    for i in range(1,smaller + 1):
        if((x % i == 0) and (y % i == 0)):
            hcf = i

    return hcf

>>> num1 = int(input("Enter first number: "))
Enter first number: 24
>>> num2 = int(input("Enter second number: "))
Enter second number: 54
>>> print("The H.C.F. of", num1,"and", num2,"is", hcf(num1, num2))
The H.C.F. of 24 and 54 is 6
>>> |
```

The status bar at the bottom right indicates "Ln: 129 Col: 4".

## Python Program to Convert Decimal to Binary, Octal and Hexadecimal

**Decimal System:** The most widely used number system is decimal system. This system is base 10 number system. In this system, ten numbers (0-9) are used to represent a number.

**Binary System:** Binary system is base 2 number system. Binary system is used because computers only understand binary numbers (0 and 1).

**Octal System:** Octal system is base 8 number system.

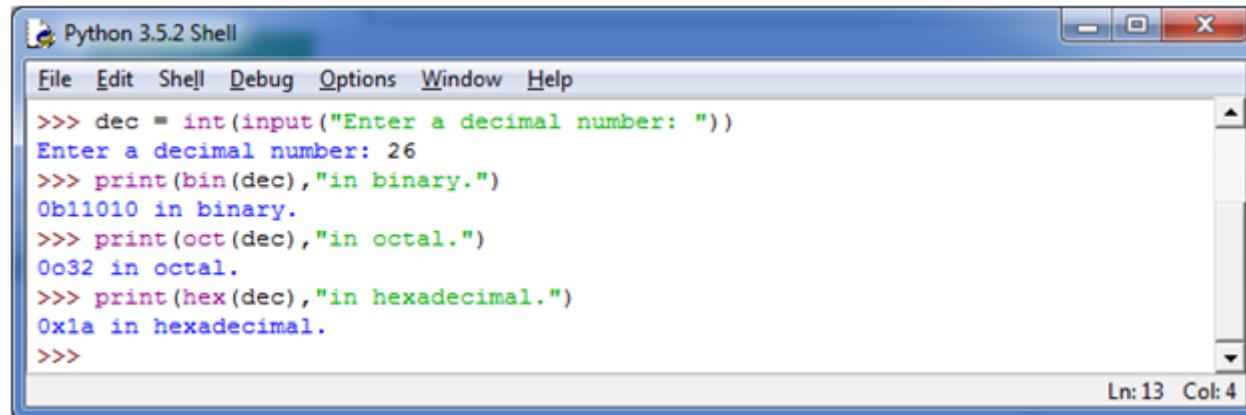
**Hexadecimal System:** Hexadecimal system is base 16 number system.

This program is written to convert decimal to binary, octal and hexadecimal.

**See this example:**

1. dec = int(input("Enter a decimal number: "))
- 2.
3. **print(bin(dec),"in binary.")**
4. **print(oct(dec),"in octal.")**
5. **print(hex(dec),"in hexadecimal."**

**Output:**



The screenshot shows the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The command line shows the following code execution:

