

Python Tutorial

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.

Introduction to Python

It covers the topics such as python programming, features, history, versions, how to install, example, how to execute, variables, keywords, identifiers, literals, operators and comments.

Control Statement

The control statement in python covers if statement, for loop, while loop, do while loop, break statement, continue statement and pass statement.

Python Strings

The string chapter in python provides the full functionality to work on strings such as accessing string, applying string operators, details of slice notation, applying different functions etc.

Python Lists

The list chapter in python covers the data structure part such as storing data in list, accessing data, manipulating data etc.

Python Tuples

A sequence of immutable objects is known as tuple, it covers accessing tuple, adding tuple, replacating tuple, updating tuple etc.

Python Dictionary

The python dictionary provides details about dictionary operations.

Python Functions

It provides a list of python functions with its implementations.

Python Files I/O

How to write data into file and read data from file in python?

Python Modules

What is python module? What are the usage of modules?

Python Exceptions

It explains the errors and exceptions in python.

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 History

- 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 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 Version

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
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 Applications Area

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.

9) Enterprise Applications

Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

10) Applications for Images

Using Python several application can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

There are several such applications which can be developed using Python

HOW TO INSTALL PYTHON

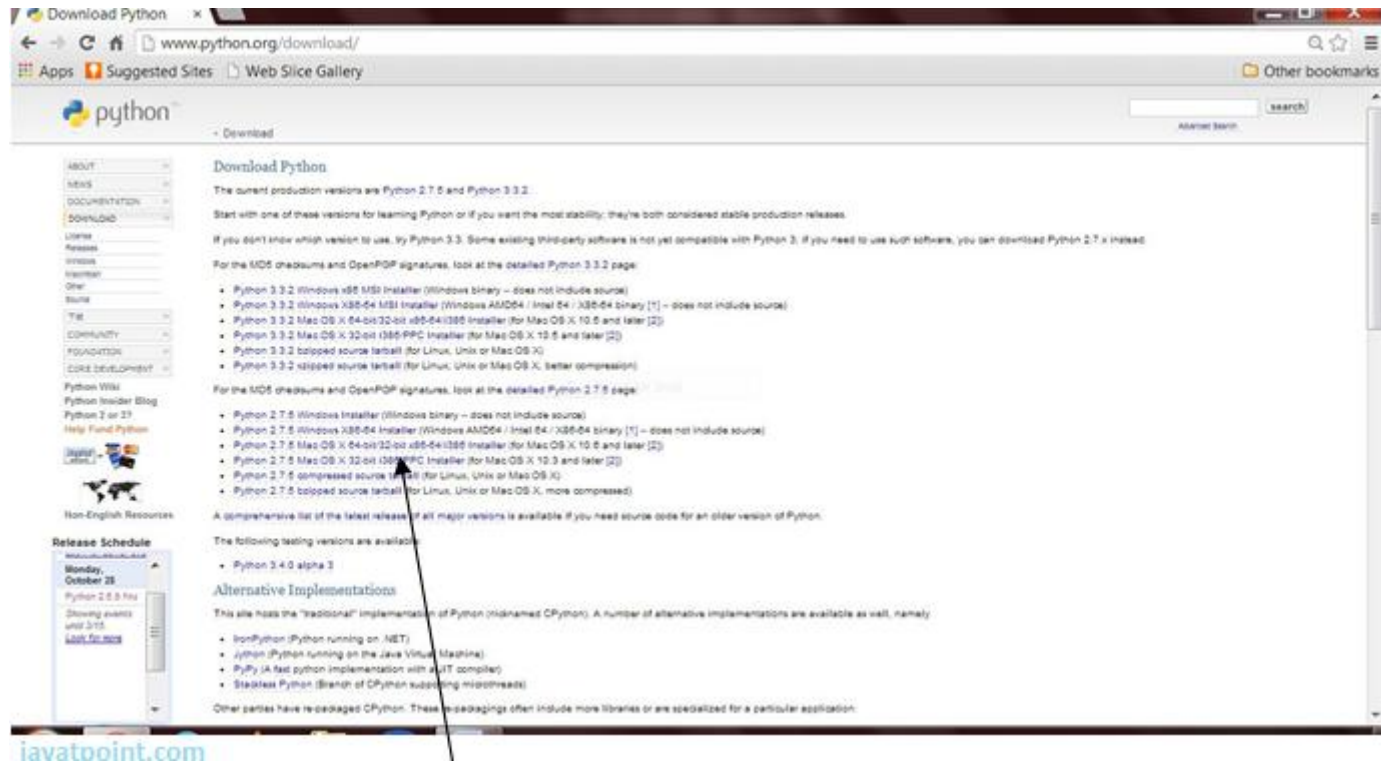
To start with Python, first make sure that the Python is installed on local computer.

To install Python, visit the [official site](https://www.python.org/) and download Python from the download section.

To install **Python on Ubuntu** operating system, visit our [installation section](#) where we have provided detailed installation process.

For **Windows operating system**, the installation process is given below.

1. To install Python, firstly download the Python distribution from www.python.org/download.



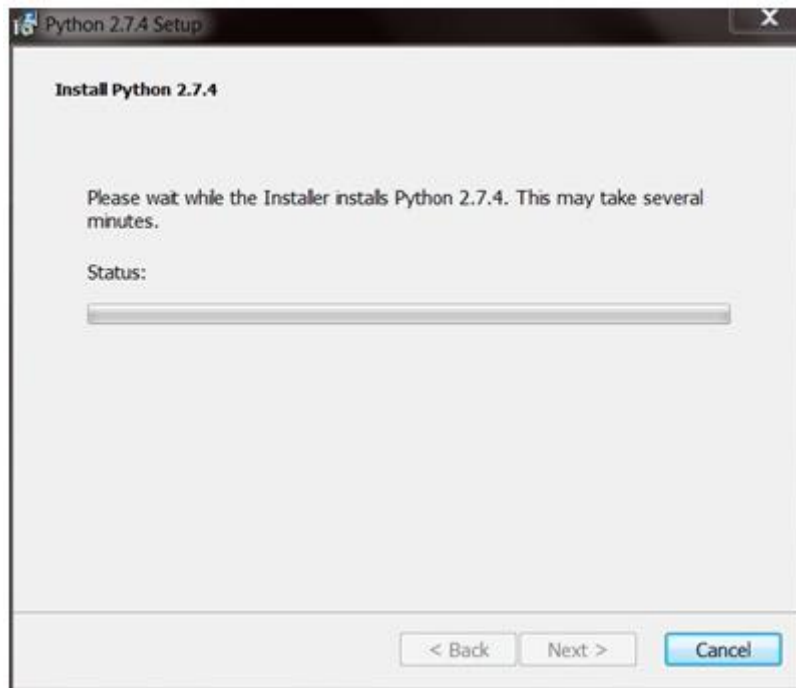
2. After downloading the Python distribution, double click on the downloaded software to execute it. Follow the following installation steps.



iavatpoint.com



iavatpoint.com



Click the Finish button and Python will be installed on your system.

SETTING PATH IN PYTHON

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

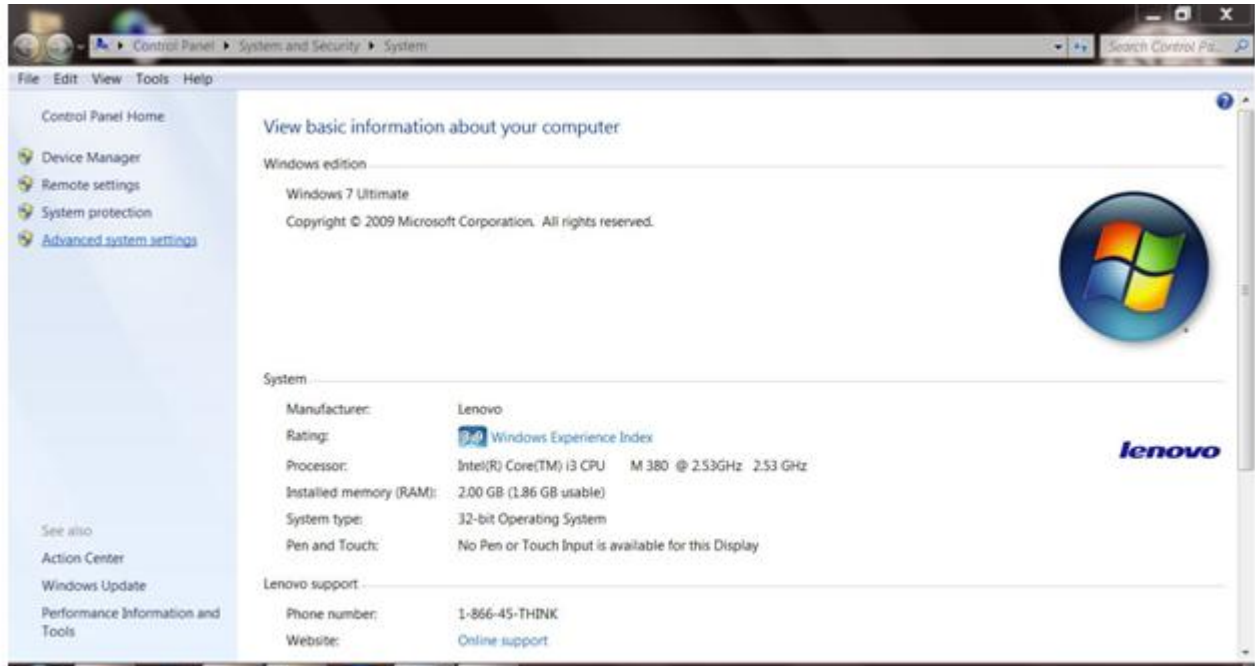
- Your Python program and executable code can reside in any directory of your system, therefore Operating System provides a specific search path that index the directories Operating System should search for executable code.
- The Path is set in the Environment Variable of My Computer properties:
- To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

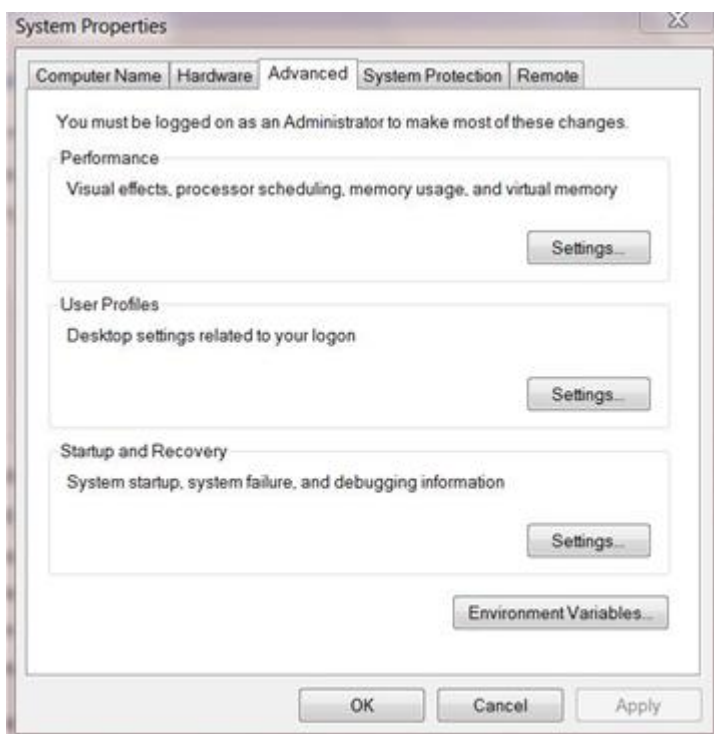
Path will be set for executing Python programs.

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



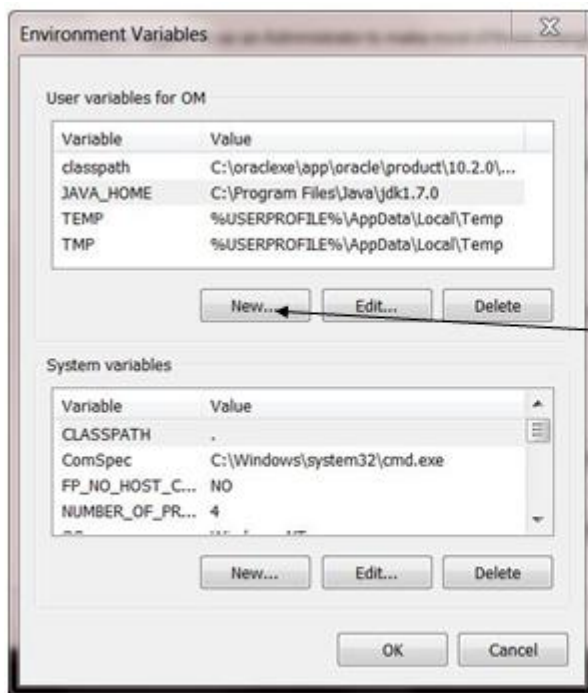
javatpoint.com

3. Click on Environment Variable tab.



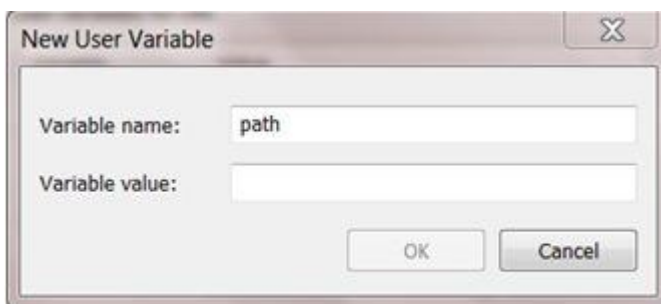
javatpoint.com

4. Click on new tab of user variables.



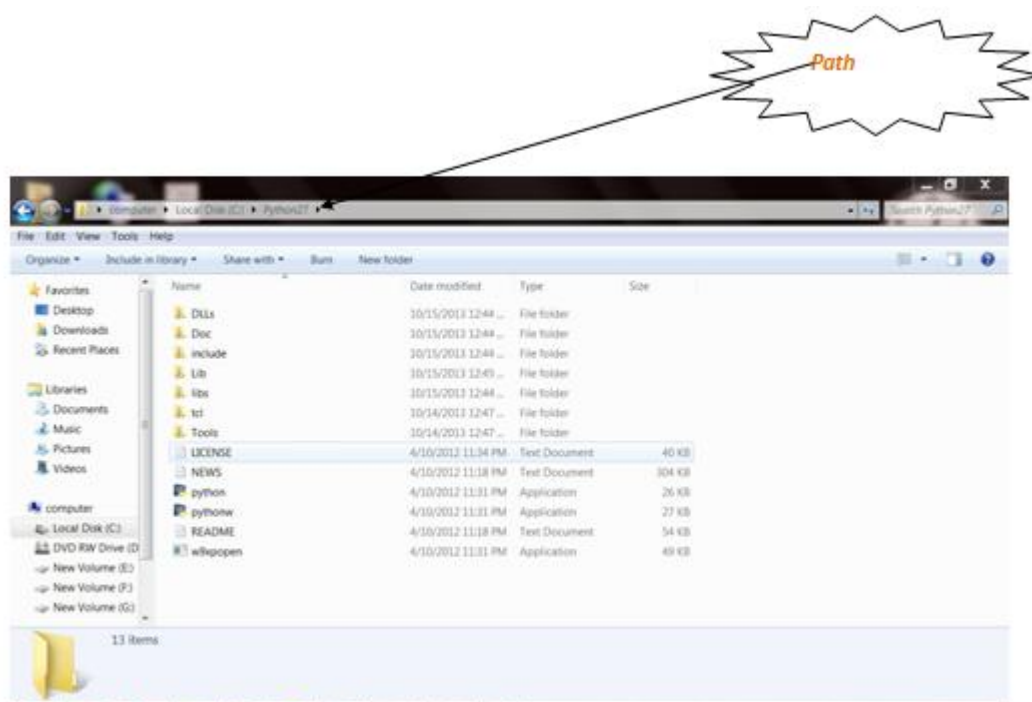
javatpoint.com

5. Write path in variable name



javatpoint.com

6. Copy the path of Python folder



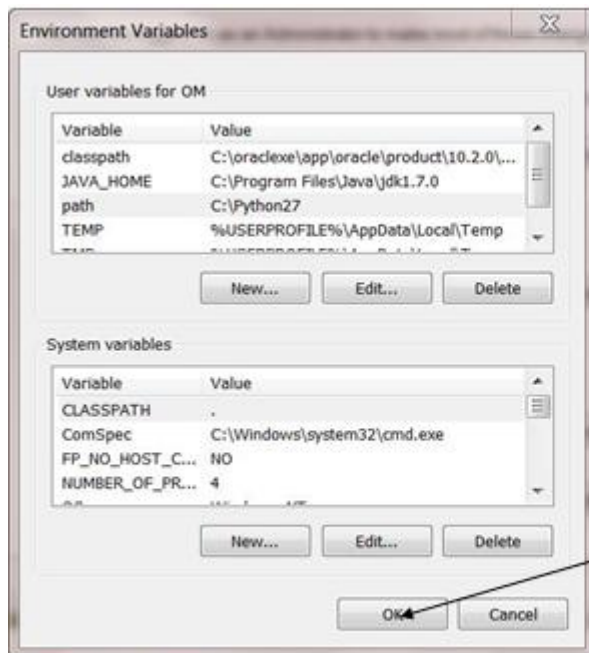
javatpoint.com

7. Paste path of Python in variable value.



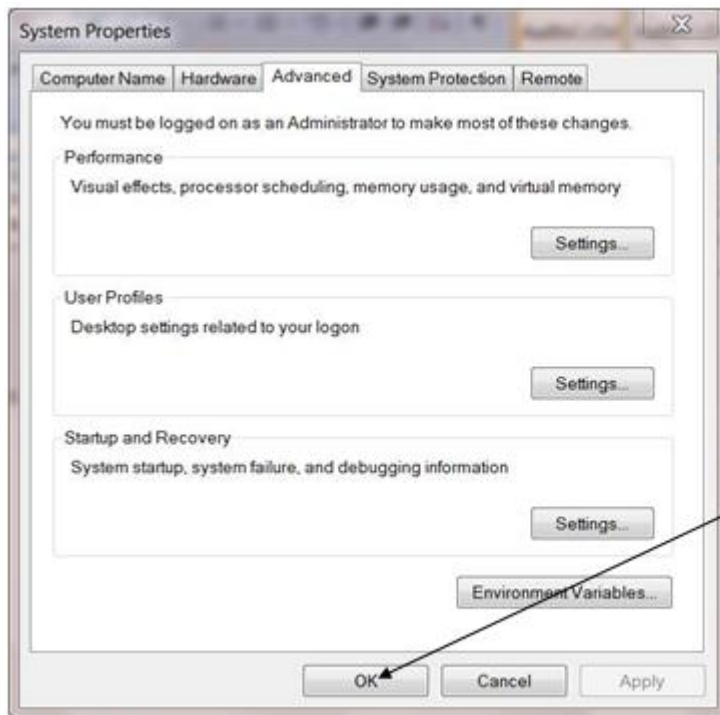
javatpoint.com

8. Click on Ok button:



javatpoint.com

9. Click on Ok button:



OK button

Python Example

Python is easy to learn and code and can be execute with python interpreter. We can also use Python interactive shell to test python code immediately.

A simple hello world example is given below. Write below code in a file and save with **.py** extension. Python source file has **.py** extension.

hello.py

1. `print("hello world by python!")`

Execute this example by using following command.

1. `Python3 hello.py`

After executing, it produces the following output to the screen.

Output

```
hello world by python!
```

Python Example using Interactive Shell

Python interactive shell is used to test the code immediately and does not require to write and save code in file.

Python code is simple and easy to run. Here is a simple Python code that will print "Welcome to Python".

A simple python example is given below.

1. `>>> a="Welcome To Python"`
2. `>>> print a`
3. Welcome To Python
4. `>>>`

Explanation:

- Here we are using IDLE to write the Python code. Detail explanation to run code is given in Execute Python section.
- A variable is defined named "a" which holds "Welcome To Python".
- "print" statement is used to print the content. Therefore "print a" statement will print the content of the variable. Therefore, the output "Welcome To Python" is produced.

Python 3.4 Example

In python 3.4 version, you need to add parenthesis () in a string code to print it.

1. `>>> a=("Welcome To Python Example")`
2. `>>> print a`
3. Welcome To Python Example
4. `>>>`

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 immediately and produce output instantly. To get into this shell, write python in the command prompt and start working with Python.