```
>>> dec = int(input("Enter a decimal number: "))
Enter a decimal number: 26
>>> print(bin(dec),"in binary.")
0b11010 in binary.
>>> print(oct(dec),"in octal.")
0o32 in octal.
>>> print(hex(dec),"in hexadecimal.")
0x1a in hexadecimal.
>>>
```

The status bar at the bottom right indicates Ln: 13 Col: 4.

## Python Program To Find ASCII value of a character

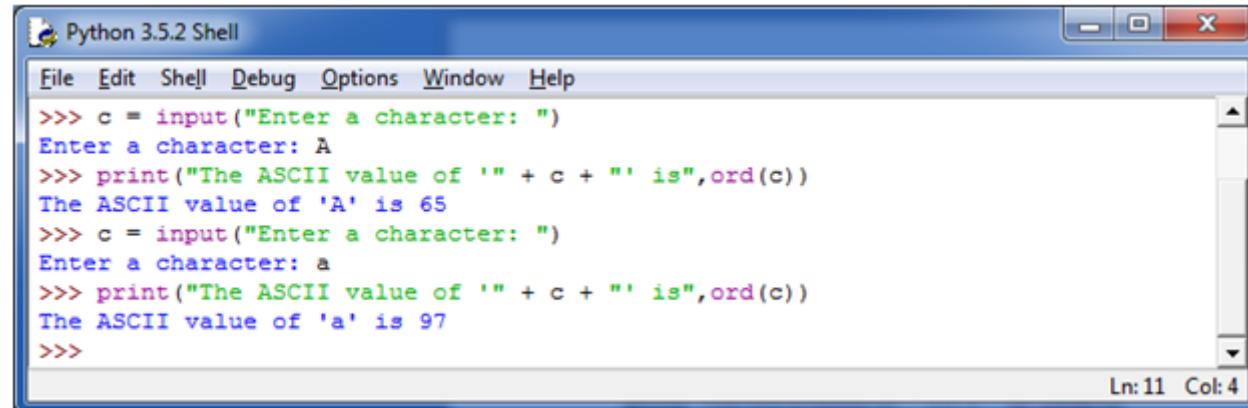
**ASCII:** ASCII is an acronym stands for **American Standard Code for Information Interchange**. In ASCII, a specific numerical value is given to different characters and symbols, for computers to store and manipulate.

It is case sensitive. Same character, having different format (upper case and lower case) has different value. For example: The ASCII value of "A" is 65 while the ASCII value of "a" is 97.

**See this example:**

1. `c = input("Enter a character: ")`
- 2.
3. `print("The ASCII value of '" + c + "' is",ord(c))`

**Output:**



A screenshot of the Python 3.5.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the following interaction:

```
>>> c = input("Enter a character: ")
Enter a character: A
>>> print("The ASCII value of '" + c + "' is",ord(c))
The ASCII value of 'A' is 65
>>> c = input("Enter a character: ")
Enter a character: a
>>> print("The ASCII value of '" + c + "' is",ord(c))
The ASCII value of 'a' is 97
>>>
```

The status bar at the bottom right indicates Ln: 11 Col: 4.

## Python Program to Make a Simple Calculator

In Python, you can create a simple calculator, displaying the different arithmetical operations i.e. addition, subtraction, multiplication and division.

The following program is intended to write a simple calculator in Python:

**See this example:**

```
1. # define functions
2. def add(x, y):
3.     """This function adds two numbers"""
4.     return x + y
5. def subtract(x, y):
6.     """This function subtracts two numbers"""
7.     return x - y
8. def multiply(x, y):
9.     """This function multiplies two numbers"""
10.    return x * y
11. def divide(x, y):
12.    """This function divides two numbers"""
13.    return x / y
14. # take input from the user
15. print("Select operation.")
16. print("1.Add")
17. print("2.Subtract")
18. print("3.Multiply")
19. print("4.Divide")
20.
21. choice = input("Enter choice(1/2/3/4):")
22.
23. num1 = int(input("Enter first number: "))
24. num2 = int(input("Enter second number: "))
25.
26. if choice == '1':
27.     print(num1,"+",num2,"=", add(num1,num2))
28.
```

```
29. elif choice == '2':  
30.     print(num1,"-",num2,"=", subtract(num1,num2))  
31.  
32. elif choice == '3':  
33.     print(num1,"*",num2,"=", multiply(num1,num2))  
34. elif choice == '4':  
35.     print(num1,"/",num2,"=", divide(num1,num2))  
36. else:  
37.     print("Invalid input")
```

**Output:**

Python 3.5.2 Shell

```
File Edit Shell Debug Options Window Help
>>> # define functions
def add(x, y):
    """This function adds two numbers"""

    return x + y

>>> def subtract(x, y):
    """This function subtracts two numbers"""

    return x - y

>>> def multiply(x, y):
    """This function multiplies two numbers"""