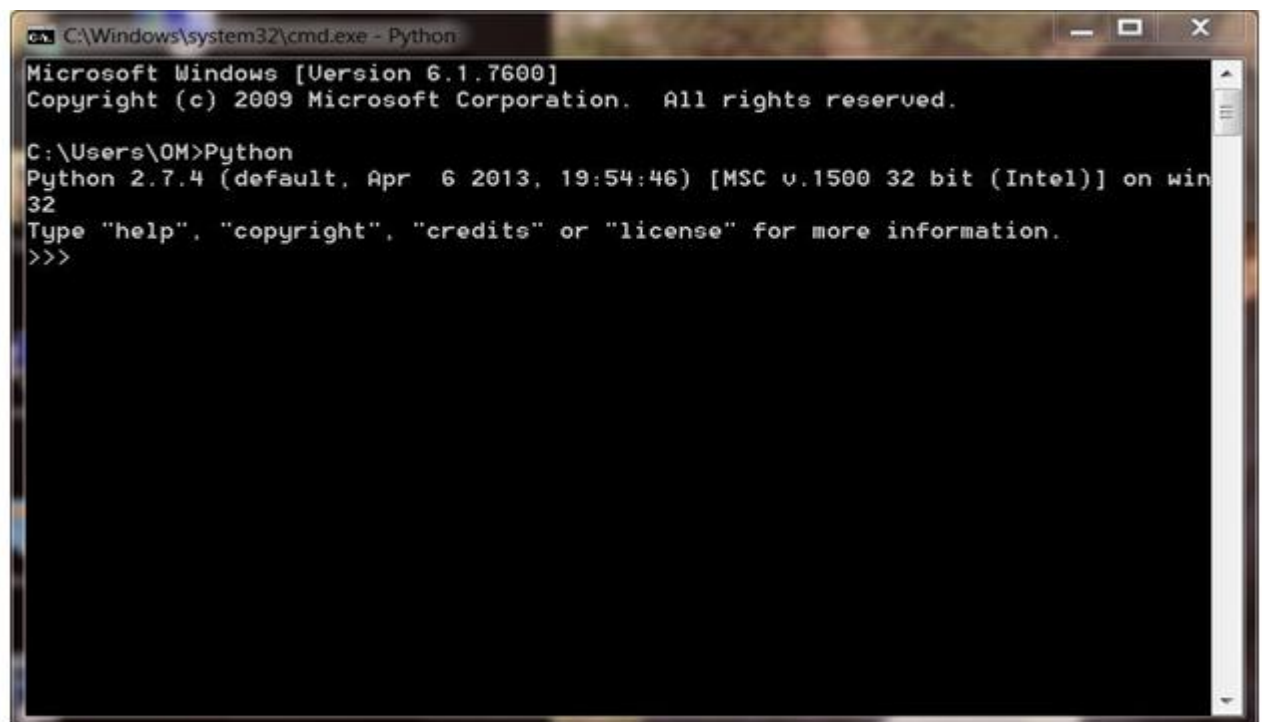


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>Python
```

javatpoint.cpm

Press Enter key and the Command Prompt will appear like:



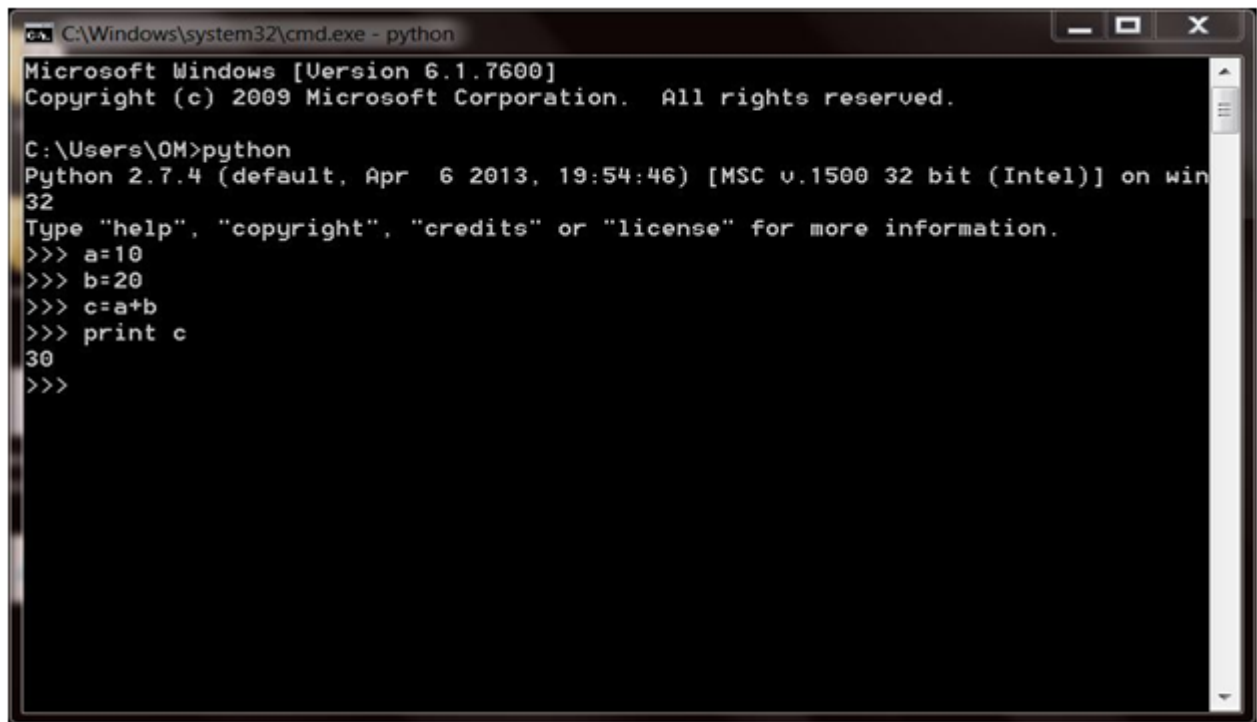
```
C:\Windows\system32\cmd.exe - Python
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

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

Now we can execute our Python commands.

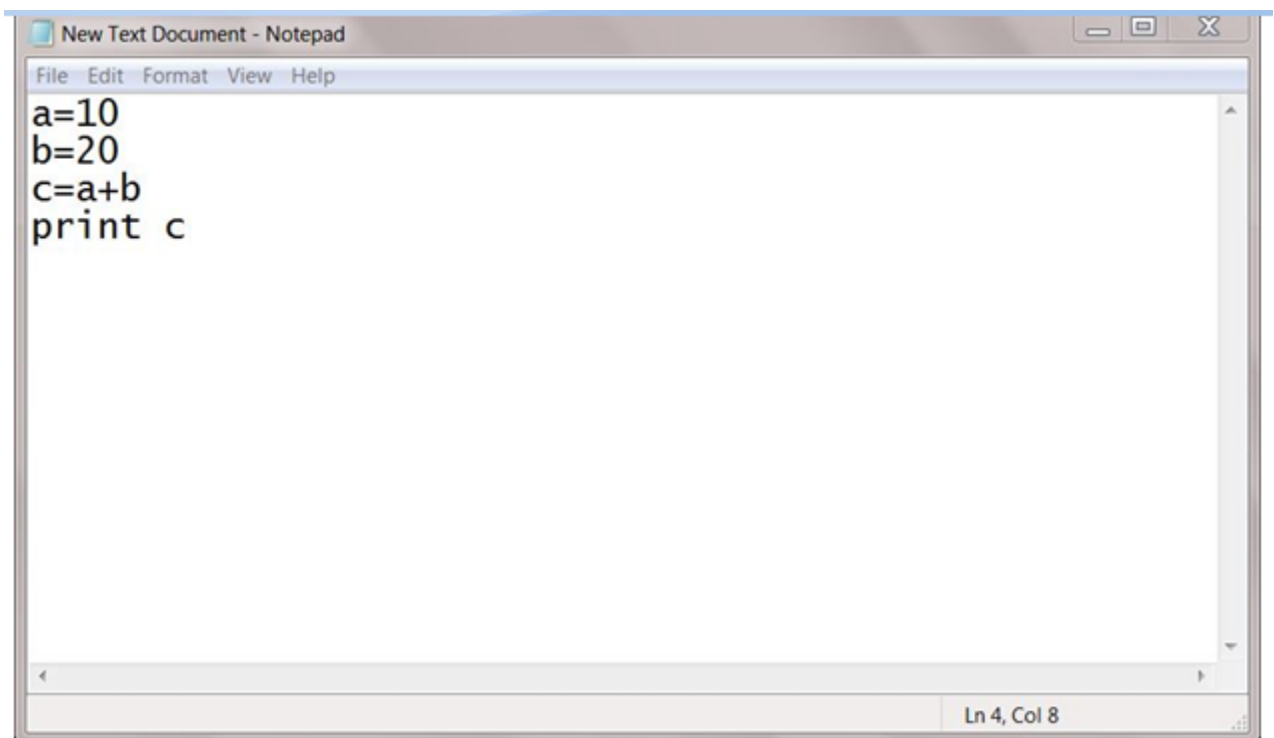
Eg:

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe - python". The window shows the output of running the Python interpreter. It displays the Microsoft Windows version (6.1.7600), copyright information (© 2009 Microsoft Corporation), and the Python version (2.7.4) along with the date and time (Apr 6 2013, 19:54:46). The prompt shows the user has entered 'python' at the C:\Users\OM> prompt. The Python shell then prompts for help information. The user enters several lines of code: 'a=10', 'b=20', 'c=a+b', and 'print c'. The shell outputs '30' after the 'print c' command. The prompt then returns to '>>>'.

iavatooint.com

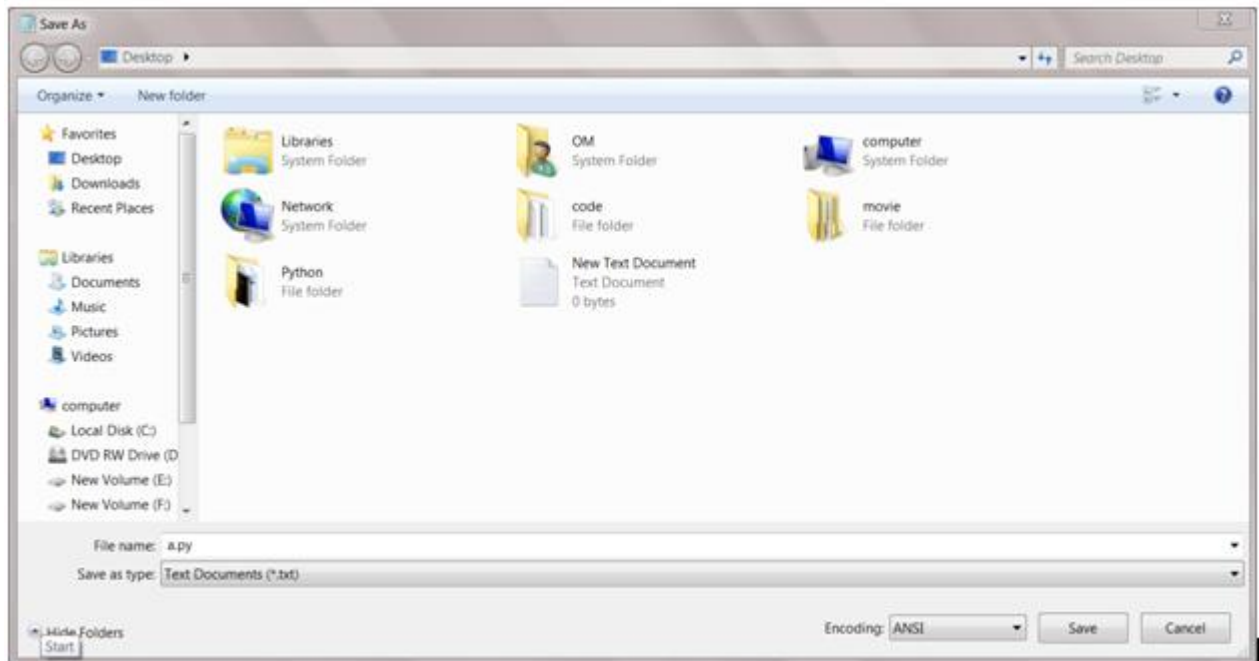
2) Script Mode

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

A screenshot of a Notepad window titled "New Text Document - Notepad". The window shows a menu bar with "File", "Edit", "Format", "View", and "Help". The text area contains the following Python code: 'a=10', 'b=20', 'c=a+b', and 'print c'. The status bar at the bottom right indicates "Ln 4, Col 8".

iavatooint.com

Save it by **.py** extension.



Now open Command prompt and execute it by :

```
C:\Windows\system32\cmd.exe
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
```

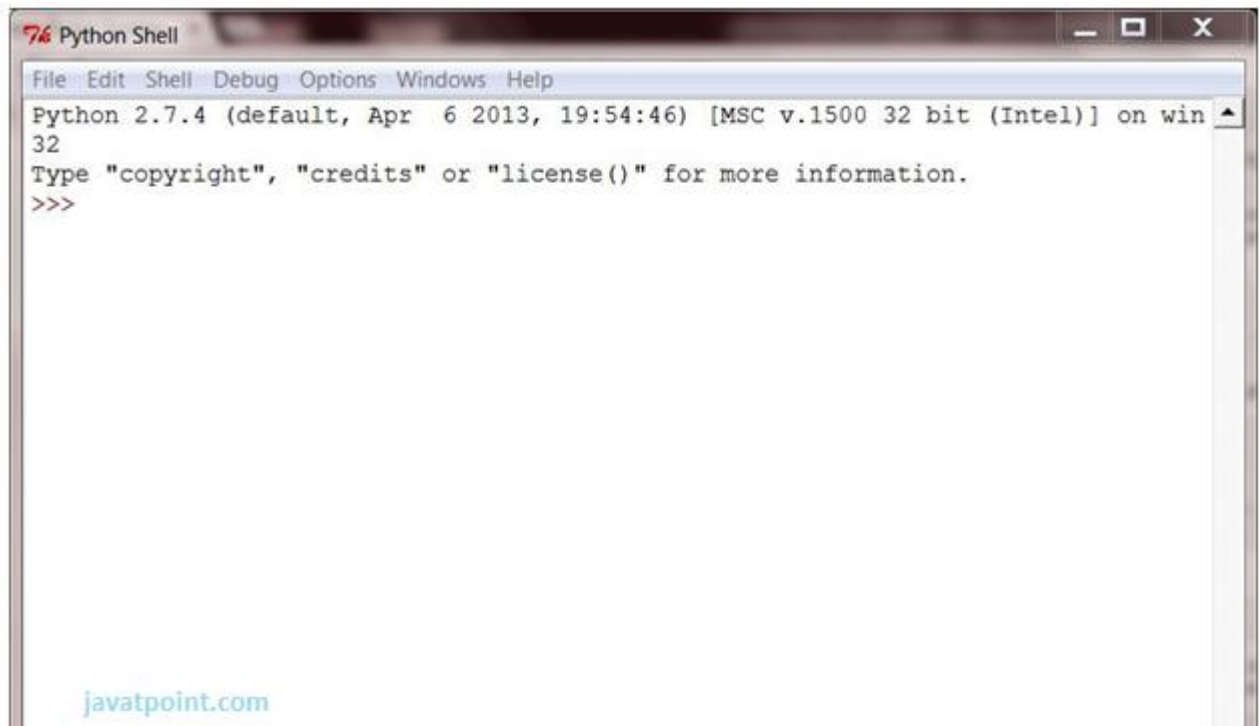
NOTE: Path in the command prompt should be location of saved file.where you have saved your file. In the above case file should be saved at desktop.

3) Using IDE (Integrated Development Environment)

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

All you need to do is:

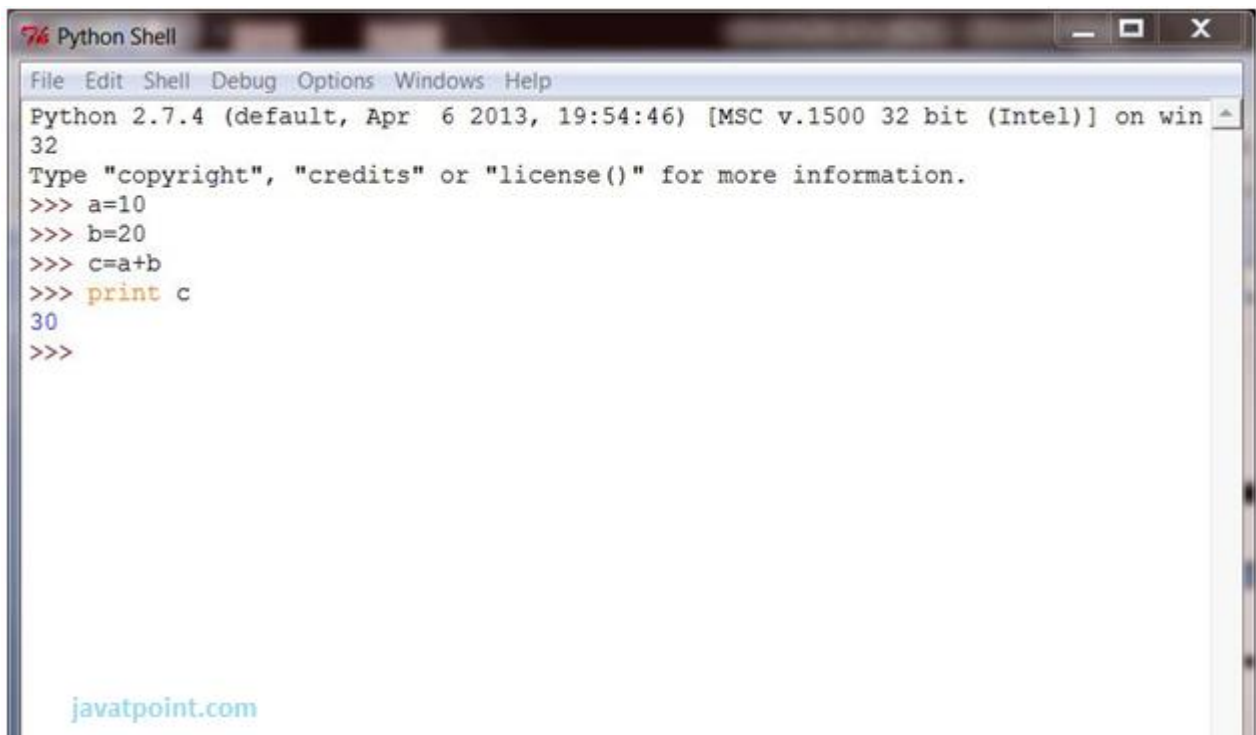
Click on Start button -> All Programs -> Python -> IDLE(Python GUI)



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.

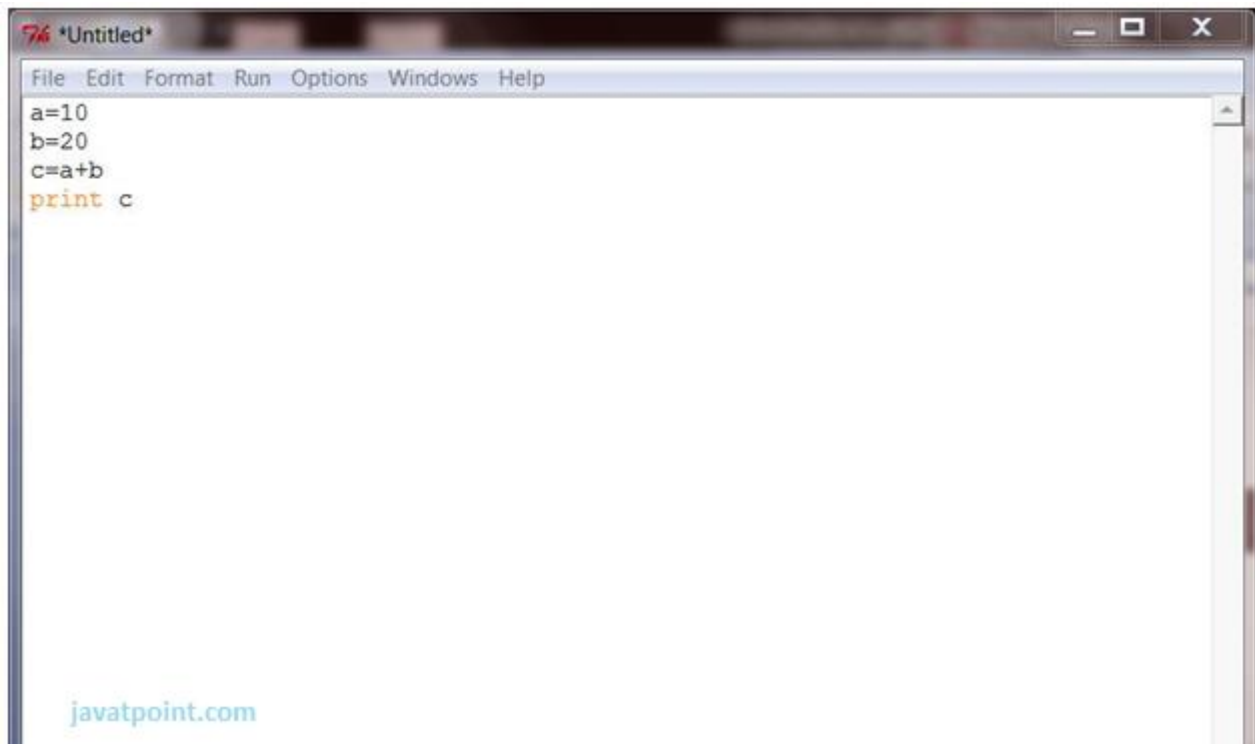


2) Using Script Mode:

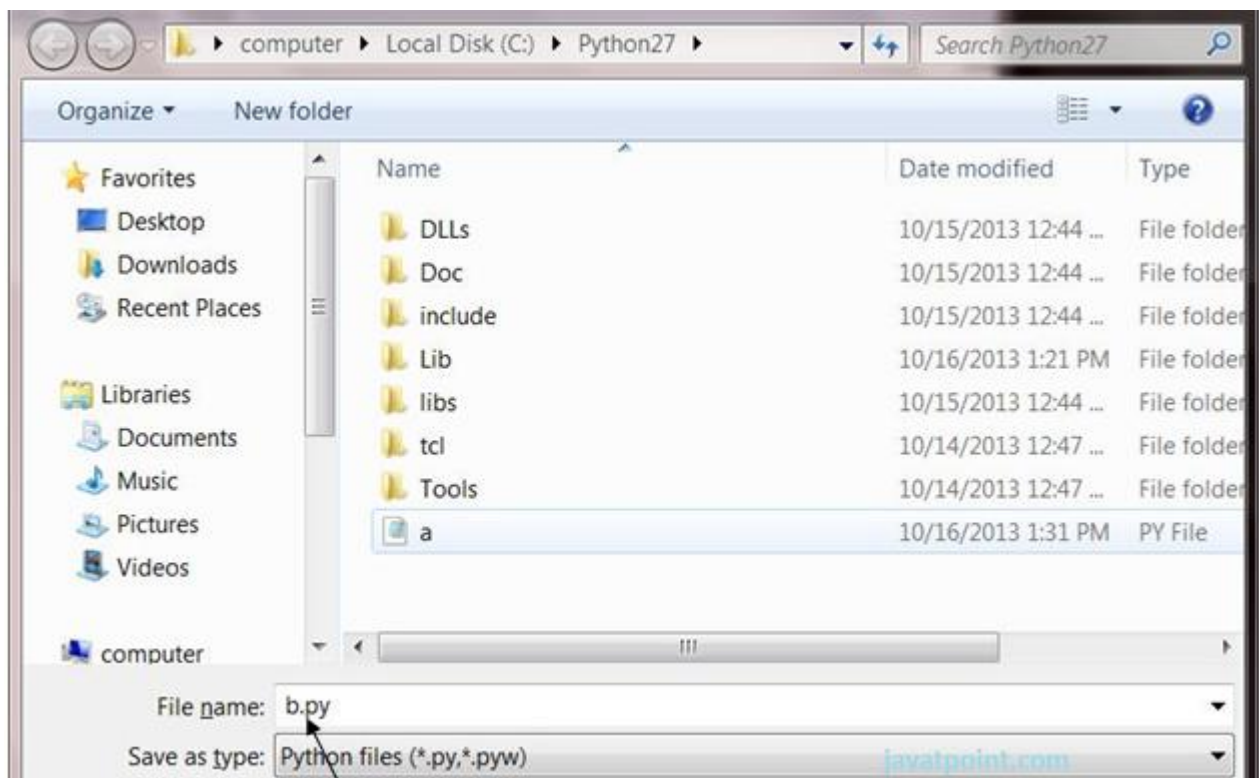
i) Click on Start button -> All Programs -> Python -> IDLE(Python GUI)

ii) Python Shell will be opened. Now click on File -> New Window.

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



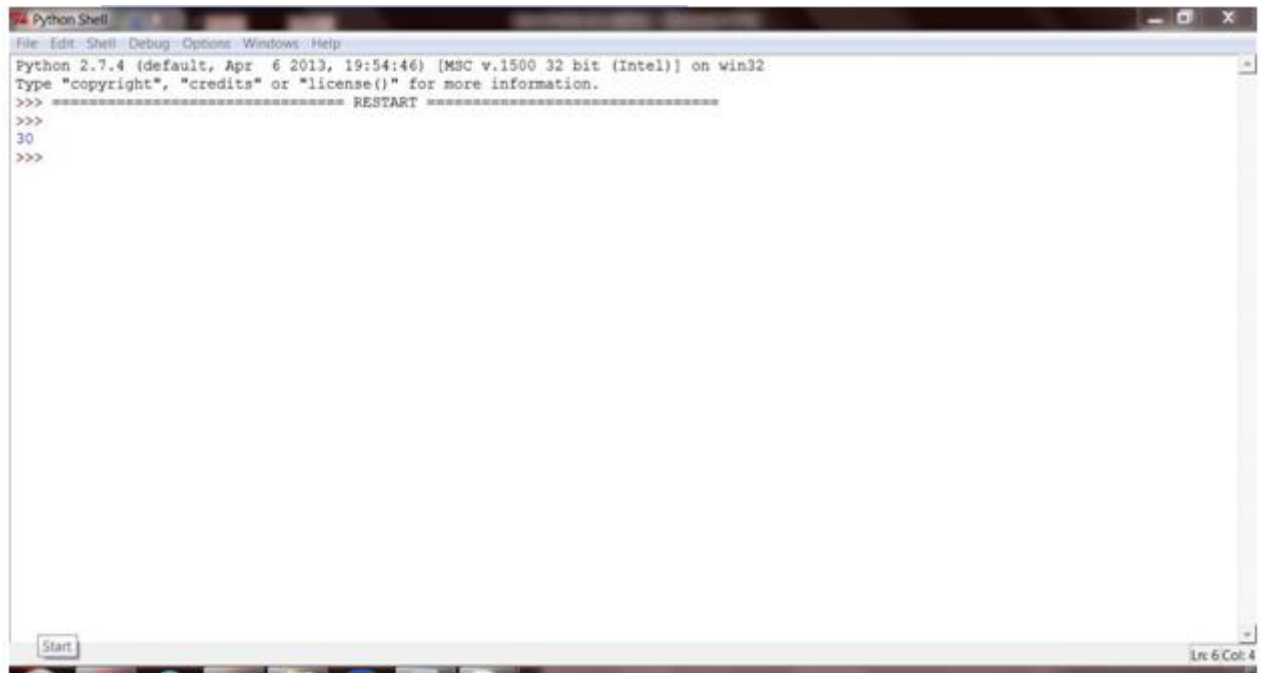
Click on file -> save as



Run code by clicking on Run in the Menu bar.

Run -> Run Module

Result will be displayed on a new Python shell as:

A screenshot of a Python Shell window. The title bar says 'Python Shell'. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area displays 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', followed by 'Type "copyright", "credits" or "license()" for more information.' Below this is a line of dashes and the word 'RESTART'. The prompt '>>>' is followed by the number '30'. At the bottom left is a 'Start' button, and at the bottom right is a status bar showing 'Line 6 Col: 4'.

javatpoint.com

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.

Note - Variable name should not be a keyword.

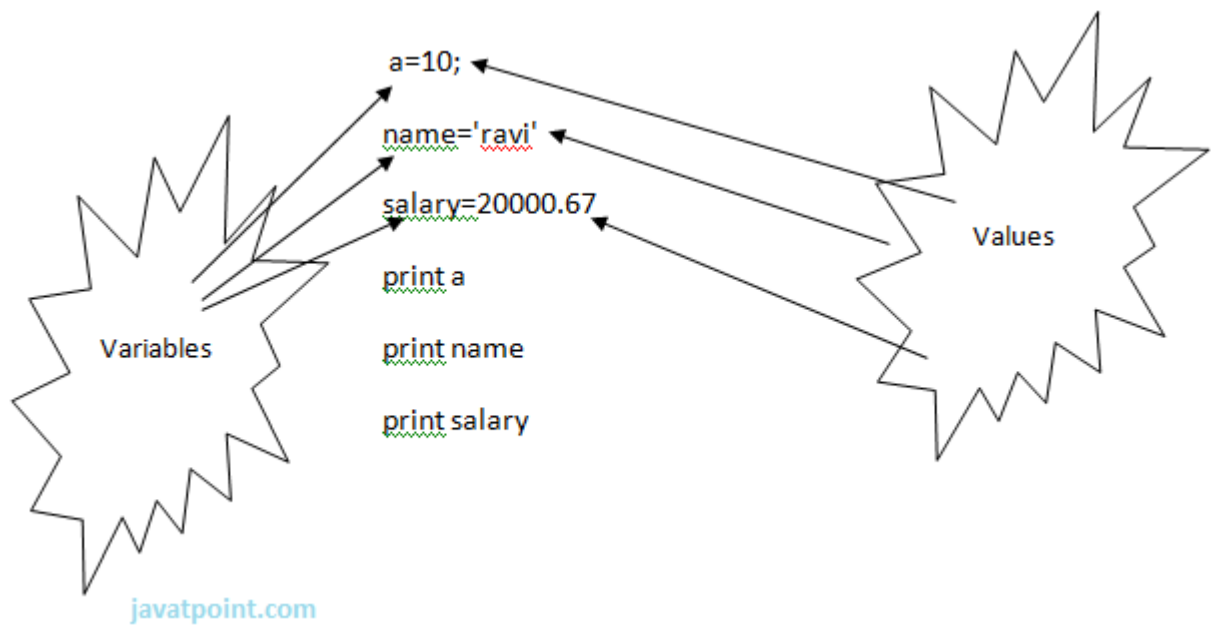
Declaring Variable and Assigning Values

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

Eg:

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:

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

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- 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 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

Identifiers

Identifiers are the names given to the fundamental building blocks in a program.

These can be variables ,class ,object ,functions , lists , dictionaries etc.

There are certain rules defined for naming i.e., Identifiers.

1. An identifier is a long sequence of characters and numbers.
2. No special character except underscore (_) can be used as an identifier.
3. Keyword should not be used as an identifier name.
4. Python is case sensitive. So using case is significant.
5. First character of an identifier can be character, underscore (_) but not digit.

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.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(compl
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of the real part and the imaginary part 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

Operators are particular symbols that are used to perform operations on operands. It returns result that can be used in application.

Example

1. $4 + 5 = 9$

Here 4 and 5 are Operands and (+) , (=) signs are the operators. This expression produces the output 9.

Types of Operators

Python supports the following operators

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.
7. Bitwise Operators.

Arithmetic Operators

The following table contains the arithmetic operators that are used to perform arithmetic operations.

Operators	Description
//	Perform Floor division(gives integer value after division)
+	To perform addition
-	To perform subtraction
*	To perform multiplication
/	To perform division
%	To return remainder after division(Modulus)
**	Perform exponent(raise to power)

Example

1. >>> 10+20
2. 30
3. >>> 20-10
4. 10
5. >>> 10*2
6. 20
7. >>> 10/2
8. 5
9. >>> 10%3
10. 1
11. >>> 2**3
12. 8
13. >>> 10//3
14. 3
15. >>>

Relational Operators

The following table contains the relational operators that are used to check relations.

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<>	Not equal to(similar to !=)

eg:

- >>> 10<20
- True
- >>> 10>20
- False
- >>> 10<=10
- True
- >>> 20>=15
- True
- >>> 5==6
- False
- >>> 5!=6
- True
- >>> 10<>2
- True
- >>>

Assignment Operators

The following table contains the assignment operators that are used to assign values to the variables.

Operators	Description
=	Assignment

<code>/=</code>	Divide and Assign
<code>+=</code>	Add and assign
<code>-=</code>	Subtract and Assign
<code>*=</code>	Multiply and assign
<code>%=</code>	Modulus and assign
<code>**=</code>	Exponent and assign
<code>//=</code>	Floor division and assign

Example

- `>>> c=10`
- `>>> c`
- `10`
- `>>> c+=5`
- `>>> c`
- `15`
- `>>> c-=5`
- `>>> c`
- `10`
- `>>> c*=2`
- `>>> c`
- `20`
- `>>> c/=2`
- `>>> c`
- `10`
- `>>> c%=3`
- `>>> c`
- `1`
- `>>> c=5`
- `>>> c**=2`
- `>>> c`
- `25`
- `>>> c//=2`
- `>>> c`
- `12`
- `>>>`

Logical Operators

The following table contains the arithmetic operators that are used to perform arithmetic operations.

Operators	Description
and	Logical AND(When both conditions are true output will be true)
or	Logical OR (If any one condition is true output will be true)
not	Logical NOT(Compliment the condition i.e., reverse)

Example

1. `a=5>4 and 3>2`
2. `print a`
3. `b=5>4 or 3<2`
4. `print b`
5. `c=not(5>4)`
6. `print c`

Output:

1. `>>>`
2. `True`
3. `True`
4. `False`
5. `>>>`

Membership Operators

The following table contains the membership operators.

Operators	Description
in	Returns true if a variable is in sequence of another variable, else false.
not in	Returns true if a variable is not in sequence of another variable, else false.

Example

1. `a=10`
2. `b=20`
3. `list=[10,20,30,40,50];`

```

4. if (a in list):
5.     print "a is in given list"
6. else:
7.     print "a is not in given list"
8. if(b not in list):
9.     print "b is not given in list"
10. else:
11.     print "b is given in list"

```

Output:

```

1. >>>
2. a is in given list
3. b is given in list
4. >>>

```

Identity Operators

The following table contains the identity operators.

Operators	Description
is	Returns true if identity of two operands are same, else false
is not	Returns true if identity of two operands are not same, else false.

Example

```

1. a=20
2. b=20
3. if( a is b):
4.     print a,b have same identity
5. else:
6.     print a, b are different
7. b=10
8. if( a is not b):
9.     print a,b have different identity
10. else:
11.     print a,b have same identity

```

Output

```

1. >>>
2. a,b have same identity
3. a,b have different identity
4. >>>

```


Python Comments

Python supports two types of comments:

1) Single lined comment:

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

Eg:

1. `# This is single line comment.`

2) Multi lined 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"""`

Python If Statements

The Python if statement is a statement which is used to test specified condition. We can use if statement to perform conditional operations in our Python application.

The if statement executes only when specified condition is **true**. We can pass any valid expression into the if parentheses.

There are various types of if statements in Python.

- if statement
- if-else statement
- nested if statement

Python If Statement Syntax

1. **if**(condition):
2. statements

Python If statement flow chart

Python If Statement Example

1. a=10
2. **if** a==10:
3. **print** "Welcome to javatpoint"

Output:

```
Hello User
```

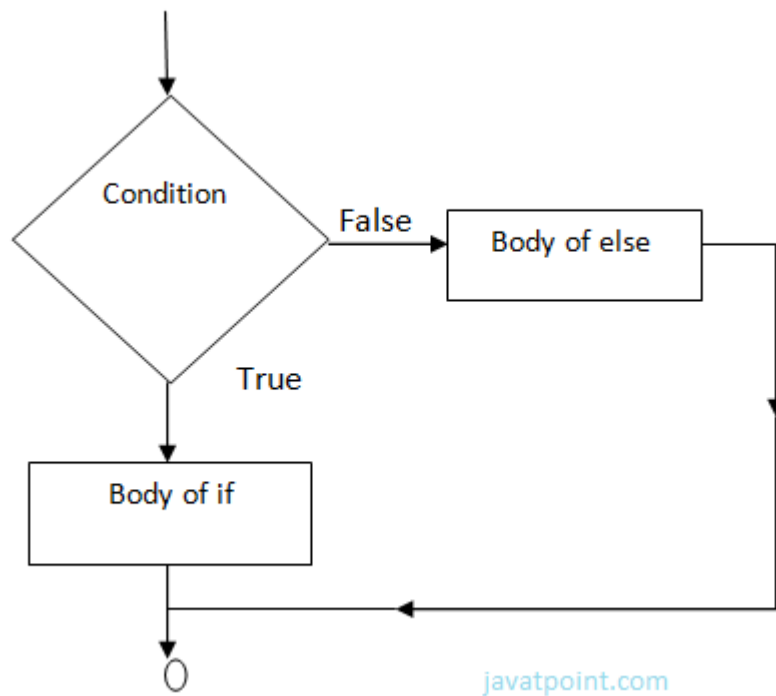
Python If Else Statements

The If statement is used to test specified condition and if the condition is true, if block executes, otherwise else block executes.

The else statement executes when the if statement is false.

Python If Else Syntax

1. **if**(condition): False
2. statements
3. **else**: True
4. statements



Example-

1. year=2000
2. **if** year%4==0:
3. **print** "Year is Leap"
4. **else**:
5. **print** "Year is not Leap"

Output:

```
Year is Leap
```

Python Nested If Else Statement

In python, we can use nested If Else to check multiple conditions. Python provides **elif** keyword to make nested If statement.

This statement is like executing a if statement inside a else statement.

Python Nested If Else Syntax

1. If statement:
2. Body
3. **elif** statement:
4. Body
5. **else**:
6. Body

Python Nested If Else Example

1. a=10
2. **if** a>=20:
3. **print** "Condition is True"
4. **else**:
5. **if** a>=15:
6. **print** "Checking second value"
7. **else**:
8. **print** "All Conditions are false"

Output:

```
All Conditions are false.
```

For Loop

Python **for loop** is used to iterate the elements of a collection in the order that they appear. This collection can be a sequence(list or string).

Python For Loop Syntax

1. **for** <variable> **in** <sequence>:

Output:

1. 1
2. 7
3. 9

Explanation:

- Firstly, the first value will be assigned in the variable.
- Secondly all the statements in the body of the loop are executed with the same value.
- Thirdly, once step second is completed then variable is assigned the next value in the sequence and step second is repeated.
- Finally, it continues till all the values in the sequence are assigned in the variable and processed.

Python For Loop Simple Example

1. num=2
2. **for** a **in** range (1,6):
3. **print** num * a

Output:

1. 2
- 2.
3. 4
- 4.
5. 6
- 6.
7. 8
- 8.
9. 10

Python Example to Find Sum of 10 Numbers

1. sum=0
2. **for** n **in** range(1,11):
3. sum+=n
4. **print** sum

Output:

1. 55

Python Nested For Loops

Loops defined within another Loop are called Nested Loops. Nested loops are used to iterate matrix elements or to perform complex computation.

When an outer loop contains an inner loop in its body it is called Nested Looping.

Python Nested For Loop Syntax

1. **for** <expression>:
2. **for** <expression>:
3. Body

Python Nested For Loop Example

1. **for** i **in** range(1,6):
2. **for** j **in** range (1,i+1):
3. **print** i,
4. **print**

Output:

1. >>>
2. 1
3. 2 2
4. 3 3 3
5. 4 4 4 4
6. 5 5 5 5 5
7. >>>

Explanation:

For each value of Outer loop the whole inner loop is executed.

For each value of inner loop the Body is executed each time.

Python Nested Loop Example 2

1. **for** i **in** range (1,6):
2. **for** j **in** range (5,i-1,-1):
3. **print** "*",
4. **print**

Output:

1. >>>
2. * * * * *
3. * * * *

4. * * *
5. * *
6. *

Python While Loop

In Python, while loop is used to execute number of statements or body till the specified condition is true. Once the condition is false, the control will come out of the loop.

Python While Loop Syntax

1. **while** <expression>:
2. Body

Here, loop Body will execute till the expression passed is true. The Body may be a single statement or multiple statement.

Python While Loop Example 1

1. a=10
2. **while** a>0:
3. **print** "Value of a is",a
4. a=a-2

print "Loop is Completed"

Output:

1. >>>
2. Value of a **is** 10
3. Value of a **is** 8
4. Value of a **is** 6
5. Value of a **is** 4
6. Value of a **is** 2
7. Loop **is** Completed
8. >>>

Explanation:

- Firstly, the value in the variable is initialized.
- Secondly, the condition/expression in the while is evaluated. Consequently if condition is true, the control enters in the body and executes all the statements . If the condition/expression passed results in false then the control exists the body and straight away control goes to next instruction after body of while.
- Thirdly, in case condition was true having completed all the statements, the variable is incremented or decremented. Having changed the value of variable step second is followed. This process continues till the expression/condition becomes false.
- Finally Rest of code after body is executed.