Ln: 52 Col: 0
```

Python 3.5.2 Shell

```
File Edit Shell Debug Options Window Help
return x * y

>>> def divide(x, y):
    """This function divides two numbers"""

| return x / y

>>> # take input from the user
>>> print("Select operation.")
Select operation.
>>> print("1.Add")
1.Add
>>> print("2.Subtract")
2.Subtract
>>> print("3.Multiply")
3.Multiply
>>> print("4.Divide")
4.Divide
>>> choice = input("Enter choice(1/2/3/4):")
Enter choice(1/2/3/4):3
>>> num1 = int(input("Enter first number: "))
Enter first number: 50
>>> num2 = int(input("Enter second number: "))
Enter second number: 30
>>> if choice == '1':
    print(num1,"+",num2,"=", add(num1,num2))

elif choice == '2':
    print(num1,"-",num2,"=", subtract(num1,num2))

elif choice == '3':
    print(num1,"*",num2,"=", multiply(num1,num2))

elif choice == '4':
    print(num1,"/",num2,"=", divide(num1,num2))
else:
    print("Invalid input")

50 * 30 = 1500
```

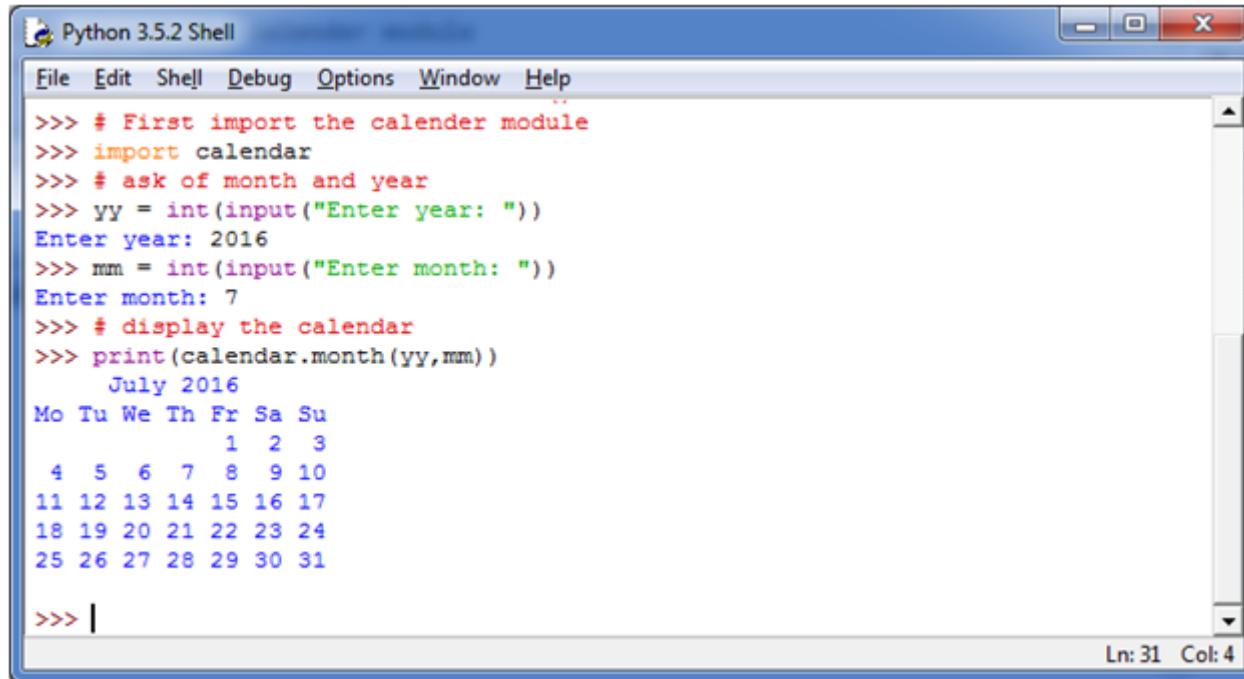
# Python Function to Display Calendar

In Python, we can display the calendar of any month of any year by importing the calendar module.

**See this example:**

1. `# First import the calendar module`
2. `import calendar`
3. `# ask of month and year`
4. `yy = int(input("Enter year: "))`
5. `mm = int(input("Enter month: "))`
6. `# display the calendar`
7. `print(calendar.month(yy,mm))`

**Output:**



The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code and its output:

```
>>> # First import the calender module
>>> import calendar
>>> # ask of month and year
>>> yy = int(input("Enter year: "))
Enter year: 2016
>>> mm = int(input("Enter month: "))
Enter month: 7
>>> # display the calendar
>>> print(calendar.month(yy,mm))
    July 2016
Mo Tu We Th Fr Sa Su
        1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

The cursor is at the bottom left, and the status bar at the bottom right shows "Ln: 31 Col: 4".

## Python Program to Display Fibonacci Sequence Using Recursion

### Fibonacci sequence:

A Fibonacci sequence is a sequence of integers which first two terms are 0 and 1 and all other terms of the sequence are obtained by adding their preceding two numbers.

For example: 0, 1, 1, 2, 3, 5, 8, 13 and so on...

### See this example:

1. `def recur_fibo(n):`

```
2. if n <= 1:
3.     return n
4. else:
5.     return(recur_fibo(n-1) + recur_fibo(n-2))
6. # take input from the user
7. nterms = int(input("How many terms? "))
8. # check if the number of terms is valid
9. if nterms <= 0:
10.    print("Please enter a positive integer")
11. else:
12.    print("Fibonacci sequence:")
13.    for i in range(nterms):
14.        print(recur_fibo(i))
```

**Output:**

The screenshot shows a Windows-style window titled "Python 3.5.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains Python code for generating a Fibonacci sequence. The code defines a recursive function `recur\_fibo` and prompts the user for the number of terms. It then prints the sequence up to the specified term. The output shows the first 12 terms of the Fibonacci sequence.

```
>>> def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

>>> # take input from the user
>>> nterms = int(input("How many terms? "))
How many terms? 12
>>> # validate the number of terms
>>> if nterms <= 0:
    print("Please enter a positive integer")
else:
    print("Fibonacci sequence:")
    for i in range(nterms):
        print(recur_fibo(i))

Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
55
89
>>> |
```

## Python Program to Find Factorial of Number Using Recursion

**Factorial:** Factorial of a number specifies a product of all integers from 1 to that number. It is defined by the symbol explanation mark (!).

For example: The factorial of 5 is denoted as  $5! = 1*2*3*4*5 = 120$ .

### See this example:

```
1. def recur_factorial(n):
2.     if n == 1:
3.         return n
4.     else:
5.         return n*recur_factorial(n-1)
6. # take input from the user
7. num = int(input("Enter a number: "))
8. # check is the number is negative
9. if num < 0:
10.    print("Sorry, factorial does not exist for negative numbers")
11. elif num == 0:
12.    print("The factorial of 0 is 1")
13. else:
14.    print("The factorial of",num,"is",recur_factorial(num))
```

### Output:

Python 3.5.2 Shell

```
File Edit Shell Debug Options Window Help
>>> def recur_factorial(n):
    if n == 1:
        return n
    else:
        return n*recur_factorial(n-1)

>>> # take input from the user
>>> num = int(input("Enter a number: "))
Enter a number: 6
>>> if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of",num,"is",recur_factorial(num))

The factorial of 6 is 720
>>> #check the second condition by taking negative number.
>>> num = int(input("Enter a number: "))
Enter a number: -5
>>> if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of",num,"is",recur_factorial(num))

Sorry, factorial does not exist for negative numbers
>>>
```

Ln: 53 Col: 0