Python While Loop Example 2

```
1. n=153
2. sum=0
3. while n>0:
4.     r=n%10
5.     sum+=r
6.     n=n/10
7. print sum
```

Output:

```
1. >>>
2. 9
3. >>>
```

Python Break

Break statement is a jump statement which is used to transfer execution control. It breaks the current execution and in case of inner loop, inner loop terminates immediately.

When break statement is applied the control points to the line following the body of the loop, hence applying break statement makes the loop to terminate and controls goes to next line pointing after loop body.

Python Break Example 1

```
1. for i in [1,2,3,4,5]:
2.     if i==4:
3.         print "Element found"
4.         break
5.     print i,
```

Output:

```
1. >>>
2. 1 2 3 Element found
3. >>>
```

Python Break Example 2

```
1. for letter in 'Python3':
2.     if letter == 'o':
3.         break
4.     print (letter)
```

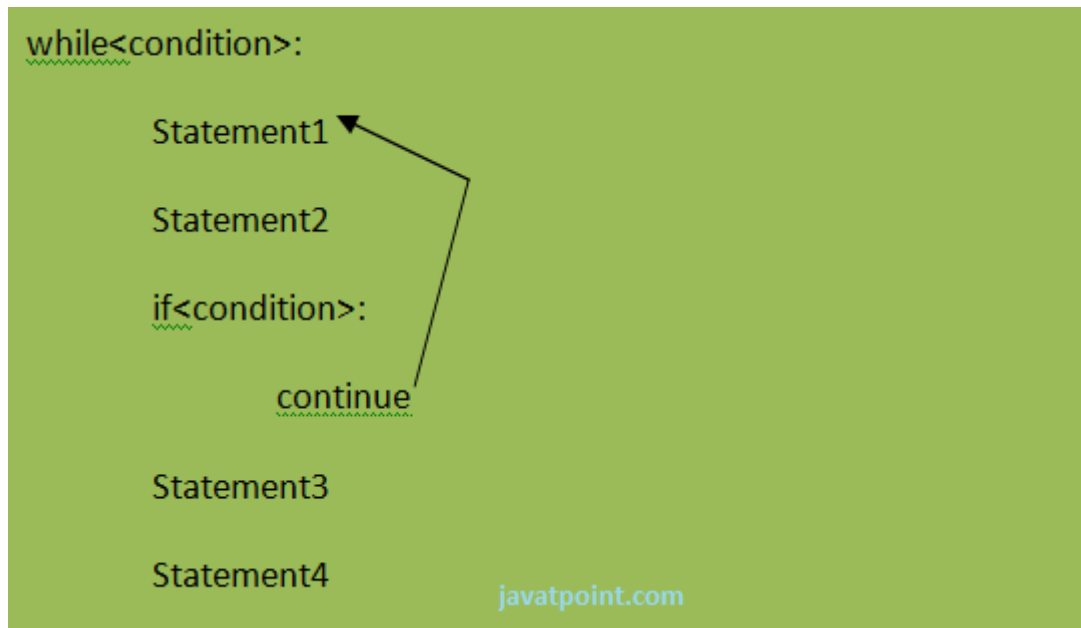
Output:

```
1. P
2. y
3. t
4. h
```


Python Continue Statement

Python Continue Statement is a jump statement which is used to skip execution of current iteration. After skipping, loop continue with next iteration.

We can use continue statement with for as well as while loop in Python.



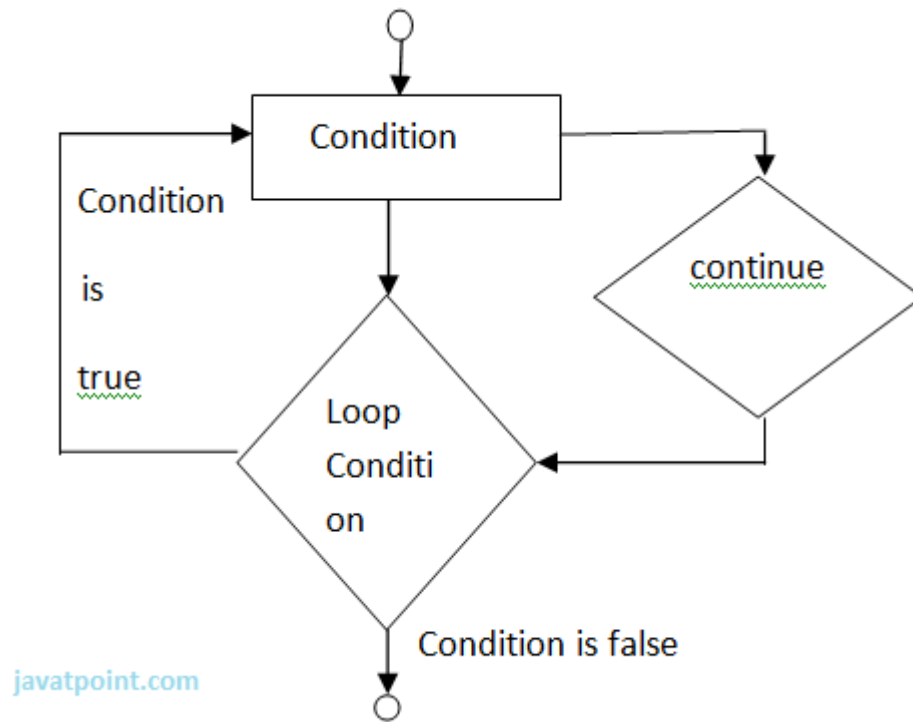
Python Continue Statement Example

1. `a=0`
2. `while a<=5:`
3. `a=a+1`
4. `if a%2==0:`
5. `continue`
6. `print a`
7. `print "End of Loop"`

Output:

1. `>>>`
2. `1`
3. `3`
4. `5`
5. `End of Loop`
6. `>>>`

Python Continue Statement Flow chart



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. >>>

PYTHON STRINGS

Strings are the simplest and easy to use in Python.

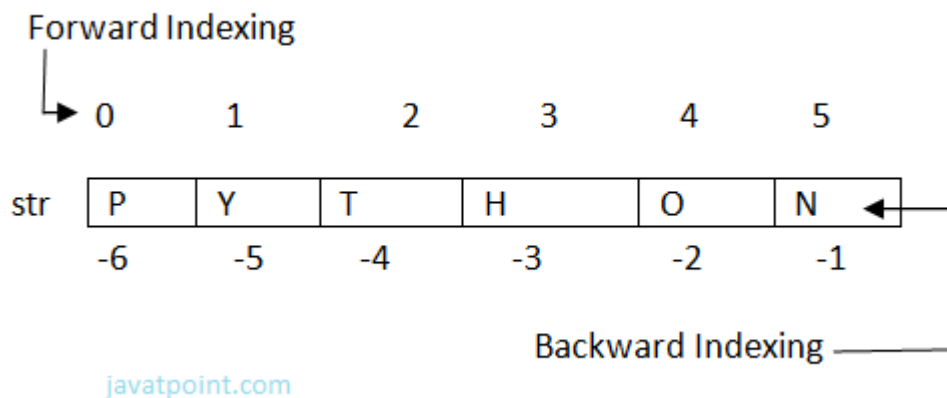
String python are immutable.

We can simply create Python String by enclosing a text in single as well as double quotes. Python treat both single and double quotes statements same.

Accessing Strings:

- In Python, Strings are stored as individual characters in a contiguous memory location.
- The benefit of using String is that it can be accessed from both the directions in forward and backward.
- Both forward as well as backward indexing are provided using Strings in Python.
 - Forward indexing starts with 0,1,2,3,....
 - Backward indexing starts with -1,-2,-3,-4,....

eg:



1. `str[0]='P'=str[-6]` , `str[1]='Y' = str[-5]` , `str[2] = 'T' = str[-4]` , `str[3] = 'H' = str[-3]`
2. `str[4] = 'O' = str[-2]` , `str[5] = 'N' = str[-1]`.

Simple program to retrieve String in reverse as well as normal form.

1. `name="Rajat"`
2. `length=len(name)`
3. `i=0`
4. `for n in range(-1,(-length-1),-1):`
5. `print name[i],"\t",name[n]`
6. `i+=1`

Output:

```
>>>
R      t
a      a
j      j
a      a
```

```
t      R
>>>
```

Strings Operators

There are basically 3 types of Operators supported by String:

1. Basic Operators.
2. Membership Operators.
3. Relational Operators.

Basic Operators:

There are two types of basic operators in String. They are "+" and "*".

String Concatenation Operator :(+)

The concatenation operator (+) concatenate two Strings and forms a new String.

eg:

```
>>> "ratan" + "jaiswal"
```

Output:

```
'ratanjaiswal'
>>>
```

Expression	Output
'10' + '20'	'1020'
"s" + "007"	's007'
'abcd123' + 'xyz4'	'abcd123xyz4'

NOTE: Both the operands passed for concatenation must be of same type, else it will show an error.

Eg:

```
'abc' + 3
>>>
```

output:

```
Traceback (most recent call last):
  File "", line 1, in
    'abc' + 3
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Replication Operator: (*)

Replication operator uses two parameter for operation. One is the integer value and the other one is the String.

The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

Eg:

1. `>>> 5*"Vimal"`

Output:

```
'VimalVimalVimalVimalVimal'
```

Expression	Output
<code>"soono"*2</code>	<code>'soonosoono'</code>
<code>3*'1'</code>	<code>'111'</code>
<code>'\$'*5</code>	<code>'\$\$\$\$\$'</code>

NOTE: We can use Replication operator in any way i.e., `int * string` or `string * int`. Both the parameters passed cannot be of same type.

Membership Operators

Membership Operators are already discussed in the Operators section. Let see with context of String.

There are two types of Membership operators:

1) in: "in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

2) not in: "not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

Eg:

- `>>> str1="javatpoint"`
- `>>> str2='ssit'`
- `>>> str3="seomount"`
- `>>> str4='java'`
- `>>> st5="it"`
- `>>> str6="seo"`

```
7. >>> str4 in str1
8. True
9. >>> str5 in str2
10. >>> str5 in str2
11. True
12. >>> str6 in str3
13. True
14. >>> str4 not in str1
15. False
16. >>> str1 not in str4
17. True
```

Relational Operators:

All the comparison operators i.e., (<,>,<=,>=,==,!=,<>) are also applicable to strings. The Strings are compared based on the ASCII value or Unicode(i.e., dictionary Order).

Eg:

```
1. >>> "RAJAT"=="RAJAT"
2. True
3. >>> "afsha">='Afsha'
4. True
5. >>> "Z"<>"z"
6. True
```

Explanation:

The ASCII value of a is 97, b is 98, c is 99 and so on. The ASCII value of A is 65,B is 66,C is 67 and so on. The comparison between strings are done on the basis on ASCII value.

Slice Notation:

String slice can be defined as substring which is the part of string. Therefore further substring can be obtained from a string.

There can be many forms to slice a string. As string can be accessed or indexed from both the direction and hence string can also be sliced from both the direction that is left and right.

Syntax:

```
1. <string_name>[startIndex:endIndex],
2. <string_name>[:endIndex],
3. <string_name>[startIndex:]
```

Example:

```
1. >>> str="Nikhil"
2. >>> str[0:6]
3. 'Nikhil'
4. >>> str[0:3]
```

5. 'Nik'
6. >>> str[2:5]
7. 'khi'
8. >>> str[:6]
9. 'Nikhil'
10. >>> str[3:]
11. 'hil'

Note: startIndex in String slice is inclusive whereas endIndex is exclusive.

String slice can also be used with Concatenation operator to get whole string.

Eg:

1. >>> str="Mahesh"
2. >>> str[:6]+str[6:]
3. 'Mahesh'

//here 6 is the length of the string.

String Functions and Methods:

There are many predefined or built in functions in String. They are as follows:

capitalize()	It capitalizes the first character of the String.
count(string,begin,end)	Counts number of times substring occurs in a String between b
endswith(suffix ,begin=0,end=n)	Returns a Boolean value if the string terminates with given su end.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is four and end index.
index(subsring, beginIndex, endIndex)	Same as find() except it raises an exception if string is not four
isalnum()	It returns True if characters in the string are alphanumeric i.e. and there is at least 1 character. Otherwise it returns False.
isalpha()	It returns True when all the characters are alphabets and character, otherwise False.

isdigit()	It returns True if all the characters are digit and there is a digit, otherwise False.
islower()	It returns True if the characters of a string are in lower case, otherwise False.
isupper()	It returns True if characters of a string are in Upper case, otherwise False.
isspace()	It returns True if the characters of a string are whitespace, otherwise False.
len(string)	len() returns the length of a string.
lower()	Converts all the characters of a string to Lower case.
upper()	Converts all the characters of a string to Upper Case.
startswith(str ,begin=0,end=n)	Returns a Boolean value if the string starts with given str between begin and end.
swapcase()	Inverts case of all characters in a string.
lstrip()	Remove all leading whitespace of a string. It can also be used to remove a character from leading.
rstrip()	Remove all trailing whitespace of a string. It can also be used to remove a character from trailing.

Examples:

1) capitalize()

```
1. >>> 'abc'.capitalize()
```

Output:

```
'Abc'
```

2) count(string)

```
1. msg = "welcome to sssit";
2. substr1 = "o";
3. print msg.count(substr1, 4, 16)
4. substr2 = "t";
5. print msg.count(substr2)
```

Output:


```
>>>
2
2
>>>
```

3) endswith(string)

1. string1="Welcome to SSSIT";
2. substring1="SSSIT";
3. substring2="to";
4. substring3="of";
5. **print** string1.endswith(substring1);
6. **print** string1.endswith(substring2,2,16);
7. **print** string1.endswith(substring3,2,19);
8. **print** string1.endswith(substring3);

Output:

```
>>>
True
False
False
False
>>>
```

4) find(string)

1. str="Welcome to SSSIT";
2. substr1="come";
3. substr2="to";
4. **print** str.find(substr1);
5. **print** str.find(substr2);
6. **print** str.find(substr1,3,10);
7. **print** str.find(substr2,19);

Output:

```
>>>
3
8
3
-1
>>>
```

5) index(string)

1. str="Welcome to world of SSSIT";
2. substr1="come";
3. substr2="of";
4. **print** str.index(substr1);
5. **print** str.index(substr2);
6. **print** str.index(substr1,3,10);
7. **print** str.index(substr2,19);

Output:

```
>>>
3
17
3
Traceback (most recent call last):
  File "C:/Python27/fin.py", line 7, in
    print str.index(substr2,19);
ValueError: substring not found
>>>
```

6) isalnum()

1. str="Welcome to sssit";
2. **print** str.isalnum();
3. str1="Python47";
4. **print** str1.isalnum();

Output:

```
>>>
False
True
>>>
```

7) isalpha()

1. string1="HelloPython"; # Even space is not allowed
2. **print** string1.isalpha();
3. string2="This is Python2.7.4"
4. **print** string2.isalpha();

Output:

```
>>>
True
False
>>>
```

8) isdigit()

1. string1="HelloPython";
2. **print** string1.isdigit();
3. string2="98564738"
4. **print** string2.isdigit();

Output:

```
>>>
False
True
>>>
```

9) islower()

1. string1="Hello Python";

2. **print** string1.islower();
3. string2="welcome to "
4. **print** string2.islower();

Output:

```
>>>
False
True
>>>
```

10) isupper()

1. string1="Hello Python";
2. **print** string1.isupper();
3. string2="WELCOME TO"
4. **print** string2.isupper();

Output:

```
>>>
False
True
>>>
```

11) isspace()

1. string1=" ";
2. **print** string1.isspace();
3. string2="WELCOME TO WORLD OF PYT"
4. **print** string2.isspace();

Output:

```
>>>
True
False
>>>
```

12) len(string)

1. string1=" ";
2. **print** len(string1);
3. string2="WELCOME TO SSSIT"
4. **print** len(string2);

Output:

```
>>>
4
16
>>>
```

13) lower()

1. string1="Hello Python";
2. **print** string1.lower();
3. string2="WELCOME TO SSSIT"
4. **print** string2.lower();

Output:

```
>>>
hello python
welcome to sssit
>>>
```

14) upper()

1. string1="Hello Python";
2. **print** string1.upper();
3. string2="welcome to SSSIT"
4. **print** string2.upper();

Output:

```
>>>
HELLO PYTHON
WELCOME TO SSSIT
>>>
```

15) startswith(string)

1. string1="Hello Python";
2. **print** string1.startswith('Hello');
3. string2="welcome to SSSIT"
4. **print** string2.startswith('come',3,7);

Output:

```
>>>
True
True
>>>
```

16) swapcase()

1. string1="Hello Python";
2. **print** string1.swapcase();
3. string2="welcome to SSSIT"
4. **print** string2.swapcase();

Output:

```
>>>
hELLO pYTHON
WELCOME TO sssit
>>>
```

17) lstrip()

1. `string1=" Hello Python";`
2. `print string1.lstrip();`
3. `string2="@@@@@@@welcome to SSSIT"`
4. `print string2.lstrip('@');`

Output:

```
>>>
Hello Python
welcome to world to SSSIT
>>>
```

18) rstrip()

1. `string1=" Hello Python ";`
2. `print string1.rstrip();`
3. `string2="@welcome to SSSIT!!!"`
4. `print string2.rstrip('!');`

Output:

```
>>>
        Hello Python
@welcome to SSSIT
>>>
```

Python List

- 1).Python lists are the data structure that is capable of holding different type of data.
- 2).Python lists are mutable i.e., Python will not create a new list if we modify an element in the list.
- 3).It is a container that holds other objects in a given order. Different operation like insertion and deletion can be performed on lists.
- 4).A list can be composed by storing a sequence of different type of values separated by commas.
- 5).A python list is enclosed between square([]) brackets.
- 6).The elements are stored in the index basis with starting index as 0.

eg:

1. data1=[1,2,3,4];
2. data2=['x','y','z'];
3. data3=[12.5,11.6];
4. data4=['raman','rahul'];
5. data5=[];
6. data6=['abhinav',10,56.4,'a'];

Accessing Lists

A list can be created by putting the value inside the square bracket and separated by comma.

Syntax:

1. <list_name>=[value1,value2,value3,...,valuen];

For accessing list :

1. <list_name>[index]

Different ways to access list:

Eg:

1. data1=[1,2,3,4];
2. data2=['x','y','z'];
3. **print** data1[0]
4. **print** data1[0:2]
5. **print** data2[-3:-1]
6. **print** data1[0:]
7. **print** data2[:2]

Output:

```
>>>
>>>
```

```
1
[1, 2]
['x', 'y']
[1, 2, 3, 4]
['x', 'y']
>>>
```

Elements in a Lists:

1. Data=[1,2,3,4,5];

Forward indexing

0 1 2 3 4

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

-5 -4 -3 -2 -1

javatpoint.com

Backward indexing

1. Data[0]=1=Data[-5] , Data[1]=2=Data[-4] , Data[2]=3=Data[-3] ,
2. =4=Data[-2] , Data[4]=5=Data[-1].

Note: Internal Memory Organization:
List do not store the elements directly at the index. In fact a reference is stored at each index which subsequently refers to the object stored somewhere in the memory. This is due to the fact that some objects may be large enough than other objects and hence they are stored at some other memory location.

List Operations:

Various Operations can be performed on List. Operations performed on List are given as:

a) Adding Lists:

Lists can be added by using the concatenation operator(+) to join two lists.

Eg:

1. list1=[10,20]
2. list2=[30,40]
3. list3=list1+list2
4. **print** list3

Output:

1. >>>
2. [10, 20, 30, 40]
3. >>>

Note: '+' operator implies that both the operands passed must be list else error will be shown.

Eg:

1. list1=[10,20]
2. list1+30
3. **print** list1

Output:

1. Traceback (most recent call last):
2. File "C:/Python27/lis.py", line 2, in <module>
3. list1+30

b) Replicating lists:

Replicating means repeating . It can be performed by using '*' operator by a specific number of time.

Eg:

1. list1=[10,20]
2. **print** list1*1

Output:

1. >>>
2. [10, 20]
3. >>>

c) List slicing:

A subpart of a list can be retrieved on the basis of index. This subpart is known as list slice.

Eg:

1. list1=[1,2,4,5,7]
2. **print** list1[0:2]
3. **print** list1[4]
4. list1[1]=9
5. **print** list1

Output:

1. >>>
2. [1, 2]
3. 7
4. [1, 9, 4, 5, 7]
5. >>>

Note: If the index provided in the list slice is outside the list, then it raises an IndexError exception.

Other Operations:

Apart from above operations various other functions can also be performed on List such as Updating, Appending and Deleting elements from a List:

a) Updating elements in a List:

To update or change the value of particular index of a list, assign the value to that particular index of the List.

Syntax:

1. <list_name>[index]=<value>

Eg:

1. data1=[5,10,15,20,25]
2. **print** "Values of list are: "
3. **print** data1
4. data1[2]="Multiple of 5"
5. **print** "Values of list are: "
6. **print** data1

Output:

1. >>>
2. Values of list are:
3. [5, 10, 15, 20, 25]
4. Values of list are:
5. [5, 10, 'Multiple of 5', 20, 25]
6. >>>

b) Appending elements to a List:

append() method is used to append i.e., add an element at the end of the existing elements.

Syntax:

1. <list_name>.append(item)

Eg:

1. list1=[10,"rahul",'z']
2. **print** "Elements of List are: "
3. **print** list1
4. list1.append(10.45)
5. **print** "List after appending: "
6. **print** list1

Output:

1. >>>
2. Elements of List are:
3. [10, 'rahul', 'z']
4. List after appending:
5. [10, 'rahul', 'z', 10.45]
6. >>>

c) Deleting Elements from a List:

del statement can be used to delete an element from the list. It can also be used to delete all items from startIndex to endIndex.

Eg:

1. list1=[10,'rahul',50.8,'a',20,30]
2. **print** list1
3. **del** list1[0]
4. **print** list1
5. **del** list1[0:3]
6. **print** list1

Output:

1. >>>
2. [10, 'rahul', 50.8, 'a', 20, 30]
3. ['rahul', 50.8, 'a', 20, 30]
4. [20, 30]
5. >>>

Functions and Methods of Lists:

There are many Built-in functions and methods for Lists. They are as follows:

There are following List functions:

Function	Description
min(list)	Returns the minimum value from the list given.
max(list)	Returns the largest value from the given list.
len(list)	Returns number of elements in a list.
cmp(list1,list2)	Compares the two list.

list(sequence)	Takes sequence types and converts them to lists.
----------------	--

1) min(list):

Eg:

1. list1=[101,981,'abcd','xyz','m']
2. list2=['aman','shekhar',100.45,98.2]
3. **print** "Minimum value in List1: ",min(list1)
4. **print** "Minimum value in List2: ",min(list2)

Output:

1. >>>
2. Minimum value **in** List1: 101
3. Minimum value **in** List2: 98.2
4. >>>

2) max(list):

Eg:

1. list1=[101,981,'abcd','xyz','m']
2. list2=['aman','shekhar',100.45,98.2]
3. **print** "Maximum value in List : ",max(list1)
4. **print** "Maximum value in List : ",max(list2)

Output:

1. >>>
2. Maximum value **in** List : xyz
3. Maximum value **in** List : shekhar
4. >>>

3) len(list):

Eg:

1. list1=[101,981,'abcd','xyz','m']
2. list2=['aman','shekhar',100.45,98.2]
3. **print** "No. of elements in List1: ",len(list1)
4. **print** "No. of elements in List2: ",len(list2)

Output:

1. >>>
2. No. of elements **in** List1 : 5
3. No. of elements **in** List2 : 4
4. >>>

4) cmp(list1,list2):

Explanation: If elements are of the same type, perform the comparison and return the result. If elements are different types, check whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

Eg:

1. list1=[101,981,'abcd','xyz','m']
2. list2=['aman','shekhar',100.45,98.2]
3. list3=[101,981,'abcd','xyz','m']
4. **print** cmp(list1,list2)
5. **print** cmp(list2,list1)
6. **print** cmp(list3,list1)

Output:

1. >>>
2. -1
3. 1
4. 0
5. >>>

5) list(sequence):

Eg:

1. seq=(145,"abcd",'a')
2. data=list(seq)
3. **print** "List formed is : ",data

Output:

1. >>>
2. List formed **is** : [145, 'abcd', 'a']
3. >>>

There are following built-in methods of List:

Methods	Description
index(object)	Returns the index value of the object.

count(object)	It returns the number of times an object is repeated in list.
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the po
insert(index,object)	Insert an object at the given index.
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.
sort()	It is used to sort the elements of the List.

1) index(object):

Eg:

1. data = [786,'abc','a',123.5]
2. **print** "Index of 123.5:", data.index(123.5)
3. **print** "Index of a is", data.index('a')

Output:

1. >>>
2. Index of 123.5 : 3
3. Index of a **is** 2
4. >>>

2) count(object):

Eg:

1. data = [786,'abc','a',123.5,786,'rahul','b',786]
2. **print** "Number of times 123.5 occurred is", data.count(123.5)
3. **print** "Number of times 786 occurred is", data.count(786)

Output:

1. >>>
2. Number of times 123.5 occurred **is** 1
3. Number of times 786 occurred **is** 3
4. >>>

3) pop()/pop(int):

Eg:

1. data = [786,'abc','a',123.5,786]
2. **print** "Last element is", data.pop()
3. **print** "2nd position element:", data.pop(1)
4. **print** data

Output:

1. >>>
2. Last element **is** 786
3. 2nd position element:abc
4. [786, 'a', 123.5]
5. >>>

4) insert(index,object):

Eg:

1. data=['abc',123,10.5,'a']
2. data.insert(2,'hello')
3. **print** data

Output:

1. >>>
2. ['abc', 123, 'hello', 10.5, 'a']
3. >>>

5) extend(sequence):

Eg:

1. data1=['abc',123,10.5,'a']
2. data2=['ram',541]
3. data1.extend(data2)
4. **print** data1
5. **print** data2

Output:

1. >>>
2. ['abc', 123, 10.5, 'a', 'ram', 541]
3. ['ram', 541]
4. >>>

6) remove(object):

Eg:

1. data1=['abc',123,10.5,'a','xyz']
2. data2=['ram',541]

3. **print** data1
4. data1.remove('xyz')
5. **print** data1
6. **print** data2
7. data2.remove('ram')
8. **print** data2

Output:

1. >>>
2. ['abc', 123, 10.5, 'a', 'xyz']
3. ['abc', 123, 10.5, 'a']
4. ['ram', 541]
5. [541]
6. >>>

7) reverse():

Eg:

1. list1=[10,20,30,40,50]
2. list1.reverse()
3. **print** list1

Output:

1. >>>
2. [50, 40, 30, 20, 10]
3. >>>

8) sort():

Eg:

1. list1=[10,50,13,'rahul','aakash']
2. list1.sort()
3. **print** list1

Output:

1. >>>
2. [10, 13, 50, 'aakash', 'rahul']
3. >>>

Python Tuple

A tuple is a sequence of immutable objects, therefore tuple cannot be changed.

The objects are enclosed within parenthesis and separated by comma.

Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List have mutable objects whereas Tuple have immutable objects.

eg:

1. `>>> data=(10,20,'ram',56.8)`
2. `>>> data2="a",10,20.9`
3. `>>> data`
4. `(10, 20, 'ram', 56.8)`
5. `>>> data2`
6. `('a', 10, 20.9)`
7. `>>>`

NOTE: If Parenthesis is not given with a sequence, it is by default treated as Tuple.

There can be an empty Tuple also which contains no object.

eg:

1. `tuple1=()`

For a single valued tuple, there must be a comma at the end of the value.

eg:

1. `Tuple1=(10,)`

Tuples can also be nested.

eg:

1. `tpl1='a','mahesh',10.56`
2. `tpl2=tuple1,(10,20,30)`
3. `print tpl1`
4. `print tpl2`

Output:

1. `>>>`
2. `('a', 'mahesh', 10.56)`
3. `(('a', 'mahesh', 10.56), (10, 20, 30))`
4. `>>>`

Accessing Tuple

Tuple can be accessed in the same way as List.

Some examples are given below:

eg:

1. data1=(1,2,3,4)
2. data2=('x','y','z')
3. **print** data1[0]
4. **print** data1[0:2]
5. **print** data2[-3:-1]
6. **print** data1[0:]
7. **print** data2[:2]

Output:

1. >>>
2. 1
3. (1, 2)
4. ('x', 'y')
5. (1, 2, 3, 4)
6. ('x', 'y')
7. >>>

Elements in a Tuple

Data=(1,2,3,4,5,10,19,17)

Forward indexing

→ 0 1 2 3 4 5 6 7

1	2	3	4	5	10	19	17
---	---	---	---	---	----	----	----

-8 -7 -6 -5 -4 -3 -2 -1

javatpoint.com

← Backward index

1. Data[0]=1=Data[-8] , Data[1]=2=Data[-7] , Data[2]=3=Data[-6] ,
2. Data[3]=4=Data[-5] , Data[4]=5=Data[-4] , Data[5]=10=Data[-3],
3. Data[6]=19=Data[-2],Data[7]=17=Data[-1]

Tuple Operations

Various Operations can be performed on Tuple. Operations performed on Tuple are given as:

a) Adding Tuple:

Tuple can be added by using the concatenation operator(+) to join two tuples.

eg:

1. data1=(1,2,3,4)
2. data2=('x','y','z')
3. data3=data1+data2
4. **print** data1
5. **print** data2
6. **print** data3

Output:

```
>>>
(1, 2, 3, 4)
('x', 'y', 'z')
(1, 2, 3, 4, 'x', 'y', 'z')
>>>
```

Note: The new sequence formed is a new Tuple.

b) Replicating Tuple:

Replicating means repeating. It can be performed by using '*' operator by a specific number of time.

Eg:

1. tuple1=(10,20,30);
2. tuple2=(40,50,60);
3. **print** tuple1*2
4. **print** tuple2*3

Output:

```
>>>
(10, 20, 30, 10, 20, 30)
(40, 50, 60, 40, 50, 60, 40, 50, 60)
>>>
```

c) Tuple slicing:

A subpart of a tuple can be retrieved on the basis of index. This subpart is known as tuple slice.

Eg:

1. data1=(1,2,4,5,7)
2. **print** data1[0:2]
3. **print** data1[4]
4. **print** data1[:-1]
5. **print** data1[-5:]
6. **print** data1

Output:

```
>>>
(1, 2)
7
(1, 2, 4, 5)
(1, 2, 4, 5, 7)
(1, 2, 4, 5, 7)
>>>
```

Note: If the index provided in the Tuple slice is outside the list, then it raises an IndexError exception.

Other Operations:

a) Updating elements in a List:

Elements of the Tuple cannot be updated. This is due to the fact that Tuples are immutable. Whereas the Tuple can be used to form a new Tuple.

Eg:

1. data=(10,20,30)
2. data[0]=100
3. **print** data

Output:

```
>>>
Traceback (most recent call last):
  File "C:/Python27/t.py", line 2, in
    data[0]=100
TypeError: 'tuple' object does not support item assignment
>>>
```

Creating a new Tuple from existing:

Eg:

1. data1=(10,20,30)
2. data2=(40,50,60)
3. data3=data1+data2
4. **print** data3

Output:

```
>>>
(10, 20, 30, 40, 50, 60)
>>>
```

b) Deleting elements from Tuple:

Deleting individual element from a tuple is not supported. However the whole of the tuple can be deleted using the del statement.

Eg:

1. data=(10,20,'rahul',40.6,'z')
2. **print** data
3. **del** data #will delete the tuple data
4. **print** data #will show an error since tuple data is already deleted

Output:

```
>>>
(10, 20, 'rahul', 40.6, 'z')
Traceback (most recent call last):
  File "C:/Python27/t.py", line 4, in
    print data
NameError: name 'data' is not defined
>>>
```

Functions of Tuple:

There are following in-built Type Functions:

Function	Description
min(tuple)	Returns the minimum value from a tuple.
max(tuple)	Returns the maximum value from the tuple.
len(tuple)	Gives the length of a tuple
cmp(tuple1,tuple2)	Compares the two Tuples.
tuple(sequence)	Converts the sequence into tuple.

1) min(tuple):

Eg:

1. data=(10,20,'rahul',40.6,'z')
2. **print** min(data)

Output:

```
>>>
10
>>>
```

2) max(tuple):

Eg:

1. data=(10,20,'rahul',40.6,'z')
2. **print** max(data)

Output:

```
>>>  
z  
>>>
```

3) len(tuple):**Eg:**

1. data=(10,20,'rahul',40.6,'z')
2. **print** len(data)

Output:

```
>>>  
5  
>>>
```

4) cmp(tuple1,tuple2):

Explanation: If elements are of the same type, perform the comparison and return the result. If elements are different types, check whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

Eg:

1. data1=(10,20,'rahul',40.6,'z')
2. data2=(20,30,'sachin',50.2)
3. **print** cmp(data1,data2)
4. **print** cmp(data2,data1)
5. data3=(20,30,'sachin',50.2)
6. **print** cmp(data2,data3)

Output:

```
>>>  
-1  
1  
0  
>>>
```

5) tuple(sequence):**Eg:**

1. dat=[10,20,30,40]
2. data=tuple(dat)
3. **print** data

Output:

```
>>>  
(10, 20, 30, 40)  
>>>
```

Why Use Tuple?

1. Processing of Tuples are faster than Lists.
2. It makes the data safe as Tuples are immutable and hence cannot be changed.
3. Tuples are used for String formatting.

Python Dictionary

Dictionary is an unordered set of key and value pair.

It is an container that contains data, enclosed within curly braces.

The pair i.e., key and value is known as item.

The key passed in the item must be unique.

The key and the value is separated by a colon(:). This pair is known as item. Items are separated from each other by a comma(,). Different items are enclosed within a curly brace and this forms Dictionary.

eg:

1. data={100:'Ravi' ,101:'Vijay' ,102:'Rahul'}
2. **print** data

Output:

1. >>>
2. {100: 'Ravi', 101: 'Vijay', 102: 'Rahul'}
3. >>>

Dictionary is mutable i.e., value can be updated.

Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.

Dictionary is known as Associative array since the Key works as Index and they are decided by the user.

eg:

1. plant={}
2. plant[1]='Ravi'
3. plant[2]='Manoj'
4. plant['name']='Hari'
5. plant[4]='Om'
6. **print** plant[2]
7. **print** plant['name']
8. **print** plant[1]
9. **print** plant

Output:

```
>>>
Manoj
Hari
Ravi
{1: 'Ravi', 2: 'Manoj', 4: 'Om', 'name': 'Hari'}
>>>
```

Accessing Values

Since Index is not defined, a Dictionaries value can be accessed by their keys.

Syntax:

```
[key]
```

Eg:

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
print "Id of 1st employer is",data1['Id']
print "Id of 2nd employer is",data2['Id']
print "Name of 1st employer:",data1['Name']
print "Profession of 2nd employer:",data2['Profession']
```

Output:

```
>>>
Id of 1st employer is 100
Id of 2nd employer is 101
Name of 1st employer is Suresh
Profession of 2nd employer is Trainer
>>>
```

Updation

The item i.e., key-value pair can be updated. Updating means new item can be added. The values can be modified.

Eg:

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
data1['Profession']='Manager'
data2['Salary']=20000
data1['Salary']=15000
print data1
print data2
```

Output:

```
>>>
{'Salary': 15000, 'Profession': 'Manager','Id': 100, 'Name': 'Suresh'}
{'Salary': 20000, 'Profession': 'Trainer', 'Id': 101, 'Name': 'Ramesh'}
>>>
```

Deletion

del statement is used for performing deletion operation.

An item can be deleted from a dictionary using the key.

Syntax:

```
del [key]
```

Whole of the dictionary can also be deleted using the del statement.

Eg:


```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
del data[102]
print data
del data
print data    #will show an error since dictionary is deleted.
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj'}

Traceback (most recent call last):
  File "C:/Python27/dict.py", line 5, in
    print data
NameError: name 'data' is not defined
>>>
```

Functions and Methods

Python Dictionary supports the following Functions:

Dictionary Functions:

Functions	Description
len(dictionary)	Gives number of items in a dictionary.
cmp(dictionary1,dictionary2)	Compares the two dictionaries.
str(dictionary)	Gives the string representation of a string.

Dictionary Methods:

Methods	Description
keys()	Return all the keys element of a dictionary.
values()	Return all the values element of a dictionary.
items()	Return all the items(key-value pair) of a dictionary.
update(dictionary2)	It is used to add items of dictionary2 to first dictionary.
clear()	It is used to remove all items of a dictionary. It returns an
fromkeys(sequence,value1)/ fromkeys(sequence)	It is used to create a new dictionary from the sequence wh forms the key and all keys share the values ?value1?. In c

	it set the values of keys to be none.
copy()	It returns an ordered copy of the data.
has_key(key)	It returns a boolean value. True in case if key is present false.
get(key)	Returns the value of the given key. If key is not present it r

Functions:

1) len(dictionary):

Eg:

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
print data
print len(data)
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
3
>>>
```

2) cmp(dictionary1,dictionary2):

Explanation:

The comparison is done on the basis of key and value.

```
If, dictionary1 == dictionary2, returns 0.
    dictionary1 < dictionary2, returns -1.
    dictionary1 > dictionary2, returns 1.
```

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
data2={103:'abc', 104:'xyz', 105:'mno'}
data3={'Id':10, 'First':'Aman','Second':'Sharma'}
data4={100:'Ram', 101:'Suraj', 102:'Alok'}
print cmp(data1,data2)
print cmp(data1,data4)
print cmp(data3,data2)
```

Output:

```
>>>
-1
0
1
>>>
```

3) str(dictionary):

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print str(data1)
```

Output:

```
>>>  
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}  
>>>
```

Methods:

1) keys():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.keys()
```

Output:

```
>>>  
[100, 101, 102]  
>>>
```

2) values():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.values()
```

Output:

```
>>>  
['Ram', 'Suraj', 'Alok']  
>>>
```

3) items():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.items()
```

Output:

```
>>>  
[(100, 'Ram'), (101, 'Suraj'), (102, 'Alok')]  
>>>
```

4) update(dictionary2):

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
data2={103:'Sanjay'}  
data1.update(data2)  
print data1  
print data2
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok', 103: 'Sanjay'}
{103: 'Sanjay'}
>>>
```

5) clear():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print data1
data1.clear()
print data1
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
{}
>>>
```

6) fromkeys(sequence)/ fromkeys(seq,value):

Eg:

```
sequence=('Id' , 'Number' , 'Email')
data={}
data1={}
data=data.fromkeys(sequence)
print data
data1=data1.fromkeys(sequence,100)
print data1
```

Output:

```
>>>
{'Email': None, 'Id': None, 'Number': None}
{'Email': 100, 'Id': 100, 'Number': 100}
>>>
```

7) copy():

Eg:

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
data1=data.copy()
print data1
```

Output:

```
>>>
{'Age': 23, 'Id': 100, 'Name': 'Aakash'}
>>>
```

8) has_key(key):

Eg:

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
print data.has_key('Age')
print data.has_key('Email')
```

Output:

```
>>>
True
False
>>>
```

9) get(key):**Eg:**

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
print data.get('Age')
print data.get('Email')
```

Output:

```
>>>
23
None
>>>
```

Python OOPs Concepts

Python is an object-oriented programming language. You can easily create and use classes and objects in Python.

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

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

Object

Object is an entity that has state and behavior. It may be anything. It may be physical and logical. For example: 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

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. .
6. <statement-N>

Method

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

Inheritance

Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviors of parent object. By using inheritance you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class.

It provides re-usability of the code.

Polymorphism

Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways. For example: You have a class animal and all animals talk. But they talk differently. Here, the "talk" behavior is polymorphic in the sense and totally depends on the animal. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

Encapsulation

Encapsulation is also the feature 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 an approach to problem solving where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when project becomes lengthy.
3.	It simulates the real world entity. So real world problems can be easily solved	It doesn't simulate the real world. It works on step by step instructions

through oops.

divided in 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 are: C++, Java, .Net, Python, C# etc.

Example of procedural languages are: C, Fortran, Pascal, VB etc.

Python Object

Python is an object oriented programming language. So its main focus is on objects unlike procedure oriented programming languages which mainly focuses on functions.

In object oriented programming language, object is simply a collection of data (variables) and methods (functions) that act on those data.

Python Class

A class is a blueprint for the object. 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 another buildings (as many as we want) based on these details. So building is a class and we can create many objects from a class.

An object is also called an instance of a class and the process of creating this object is known as instantiation.

Python classes contain all the standard features of Object Oriented Programming. A python class is a mixture of class mechanism of C++ and Modula-3.

Define a class in Python

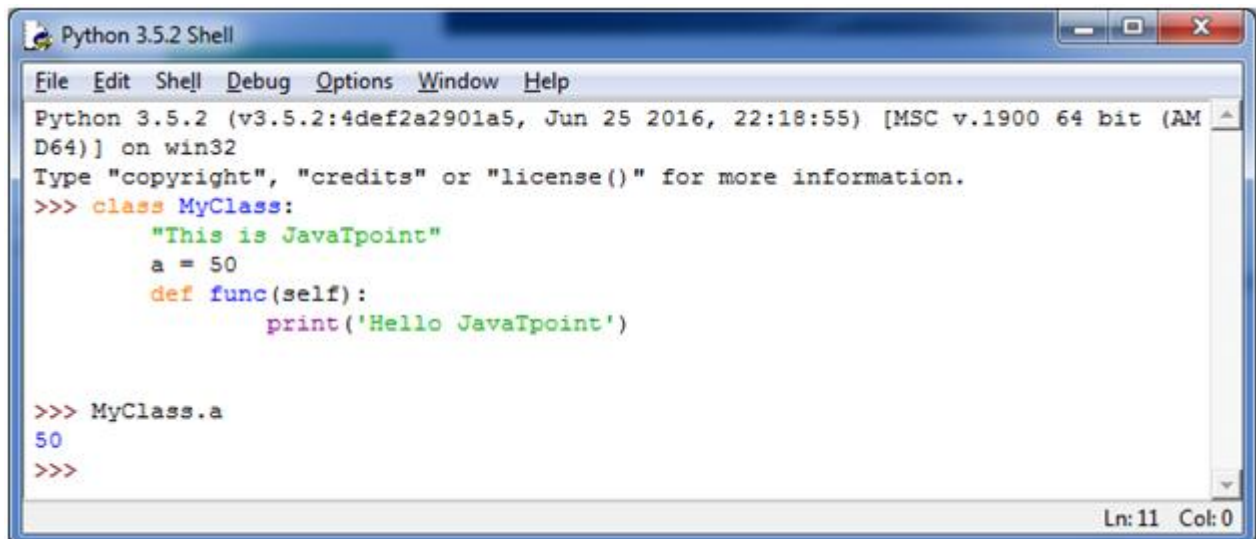
In Python, a class is defined by using a keyword class like a function definition begins with the keyword def.

Syntax of a class definition:

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

A class creates a new local namespace to define its all attribute. These attributes may be data or functions.

See this example:

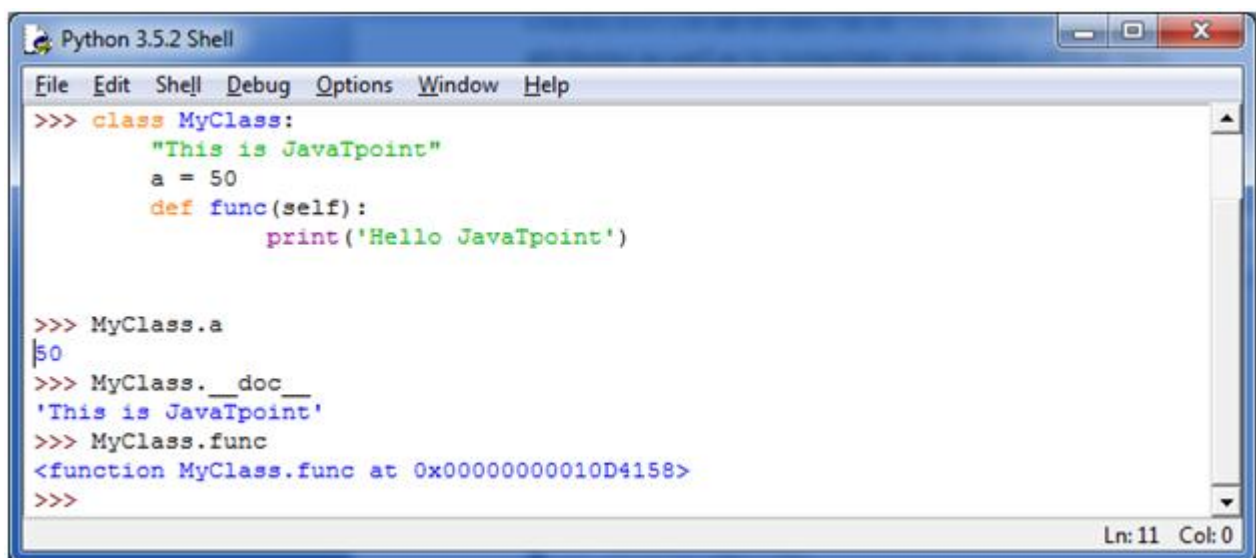


```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    "This is JavaTpoint"
    a = 50
    def func(self):
        print('Hello JavaTpoint')

>>> MyClass.a
50
>>>
```

There are also some special attributes that begins with double underscore (__). For example: `__doc__` attribute. It is used to fetch the docstring of that class. When we define a class, a new class object is created with the same class name. This new class object provides a facility to access the different attributes as well as to instantiate new objects of that class.

See this example:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> class MyClass:
    "This is JavaTpoint"
    a = 50
    def func(self):
        print('Hello JavaTpoint')

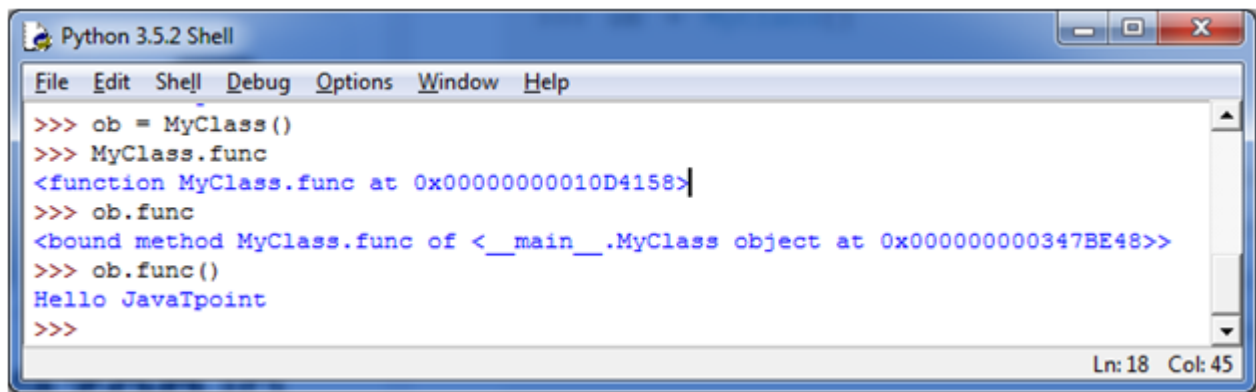
>>> MyClass.a
50
>>> MyClass.__doc__
'This is JavaTpoint'
>>> MyClass.func
<function MyClass.func at 0x00000000010D4158>
>>>
```

Create an Object in Python

We can create new object instances of the classes. The procedure to create an object is similar to a function call.

Let's take an example to create a new instance object "ob". We can access attributes of objects by using the object name prefix.

See this example:

A screenshot of a Python 3.5.2 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following code and output:

```
>>> ob = MyClass()
>>> MyClass.func
<function MyClass.func at 0x00000000010D4158>
>>> ob.func
<bound method MyClass.func of <__main__.MyClass object at 0x000000000347BE48>>
>>> ob.func()
Hello JavaTpoint
>>>
```

The status bar at the bottom right indicates 'Ln: 18 Col: 45'.

Here, attributes may be data or method. Method of an object is corresponding functions of that class. For example: `MyClass.func` is a function object and `ob.func` is a method object.

Python Object Class Example

1. **class** Student:
2. **def** `__init__`(self, rollno, name):
3. self.rollno = rollno
4. self.name = name
5. **def** displayStudent(self):
6. **print** "rollno : ", self.rollno, ", name: ", self.name
7. emp1 = Student(121, "Ajeet")
8. emp2 = Student(122, "Sonoo")
9. emp1.displayStudent()
10. emp2.displayStudent()

Output:

1. rollno : 121 , name: Ajeet
2. rollno : 122 , name: Sonoo

Python Constructors

A constructor is a special type of method (function) that is called when it instantiates an object using the definition found in your class. The constructors are normally used to initialize (assign values) to the instance variables. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating a constructor:

A constructor is a class function that begins with double underscore (`__`). The name of the constructor is always the same `__init__`().

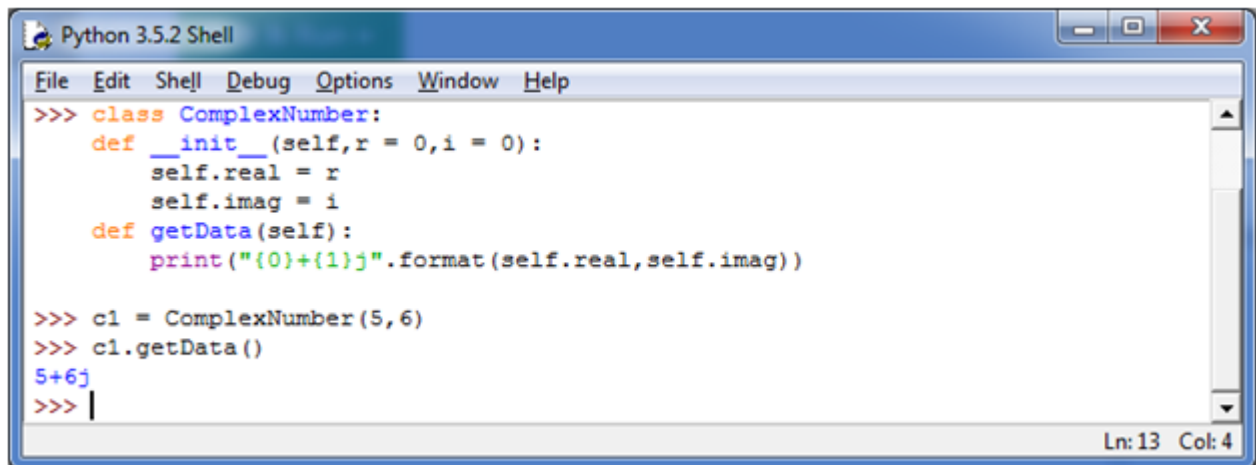
While creating an object, a constructor can accept arguments if necessary. When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default constructor.

Let's take an example:

Let's create a class named `ComplexNumber`, having two functions `__init__`() function to initialize the variable and `getData`() to display the number properly.

See this example:

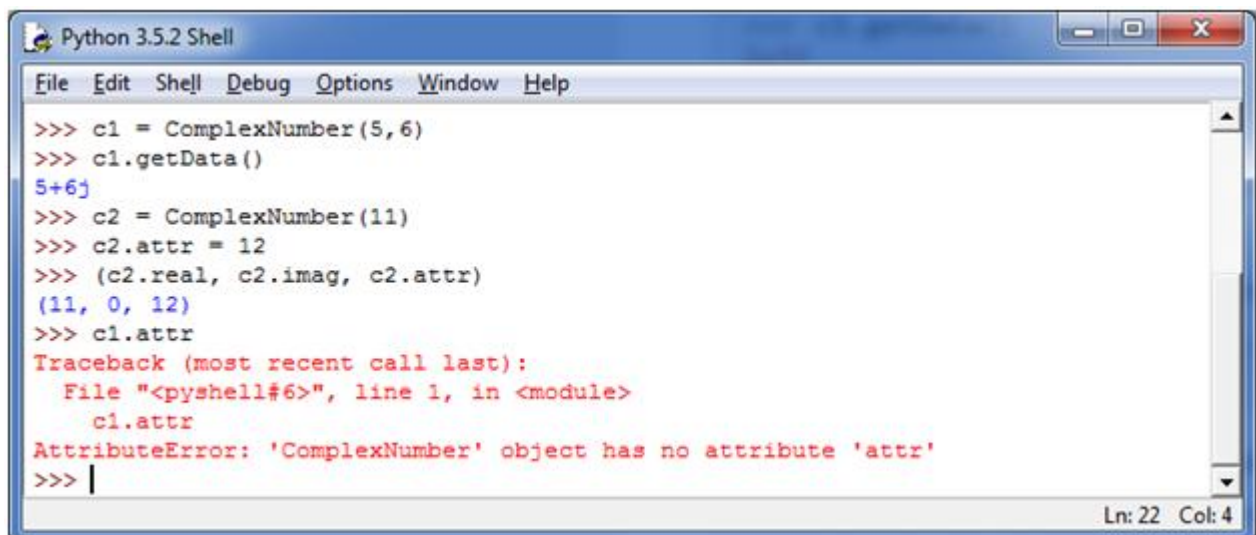


```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> class ComplexNumber:
>>>     def __init__(self, r = 0, i = 0):
>>>         self.real = r
>>>         self.imag = i
>>>     def getData(self):
>>>         print("{0}+{1}j".format(self.real, self.imag))
>>>
>>> c1 = ComplexNumber(5, 6)
>>> c1.getData()
5+6j
>>> |
```

Ln: 13 Col: 4

You can create a new attribute for an object and read it well at the time of defining the values. But you can't create the attribute for already defined objects.

See this example:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> c1 = ComplexNumber(5, 6)
>>> c1.getData()
5+6j
>>> c2 = ComplexNumber(11)
>>> c2.attr = 12
>>> (c2.real, c2.imag, c2.attr)
(11, 0, 12)
>>> c1.attr
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    c1.attr
AttributeError: 'ComplexNumber' object has no attribute 'attr'
>>> |
```

Ln: 22 Col: 4

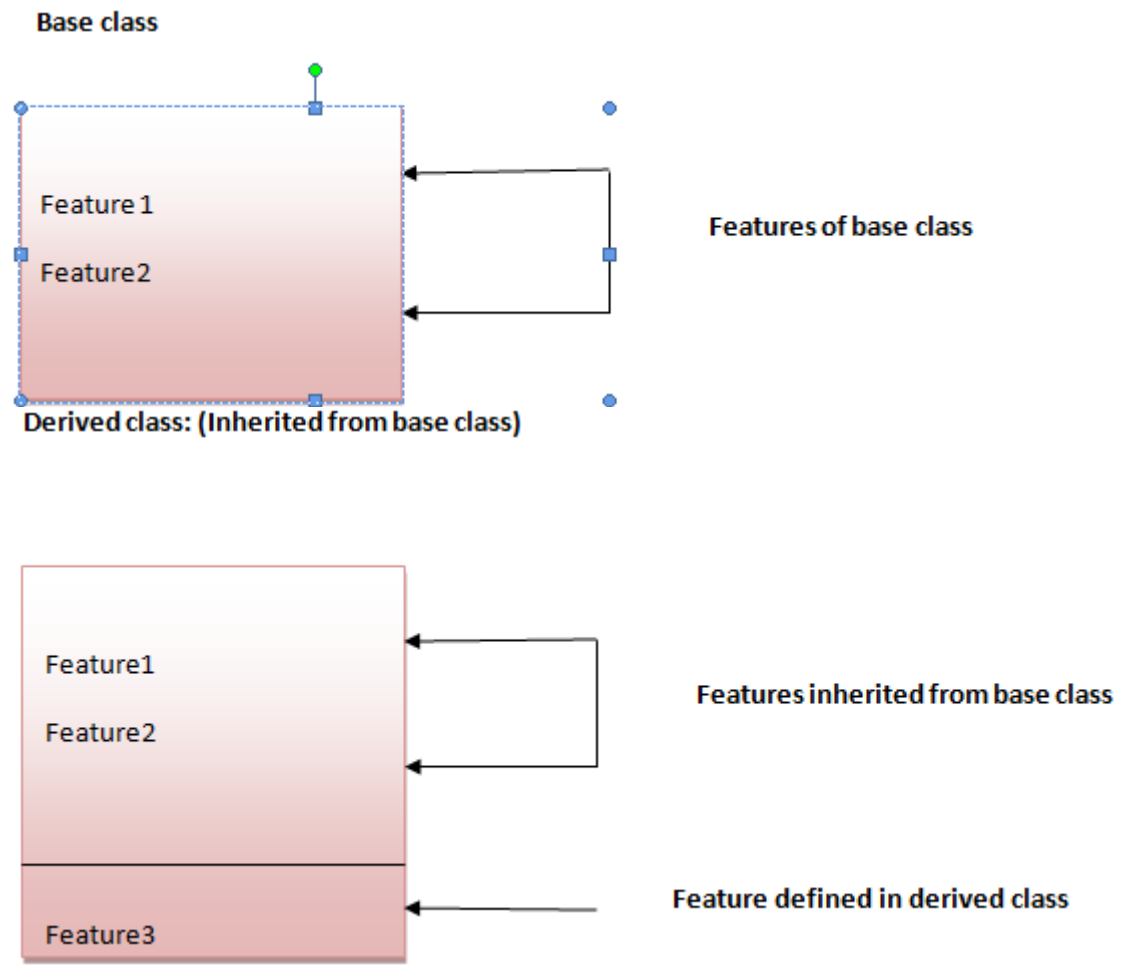
Inheritance in Python

What is Inheritance

Inheritance is used to specify that one class will get most or all of its features from its parent class. It is a feature of Object Oriented Programming. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class. The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.

The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates re-usability of code.

Image representation:



Syntax 1:

1. **class** DerivedClassName(BaseClassName):
2. <statement-1>
3. .
4. .
5. .
6. <statement-N>

Syntax 2:

1. **class** DerivedClassName(modulename.BaseClassName):
2. <statement-1>
3. .
4. .
5. .
6. <statement-N>

Parameter explanation:

The name BaseClassName must be defined in a scope containing the derived class definition. You can also use other arbitrary expressions in place of a base class name. This is used when the base class is defined in another module.

Python Inheritance Example

Let's see a simple python inheritance example where we are using two classes: Animal and Dog. Animal is the parent or base class and Dog is the child class.

Here, we are defining eat() method in Animal class and bark() method in Dog class. In this example, we are creating instance of Dog class and calling eat() and bark() methods by the instance of child class only. Since, parent properties and behaviors are inherited to child object automatically, we can call parent and child class methods by the child instance only.

1. **class** Animal:
2. **def** eat(self):
3. **print** 'Eating...'
4. **class** Dog(Animal):
5. **def** bark(self):
6. **print** 'Barking...'
7. d=Dog()
8. d.eat()
9. d.bark()

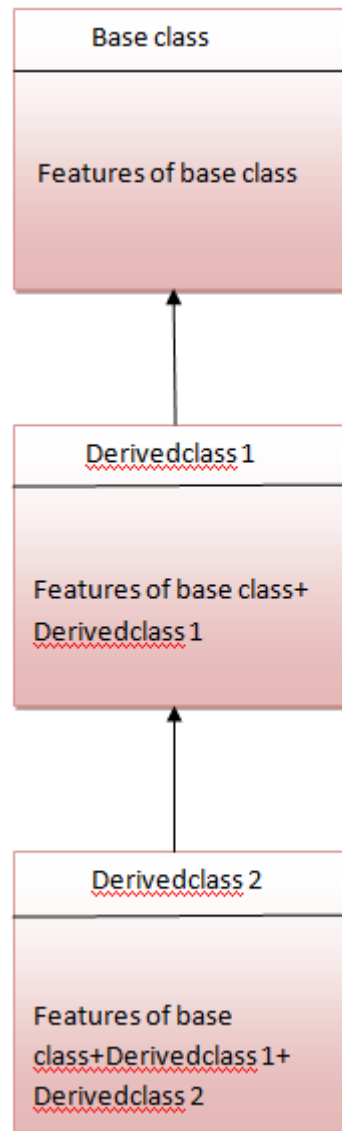
Output:

1. Eating...
2. Barking...

Multilevel Inheritance in Python

Multilevel inheritance is also possible in Python unlike other programming languages. You can inherit a derived class from another derived class. This is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

Image representation:



Python Multilevel Inheritance Example

```
1. class Animal:
2.     def eat(self):
3.         print 'Eating...'
4. class Dog(Animal):
5.     def bark(self):
6.         print 'Barking...'
7. class BabyDog(Dog):
8.     def weep(self):
9.         print 'Weeping...'
10. d=BabyDog()
11. d.eat()
12. d.bark()
13. d.weep()
```

Output:

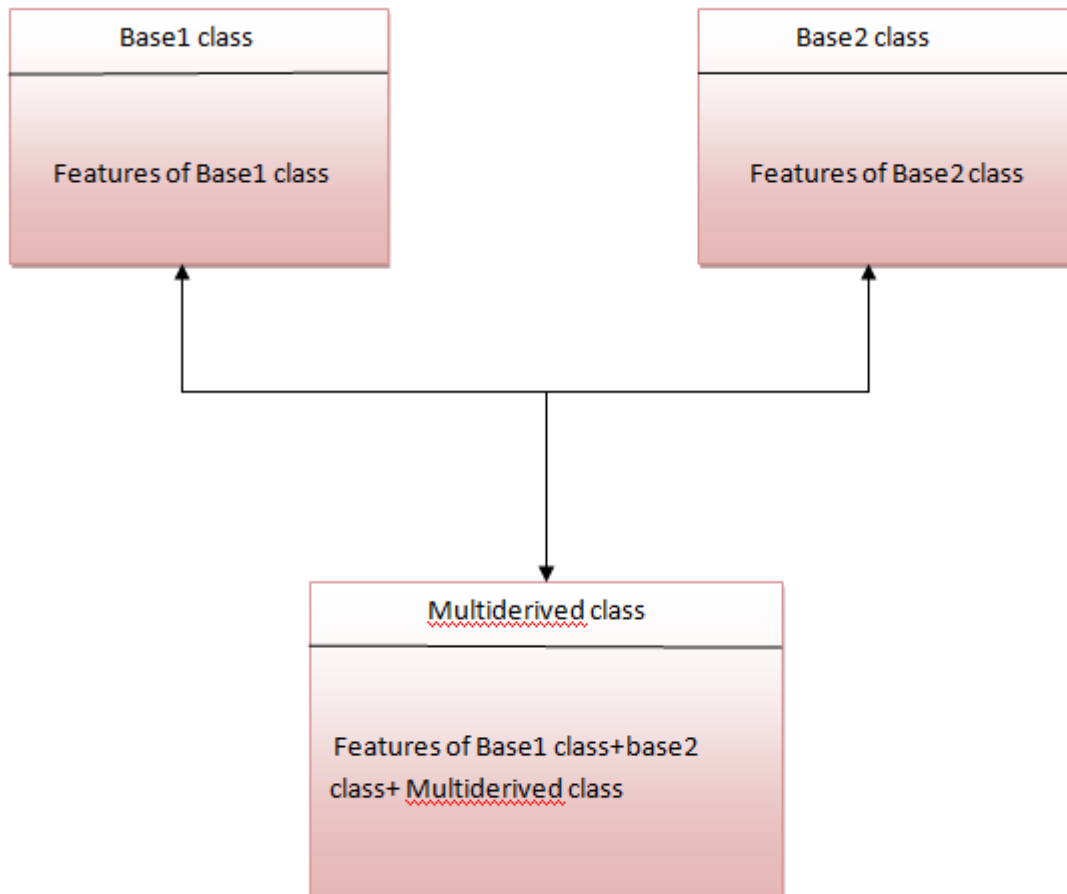
```
1. Eating...
```

2. Barking...
3. Weeping

Multiple Inheritance in Python

Python supports multiple inheritance also. You can derive a child class from more than one base (parent) class.

Image representation:



The multiderived class inherits the properties of both class base1 and base2.

Let's see the syntax of multiple inheritance in Python.

Syntax:

1. **class** DerivedClassName(Base1, Base2, Base3):
2. <statement-1>
3. .
4. .
5. .
6. <statement-N>

Or

1. **class** Base1:

```
2.     pass
3.
4. class Base2:
5.     pass
6.
7. class MultiDerived(Base1, Base2):
8.     pass
```

Example:

```
1. class First(object):
2.     def __init__(self):
3.         super(First, self).__init__()
4.         print("first")
5.
6. class Second(object):
7.     def __init__(self):
8.         super(Second, self).__init__()
9.         print("second")
10.
11. class Third(Second, First):
12.     def __init__(self):
13.         super(Third, self).__init__()
14.         print("third")
15.
16. Third();
```

Output:

```
1. first
2. second
3. third
```

Why super () keyword

The most commonly super() is used with __init__ function in base classes. This is usually the only place where you need to do some things in a child then complete the initialization in the parent.

See this example:

```
1. class Child(Parent):
2.     def __init__(self, stuff):
3.         self.stuff = stuff
4.         super(Child, self).__init__()
```

Composition in Python

Composition is used to do the same thing which can be done by inheritance.

Python Functions

A Function is a self block of code.

A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.

A Function is a subprogram that works on data and produce some output.

Types of Functions:

There are two types of Functions.

a) Built-in Functions: Functions that are predefined. We have used many predefined functions in Python.

b) User- Defined: Functions that are created according to the requirements.

Defining a Function:

A Function defined in Python should follow the following format:

1) Keyword `def` is used to start the Function Definition. `def` specifies the starting of Function block.

2) `def` is followed by function-name followed by parenthesis.

3) Parameters are passed inside the parenthesis. At the end a colon is marked.

Syntax:

1. `def <function_name>([parameters]):`
2. `</function_name>`

eg:

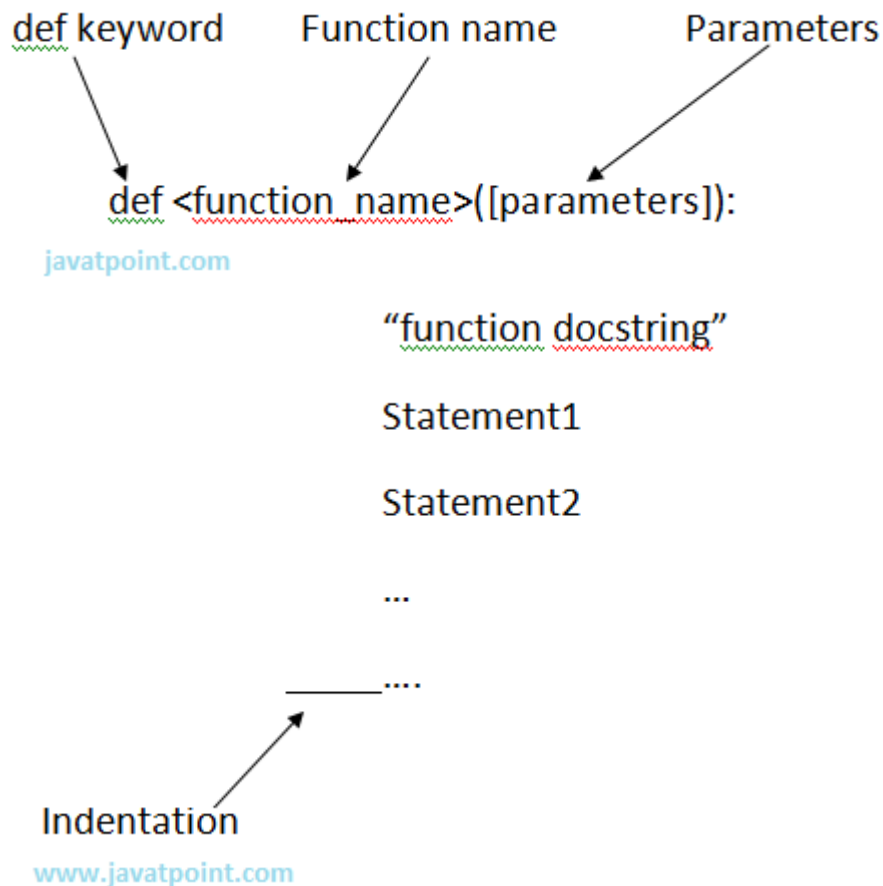
1. `def sum(a,b):`

4) Before writing a code, an Indentation (space) is provided before every statement. It should be same for all statements inside the function.

5) The first statement of the function is optional. It is ?Documentation string? of function.

6) Following is the statement to be executed.

Syntax:



Invoking a Function:

To execute a function it needs to be called. This is called function calling.

Function Definition provides the information about function name, parameters and the definition what operation is to be performed. In order to execute the Function Definition it is to be called.

Syntax:

1. <function_name>(parameters)
2. </function_name>

eg:

1. sum(a,b)

here sum is the function and a, b are the parameters passed to the Function Definition.

Let's have a look over an example:

eg:

1. #Providing Function Definition
2. **def** sum(x,y):
3. "Going to add x and y"
4. s=x+y
5. **print** "Sum of two numbers is"

```
6.     print s
7.     #Calling the sum Function
8.     sum(10,20)
9.     sum(20,30)
```

Output:

```
1. >>>
2. Sum of two numbers is
3. 30
4. Sum of two numbers is
5. 50
6. >>>
```

NOTE: Function call will be executed in the order in which it is called.

return Statement:

return[expression] is used to send back the control to the caller with the expression.

In case no expression is given after return it will return None.

In other words return statement is used to exit the Function definition.

Eg:

```
1. def sum(a,b):
2.     "Adding the two values"
3.     print "Printing within Function"
4.     print a+b
5.     return a+b
6. def msg():
7.     print "Hello"
8.     return
9.
10. total=sum(10,20)
11. print "Printing Outside: ",total
12. msg()
13. print "Rest of code"
```

Output:

```
1. >>>
2. Printing within Function
3. 30
4. Printing outside: 30
5. Hello
```

6. Rest of code

7. >>>

Argument and Parameter:

There can be two types of data passed in the function.

1) The First type of data is the data passed in the function call. This data is called ?arguments?.

2) The second type of data is the data received in the function definition. This data is called ?parameters?.

Arguments can be literals, variables and expressions.

Parameters must be variable to hold incoming values.

Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

Eg:

1. **def** addition(x,y):
2. **print** x+y
3. x=15
4. addition(x ,10)
5. addition(x,x)
6. y=20
7. addition(x,y)

Output:

1. >>>
2. 25
3. 30
4. 35
5. >>>

Passing Parameters

Apart from matching the parameters, there are other ways of matching the parameters.

Python supports following types of formal argument:

- 1) Positional argument (Required argument).
- 2) Default argument.
- 3) Keyword argument (Named argument)

Positional/Required Arguments:

When the function call statement must match the number and order of arguments as defined in the function definition it is Positional Argument matching.

Eg:

1. `#Function definition of sum`
2. `def sum(a,b):`
3. `"Function having two parameters"`
4. `c=a+b`
5. `print c`
- 6.
7. `sum(10,20)`
8. `sum(20)`

Output:

1. `>>>`
2. `30`
- 3.
4. `Traceback (most recent call last):`
5. `File "C:/Python27/su.py", line 8, in <module>`
6. `sum(20)`
7. `TypeError: sum() takes exactly 2 arguments (1 given)`
8. `>>>`
9. `</module>`

Explanation:

1) In the first case, when `sum()` function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and printed.

2) In the second case, when `sum()` function is called passing a single value i.e., 20 , it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

Default Arguments

Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.

Eg:

1. `#Function Definition`
2. `def msg(Id,Name,Age=21):`
3. `"Printing the passed value"`
4. `print Id`
5. `print Name`
6. `print Age`
7. `return`
8. `#Function call`
9. `msg(Id=100,Name='Ravi',Age=20)`
10. `msg(Id=101,Name='Ratan')`

Output:

1. >>>
2. 100
3. Ravi
4. 20
5. 101
6. Ratan
7. 21
8. >>>

Explanation:

1) In first case, when msg() function is called passing three different values i.e., 100 , Ravi and 20, these values will be assigned to respective parameters and thus respective values will be printed.

2) In second case, when msg() function is called passing two values i.e., 101 and Ratan, these values will be assigned to Id and Name respectively. No value is assigned for third argument via function call and hence it will retain its default value i.e, 21.

Keyword Arguments:

Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

Eg:

1. **def** msg(id,name):
2. "Printing passed value"
3. **print** id
4. **print** name
5. **return**
6. msg(id=100,name='Raj')
7. msg(name='Rahul',id=101)

Output:

1. >>>
2. 100
3. Raj
4. 101
5. Rahul
6. >>>

Explanation:

1) In the first case, when msg() function is called passing two values i.e., id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.

2) In second case, when msg() function is called passing two values i.e., name and id, although the position of two parameters is different it initialize the value of id in Function call to id in

Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

Anonymous Function:

Anonymous Functions are the functions that are not bond to name.

Anonymous Functions are created by using a keyword "lambda".

Lambda takes any number of arguments and returns an evaluated expression.

Lambda is created without using the def keyword.

Syntax:

1. **lambda** arg1,args2,args3,?,argsn :expression

Output:

1. #Function Definiton
2. square=**lambda** x1: x1*x1
- 3.
4. #Calling square as a function
5. **print** "Square of number is",square(10)

Output:

1. >>>
2. Square of number **is** 100
3. >>>

Difference between Normal Functions and Anonymous Function:

Have a look over two examples:

Eg:

Normal function:

1. #Function Definiton
2. **def** square(x):
3. **return** x*x
- 4.
5. #Calling square function
6. **print** "Square of number is",square(10)

Anonymous function:

1. #Function Definiton
2. square=**lambda** x1: x1*x1
- 3.

4. `#Calling square as a function`
5. `print "Square of number is",square(10)`

Explanation:

Anonymous is created without using def keyword.

lambda keyword is used to create anonymous function.

It returns the evaluated expression.

Scope of Variable:

Scope of a variable can be determined by the part in which variable is defined. Each variable cannot be accessed in each part of a program. There are two types of variables based on Scope:

- 1) Local Variable.
- 2) Global Variable.

1) Local Variables:

Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Eg:

1. `def msg():`
2. `a=10`
3. `print "Value of a is",a`
4. `return`
- 5.
6. `msg()`
7. `print a` `#it will show error since variable is local`

Output:

1. `>>>`
2. Value of a **is** 10
- 3.
4. Traceback (most recent call last):
5. File "C:/Python27/lam.py", line 7, in <module>
6. `print a` `#it will show error since variable is local`
7. NameError: name 'a' **is not** defined
8. `>>>`
9. `</module>`

b) Global Variable:

Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

Eg:

```
1. b=20
2. def msg():
3.     a=10
4.     print "Value of a is",a
5.     print "Value of b is",b
6.     return
7.
8.     msg()
9.     print b
```

Output:

```
1. >>>
2. Value of a is 10
3. Value of b is 20
4. 20
5. >>>
```

Python Input And Output

Python can be used to read and write data. Also it supports reading and writing data to Files.

"print" statement:

"print" statement is used to print the output on the screen.

print statement is used to take string as input and place that string to standard output.

Whatever you want to display on output place that expression inside the inverted commas. The expression whose value is to be printed place it without inverted commas.

Syntax:

1. **print** "expression" or **print** expression.

eg:

1. a=10
2. **print** "Welcome to the world of Python"
3. **print** a

Output:

1. >>>
2. Welcome to the world of Python
3. 10
4. >>>

Input from Keyboard:

Python offers two in-built functions for taking input from user. They are:

1) input()

2) raw_input()

1) input() function input() function is used to take input from the user. Whatever expression is given by the user, it is evaluated and result is returned back.

Syntax:

1. input("Expression")

eg:

1. n=input("Enter your expression ");
2. **print** "The evaluated expression is ", n

Output:

1. >>>

2. Enter your expression 10*2
3. The evaluated expression **is** 20
4. >>>

2) raw_input() raw_input() function is used to take input from the user. It takes the input from the Standard input in the form of a string and reads the data from a line at once.

Syntax:

1. raw_input(?statement?)

eg:

1. n=raw_input("Enter your name ");
2. **print** "Welcome ", n

Output:

1. >>>
2. Enter your name Rajat
3. Welcome Rajat
4. >>>

raw_input() function returns a string. Hence in case an expression is to be evaluated, then it has to be type casted to its following data type. Some of the examples are given below:

Program to calculate Simple Interest.

1. prn=int(raw_input("Enter Principal"))
2. r=int(raw_input("Enter Rate"))
3. t=int(raw_input("Enter Time"))
4. si=(prn*r*t)/100
5. **print** "Simple Interest is ",si

Output:

1. >>>
2. Enter Principal1000
3. Enter Rate10
4. Enter Time2
5. Simple Interest **is** 200
6. >>>

Program to enter details of an user and print them.

1. name=raw_input("Enter your name ")
2. math=float(raw_input("Enter your marks in Math"))
3. physics=float(raw_input("Enter your marks in Physics"))
4. chemistry=float(raw_input("Enter your marks in Chemistry"))

5. `rollno=int(raw_input("Enter your Roll no"))`
6. `print "Welcome ",name`
7. `print "Your Roll no is ",rollno`
8. `print "Marks in Maths is ",math`
9. `print "Marks in Physics is ",physics`
10. `print "Marks in Chemistry is ",chemistry`
11. `print "Average marks is ",(math+physics+chemistry)/3`

Output:

1. `>>>`
2. Enter your name rajat
3. Enter your marks **in** Math76.8
4. Enter your marks **in** Physics71.4
5. Enter your marks **in** Chemistry88.4
6. Enter your Roll no0987645672
7. Welcome rajat
8. Your Roll no **is** 987645672
9. Marks **in** Maths **is** 76.8
10. Marks **in** Physics **is** 71.4
11. Marks **in** Chemistry **is** 88.4
12. Average marks **is** 78.8666666667
13. `>>>`

File Handling:

Python provides the facility of working on Files. A File is an external storage on hard disk from where data can be stored and retrieved.

Operations on Files:

1) Opening a File: Before working with Files you have to open the File. To open a File, Python built in function `open()` is used. It returns an object of File which is used with other functions. Having opened the file now you can perform read, write, etc. operations on the File.

Syntax:

1. `obj=open(filename , mode , buffer)`

here,

filename:It is the name of the file which you want to access.

mode:It specifies the mode in which File is to be opened. There are many types of mode. Mode depends the operation to be performed on File. Default access mode is read.

2) Closing a File:Once you are finished with the operations on File at the end you need to close the file. It is done by the `close()` method. `close()` method is used to close a File.

Syntax:

1. `fileobject.close()`

3) Writing to a File:write() method is used to write a string into a file.

Syntax:

1. fileobject.write(string str)

4) Reading from a File:read() method is used to read data from the File.

Syntax:

1. fileobject.read(value)

here, value is the number of bytes to be read. In case, no value is given it reads till end of file is reached.

Program to read and write data from a file.

1. obj=open("abcd.txt","w")
2. obj.write("Welcome to the world of Python")
3. obj.close()
4. obj1=open("abcd.txt","r")
5. s=obj1.read()
6. **print** s
7. obj1.close()
8. obj2=open("abcd.txt","r")
9. s1=obj2.read(20)
10. **print** s1
11. obj2.close()

Output:

1. >>>
2. Welcome to the world of Python
3. Welcome to the world
4. >>>

Attributes of File:

There are following File attributes.

Attribute	Description
Name	Returns the name of the file.
Mode	Returns the mode in which file is being opened.
Closed	Returns Boolean value. True, in case if file is closed else false.

Eg:

1. `obj = open("data.txt", "w")`
2. `print obj.name`
3. `print obj.mode`
4. `print obj.closed`

Output:

1. `>>>`
2. `data.txt`
3. `w`
4. `False`
5. `>>>`

Modes of File:

There are different modes of file in which it can be opened. They are mentioned in the following table.

A File can be opened in two modes:

- 1) Text Mode.
- 2) Binary Mode.

Mode	Description
R	It opens in Reading mode. It is default mode of File. Pointer is at beginning of the file.
rb	It opens in Reading mode for binary format. It is the default mode. Pointer is at beginning of the file.
r+	Opens file for reading and writing. Pointer is at beginning of file.
rb+	Opens file for reading and writing in binary format. Pointer is at beginning of file.
W	Opens file in Writing mode. If file already exists, then overwrite the file else create a new file.
wb	Opens file in Writing mode in binary format. If file already exists, then overwrite the file else create a new file.
w+	Opens file for reading and writing. If file already exists, then overwrite the file else create a new file.
wb+	Opens file for reading and writing in binary format. If file already exists, then overwrite the file else create a new file.

a	Opens file in Appending mode. If file already exists, then append the data at the end of existing file, else create a new file.
ab	Opens file in Appending mode in binary format. If file already exists, then append the data at the end of existing file, else create a new file.
a+	Opens file in reading and appending mode. If file already exists, then append the data at the end of existing file, else create a new file.
ab+	Opens file in reading and appending mode in binary format. If file already exists, then append the data at the end of existing file, else create a new file.

Methods:

There are many methods related to File Handling. They are given in the following table:

There is a module "os" defined in Python that provides various functions which are used to perform various operations on Files. To use these functions 'os' needs to be imported.

Method	Description
rename()	It is used to rename a file. It takes two arguments, existing_file_name and new_file_name.
remove()	It is used to delete a file. It takes one argument. Pass the name of the file which is to be deleted as the argument of method.
mkdir()	It is used to create a directory. A directory contains the files. It takes one argument which is the name of the directory.
chdir()	It is used to change the current working directory. It takes one argument which is the name of the directory.
getcwd()	It gives the current working directory.
rmdir()	It is used to delete a directory. It takes one argument which is the name of the directory.
tell()	It is used to get the exact position in the file.

1) rename():

Syntax:

1. `os.rename(existing_file_name, new_file_name)`

eg:

```
1. import os
2. os.rename('mno.txt', 'pqr.txt')
```

2) remove():

Syntax:

1. `os.remove(file_name)`

eg:

```
1. import os
2. os.remove('mno.txt')
```

3) mkdir()

Syntax:

`os.mkdir("file_name")`

eg:

```
1. import os
2. os.mkdir("new")
```

4) chdir()

Syntax:

`os.chdir("file_name")`

eg:

```
1. import os
2. os.chdir("new")
```

5) getcwd()

Syntax:

`os.getcwd()`

eg:

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

6) rmdir()

Syntax:

```
os.rmdir("directory_name")
```

eg:

1. **import** os
2. os.rmdir("new")

NOTE: In order to delete a directory, it should be empty. In case directory is not empty first delete the files.

Python Moudule

Modules are used to categorize code in Python into smaller part. A module is simply a file, where classes, functions and variables are defined. Grouping similar code into a single file makes it easy to access.

Have a look over example:

If the content of a book is not indexed or categorized into individual chapters, then the book might have turned boring and hectic. Hence, dividing book into chapters made it easy to understand.

In the same sense python modules are the files which have similar code. Thus module is simplify a python code where classes, variables and functions are defined.

Advantage:

Python provides the following advantages for using module:

1) Reusability: Module can be used in some other python code. Hence it provides the facility of code reusability.

2) Categorization: Similar type of attributes can be placed in one module.

Importing a Module:

There are different ways by which you we can import a module. These are as follows:

1) Using import statement:

"import" statement can be used to import a module.

Syntax:

1. **import** <file_name1, file_name2,...file_name(n)="">
2. </file_name1,>

Have a look over an example:

eg:

1. **def** add(a,b):
2. c=a+b
3. **print** c
4. **return**

Save the file by the name addition.py. To import this file "import" statement is used.

1. **import** addition
2. addition.add(10,20)
3. addition.add(30,40)

Create another python file in which you want to import the former python file. For that, import statement is used as given in the above example. The corresponding method can be used by

file_name.method (). (Here, addition.add (), where addition is the python file and add () is the method defined in the file addition.py)

Output:

1. >>>
2. 30
3. 70
4. >>>

NOTE: You can access any function which is inside a module by module name and function name separated by dot. It is also known as period. Whole notation is known as dot notation.

Example of importing multiple modules:

Eg:

1) msg.py:

1. **def** msg_method():
2. **print** "Today the weather is rainy"
3. **return**

2) display.py:

1. **def** display_method():
2. **print** "The weather is Sunny"
3. **return**

3) multiimport.py:

1. **import** msg,display
2. msg.msg_method()
3. display.display_method()

Output:

1. >>>
2. Today the weather **is** rainy
3. The weather **is** Sunny
4. >>>

2) Using from.. import statement:

from..import statement is used to import particular attribute from a module. In case you do not want whole of the module to be imported then you can use from ?import statement.

Syntax:

1. **from** <module_name> **import** <attribute1,attribute2,attribute3,...attributen>
2. </attribute1,attribute2,attribute3,...attributen></module_name>

Have a look over the example:

1) area.py

Eg:

```
1. def circle(r):
2.     print 3.14*r*r
3.     return
4.
5. def square(l):
6.     print l*l
7.     return
8.
9. def rectangle(l,b):
10.    print l*b
11.    return
12.
13. def triangle(b,h):
14.    print 0.5*b*h
15.    return
```

2) area1.py

```
1. from area import square,rectangle
2. square(10)
3. rectangle(2,5)
```

Output:

```
1. >>>
2. 100
3. 10
4. >>>
```

3) To import whole module:

You can import whole of the module using "from? import *"

Syntax:

```
1. from <module_name> import *
2. </module_name>
```

Using the above statement all the attributes defined in the module will be imported and hence you can access each attribute.

1) area.py

Same as above example

2) area1.py

1. **from** area **import** *
2. square(10)
3. rectangle(2,5)
4. circle(5)
5. triangle(10,20)

Output:

1. >>>
2. 100
3. 10
4. 78.5
5. 100.0
6. >>>

Built in Modules in Python:

There are many built in modules in Python. Some of them are as follows:

math, random , threading , collections , os , mailbox , string , time , tkinter etc..

Each module has a number of built in functions which can be used to perform various functions.

Let's have a look over each module:

1) math:

Using math module , you can use different built in mathematical functions.

Functions:

Function	Description
ceil(n)	Returns the next integer number of the given number
sqrt(n)	Returns the Square root of the given number.
exp(n)	Returns the natural logarithm e raised to the given number
floor(n)	Returns the previous integer number of the given number.
log(n,baseto)	Returns the natural logarithm of the number.
pow(baseto, exp)	Returns baseto raised to the exp power.
sin(n)	Returns sine of the given radian.

cos(n)	Returns cosine of the given radian.
tan(n)	Returns tangent of the given radian.

Useful Example of math module:

Eg:

1. **import** math
2. a=4.6
3. **print** math.ceil(a)
4. **print** math.floor(a)
5. b=9
6. **print** math.sqrt(b)
7. **print** math.exp(3.0)
8. **print** math.log(2.0)
9. **print** math.pow(2.0,3.0)
10. **print** math.sin(0)
11. **print** math.cos(0)
12. **print** math.tan(45)

Output:

1. >>>
2. 5.0
3. 4.0
4. 3.0
5. 20.0855369232
6. 0.69314718056
7. 8.0
8. 0.0
9. 1.0
10. 1.61977519054
11. >>>

Constants:

The math module provides two constants for mathematical Operations:

Constants	Descriptions
Pi	Returns constant $\pi = 3.14159...$
ceil(n)	Returns constant $e = 2.71828...$

Eg:

1. **import** math
- 2.
3. **print** math.pi
4. **print** math.e

Output:

1. >>>
2. 3.14159265359
3. 2.71828182846
4. >>>

2) random:

The random module is used to generate the random numbers. It provides the following two built in functions:

Function	Description
random()	It returns a random number between 0.0 and 1.0 where 1.0 is exclusive.
randint(x,y)	It returns a random number between x and y where both the numbers are included.

Eg:

1. **import** random
- 2.
3. **print** random.random()
4. **print** random.randint(2,8)

Output:

1. >>>
2. 0.797473843839
3. 7
4. >>>

Other modules will be covered in their respective topics.

Package

A Package is simply a collection of similar modules, sub-packages etc..

Steps to create and import Package:

- 1) Create a directory, say Info

2) Place different modules inside the directory. We are placing 3 modules msg1.py, msg2.py and msg3.py respectively and place corresponding codes in respective modules. Let us place msg1() in msg1.py, msg2() in msg2.py and msg3() in msg3.py.

3) Create a file __init__.py which specifies attributes in each module.

4) Import the package and use the attributes using package.

Have a look over the example:

1) Create the directory:

1. **import** os
2. os.mkdir("Info")

2) Place different modules in package: (Save different modules inside the Info package)

msg1.py

1. **def** msg1():
2. **print** "This is msg1"

msg2.py

1. **def** msg2():
2. **print** "This is msg2"

msg3.py

1. **def** msg3():
2. **print** "This is msg3"

3) Create __init__.py file:

1. **from** msg1 **import** msg1
2. **from** msg2 **import** msg2
3. **from** msg3 **import** msg3

4) Import package and use the attributes:

1. **import** Info
2. Info.msg1()
3. Info.msg2()
4. Info.msg3()

Output:

1. >>>
2. This **is** msg1
3. This **is** msg2
4. This **is** msg3

5. >>>

What is `__init__.py` file? `__init__.py` is simply a file that is used to consider the directories on the disk as the package of the Python. It is basically used to initialize the python packages.

EXCEPTION HANDLING

Exception can be said to be any abnormal condition in a program resulting to the disruption in the flow of the program.

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

Exception in a code can also be handled. In case it is not handled, then the code is not executed further and hence execution stops when exception occurs.

Hierarchy Of Exception:

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 end of file is reached and yet operations are being performed

etc..

Exception Handling:

The suspicious code can be handled by using the try block. Enclose the code which raises an exception inside the try block. The try block is followed except statement. It is then further followed by statements which are executed during exception and in case if exception does not occur.

Syntax:

1. **try:**
2. malicious code
3. **except** Exception1:
4. execute code
5. **except** Exception2:
6. execute code
7.
8.
9. **except** ExceptionN:
10. execute code
11. **else:**
12. In case of no exception, execute the **else** block code.

eg:

1. **try:**
2. a=10/0
3. **print** a
4. **except** ArithmeticError:
5. **print** "This statement is raising an exception"

6. **else:**
7. **print** "Welcome"

Output:

1. >>>
2. This statement **is** raising an exception
3. >>>

Explanation:

1. The malicious code (code having exception) is enclosed in the try block.
2. Try block is followed by except statement. There can be multiple except statement with a single try block.
3. Except statement specifies the exception which occurred. In case that exception is occurred, the corresponding statement will be executed.
4. At the last you can provide else statement. It is executed when no exception is occurred.

Except with no Exception:

Except statement can also be used without specifying Exception.

Syntax:

1. **try:**
2. code
3. **except:**
4. code to be executed **in** case exception occurs.
5. **else:**
6. code to be executed **in** case exception does **not** occur.

eg:

1. **try:**
2. a=10/0;
3. **except:**
4. **print** "Arithmetic Exception"
5. **else:**
6. **print** "Successfully Done"

Output:

1. >>>
2. Arithmetic Exception
3. >>>

Declaring Multiple Exception

Multiple Exceptions can be declared using the same except statement:

Syntax:

1. **try:**
2. code
3. **except** Exception1,Exception2,Exception3,...,ExceptionN
4. execute this code **in** case any Exception of these occur.
5. **else:**
6. execute code **in** case no exception occurred.

eg:

1. **try:**
2. a=10/0;
3. **except** ArithmeticError,StandardError:
4. **print** "Arithmetic Exception"
5. **else:**
6. **print** "Successfully Done"

Output:

1. >>>
2. Arithmetic Exception
3. >>>

Finally Block:

In case if there is any code which the user want to be executed, whether exception occurs or not then that code can be placed inside the finally block. Finally block will always be executed irrespective of the exception.

Syntax:

1. **try:**
2. Code
3. **finally:**
4. code which **is** must to be executed.

eg:

1. **try:**
2. a=10/0;
3. **print** "Exception occurred"
4. **finally:**
5. **print** "Code to be executed"

Output:

1. >>>
2. Code to be executed
3. Traceback (most recent call last):
4. File "C:/Python27/noexception.py", line 2, in <module>
5. a=10/0;
6. ZeroDivisionError: integer division or modulo by zero
7. >>>

In the above example finally block is executed. Since exception is not handled therefore exception occurred and execution is stopped.

Raise an Exception:

You can explicitly throw an exception in Python using `raise` statement. `raise` will cause an exception to occur and thus execution control will stop in case it is not handled.

Syntax:

1. `raise` Exception_class,<value>

eg:

1. `try:`
2. `a=10`
3. `print a`
4. `raise` NameError("Hello")
5. `except` NameError as e:
6. `print "An exception occurred"`
7. `print e`

Output:

1. >>>
2. 10
3. An exception occurred
4. Hello
5. >>>

Explanation:

- i) To raise an exception, `raise` statement is used. It is followed by exception class name.
- ii) Exception can be provided with a value that can be given in the parenthesis. (here, Hello)
- iii) To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

Custom Exception:

Refer to this section after visiting Class and Object section:

Creating your own Exception class or User Defined Exceptions are known as Custom Exception.

eg:

```
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:

```
1. >>>
2. Received error : 2000
3. >>>
```

Date and Time

Python is very useful in case of Date and Time. We can easily retrieve current date and time using Python.

Retrieve Time

To retrieve current time a predefined function `localtime()` is used. `localtime()` receives a parameter `time.time()`. Here,

`time` is a module,

`time()` is a function that returns the current system time in number of ticks since 12:00 am , January 1,1970. It is known as epoch.

Tick is simply a floating point number in seconds since epoch.

eg:

1. **import** time;
2. `localtime = time.localtime(time.time())`
3. **print** "Current Time is :", localtime

Output:

1. `>>>`
2. Current Time is :time.struct_time(tm_year=2014, tm_mon=6, tm_mday=18, tm_hour=12,
3. tm_min=35, tm_sec=44, tm_wday=2, tm_yday=169, tm_isdst=0)
4. `>>>`

Explanation:

The time returned is a time structure which includes 9 attributes. These are summoned in the table given below.

Attribute	Description
tm_year	Returns the current year
tm_mon	Returns the current month
tm_mday	Returns the current month day
tm_hour	Returns the current hour.
tm_min	Returns the current minute

tm_sec	Returns current seconds
tm_wday	Returns the week day
tm_yday	Returns the year day.
tm_isdst	It returns -1,0 or 1.

Formatted Time

Python also support formatted time. Proceed as follows:

1. Pass the time structure in a predefined function `asctime()`. It is a function defined in `time` module.
2. It returns a formatted time which includes Day ,month, date, time and year.
3. Print the formatted time.

eg:

1. **import** time;
- 2.
3. `localtime = time.asctime(time.localtime(time.time()))`
4. **print** "Formatted time :", localtime

Output:

1. `>>>`
2. Formatted time : Sun Jun 22 18:54:20 2014
3. `>>>`

time module:

There are many built in functions defined in `time` module which are used to work with time.

Methods	Description
<code>time()</code>	Returns floating point value in seconds since epoch i.e 1970
<code>asctime(time)</code>	It takes the tuple returned by <code>localtime()</code> as parameter and returns a character string.
<code>sleep(time)</code>	The execution will be stopped for the given interval of time.

<code>strptime(String,format)</code>	It returns an tuple with 9 time attributes. It receives a string and a format.
<code>gtime()/gtime(sec)</code>	It returns struct_time which contains 9 time attributes. If no argument is specified it takes current second from epoch.
<code>mktime()</code>	Returns second in floating point since epoch.
<code>strftime(format)/strftime(format,time)</code>	Returns time in particular format. If time is not given, current time is fetched.

time()

eg:

1. `import time`
2. `printtime.time()`

Output:

1. `>>>`
2. `1403700740.39`
3. `>>>`

asctime(time)

1. `import time`
2. `t = time.localtime()`
3. `printtime.asctime(t)`

Output:

1. `>>>`
2. `Wed Jun 25 18:30:25 2014`
3. `>>>`

sleep(time)

Eg:

1. `import time`
2.
3. `localtime = time.asctime(time.localtime(time.time()))`
4. `printlocaltime`
5. `time.sleep(10)`
6. `localtime = time.asctime(time.localtime(time.time()))`
7. `printlocaltime`

Output:

1. >>>
2. Wed Jun 25 18:15:30 2014
3. Wed Jun 25 18:15:40 2014
4. >>>

strftime(String str,format f)

Eg:

1. **import** time
- 2.
3. timerequired = time.strftime("26 Jun 14", "%d %b %y")
4. printtimerequired

Output:

1. >>>
2. time.struct_time(tm_year=2014, tm_mon=6, tm_mday=26, tm_hour=0, tm_min=0,
3. tm_sec=0, tm_wday=3, tm_yday=177, tm_isdst=-1)
4. >>>

Explanation:

The strftime() takes a String and format as argument. The format refers to String passed as an argument. "%a %b %d %H:%M:%S %Y" are the default directives. There are many other directives which can be used. In the given example we have used three directives: %d%b%y which specifies day of the month, month in abbreviated form and year without century respectively. Some of them are given as:

%a	weekday name.
%b	month name
%c	date and time
%e	day of a month
%m	month in digit.
%n	new line character.
%S	second
%t	tab character

etc...

gmtime()

Eg:

1. **import** time
2. printtime.gmtime()

Output:

1. >>>
2. time.struct_time(tm_year=2014, tm_mon=6, tm_mday=28, tm_hour=9, tm_min=38, tm_sec=0,
3. tm_wday=5, tm_yday=179, tm_isdst=0)
4. >>>

mktime()

Eg:

1. **import** time
2. t = (2014, 2, 17, 17, 3, 38, 1, 48, 0)
3. second = time.mktime(t)
4. **print** second

Output:

1. >>>
2. 1392636818.0
3. >>>

strftime()

Eg:

1. **import** time
2. t = (2014, 6, 26, 17, 3, 38, 1, 48, 0)
3. t = time.mktime(t)
4. printtime.strftime("%b %d %Y %H:%M:%S", time.gmtime(t))

Output:

1. >>>
2. Jun 26 2014 11:33:38
3. >>>

Calendar

Python provides calendar module to display Calendar.

Eg:

1. **import** calendar
2. **print** "Current month is:"
3. cal = calendar.month(2014, 6)
4. printcal

Output:

1. >>>
2. Current month **is**:
3. June 2014
4. Mo Tu We Th Fr Sa Su
5. 1
6. 2 3 4 5 6 7 8
7. 9 10 11 12 13 14 15
8. 16 17 18 19 20 21 22
9. 23 24 25 26 27 28 29
10. 30
11. >>>

Calendar module:

Python provides calendar module which provides many functions and methods to work on calendar. A list of methods and function used is given below:

Methods	Description
prcal(year)	Prints the whole calendar of the year.
firstweekday()	Returns the first week day. It is by default 0 which specifies Monday.
isleap(year)	Returns a Boolean value i.e., true or false. True in case given year is a leap year.
monthcalendar(year, month)	Returns the given month with each week as one list.
leapdays(year1, year2)	Return number of leap days from year1 to year2
prmonth(year, month)	Print the given month of the given year

prcal(year)

Eg:

1. **import** calendar
2. calendar.prcal(2014)

Output:

```

1. >>> ===== RESTART =====
=====

2. >>>
    2014

    January      February      March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
1 2 3 4 5          1 2          1 2
6 7 8 9 10 11 12    3 4 5 6 7 8 9    3 4 5 6 7 8 9
13 14 15 16 17 18 19 10 11 12 13 14 15 16 10 11 12 13 14 15 16
20 21 22 23 24 25 26 17 18 19 20 21 22 23 17 18 19 20 21 22 23
27 28 29 30 31      24 25 26 27 28      24 25 26 27 28 29 30
                        31

    April      May      June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
1 2 3 4 5 6          1 2 3 4          1
7 8 9 10 11 12 13    5 6 7 8 9 10 11    2 3 4 5 6 7 8
14 15 16 17 18 19 20 12 13 14 15 16 17 18 9 10 11 12 13 14 15
21 22 23 24 25 26 27 19 20 21 22 23 24 25 16 17 18 19 20 21 22
28 29 30          26 27 28 29 30 31    23 24 25 26 27 28 29
                        30

    July      August      September
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
1 2 3 4 5 6          1 2 3    1 2 3 4 5 6 7
7 8 9 10 11 12 13    4 5 6 7 8 9 10    8 9 10 11 12 13 14
14 15 16 17 18 19 20 11 12 13 14 15 16 17 15 16 17 18 19 20 21
21 22 23 24 25 26 27 18 19 20 21 22 23 24 22 23 24 25 26 27 28
28 29 30 31          25 26 27 28 29 30 31 29 30

    October      November      December
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
1 2 3 4 5          1 2    1 2 3 4 5 6 7
6 7 8 9 10 11 12    3 4 5 6 7 8 9    8 9 10 11 12 13 14
13 14 15 16 17 18 19 10 11 12 13 14 15 16 15 16 17 18 19 20 21
20 21 22 23 24 25 26 17 18 19 20 21 22 23 22 23 24 25 26 27 28
27 28 29 30 31      24 25 26 27 28 29 30 29 30 31

>>>

```

javatpoint.com

firstweekday()

Eg:

1. `import calendar`
2. `printcalendar.firstweekday()`

Output:

1. `>>>`
2. `0`
3. `>>>`

isleap(year)

Eg:

1. **import** calendar
2. printcalendar.isleap(2000)

Output:

1. >>>
2. True
3. >>>

monthcalendar(year,month)

Eg:

1. **import** calendar
2. printcalendar.monthcalendar(2014,6)

Output:

1. >>>
2. [[0, 0, 0, 0, 0, 0, 1], [2, 3, 4, 5, 6, 7, 8], [9, 10, 11, 12, 13, 14, 15],
3. [16, 17, 18, 19, 20, 21, 22],
4. [23, 24, 25, 26, 27, 28, 29], [30, 0, 0, 0, 0, 0, 0]]
5. >>>

prmonth(year,month)

Eg:

1. **import** calendar
2. printcalendar.prmonth(2014,6)

Output:

1. >>>
2. June 2014
3. Mo Tu We ThFrSa Su
4. 1
5. 2 3 4 5 6 7 8
6. 9 10 11 12 13 14 15
7. 16 17 18 19 20 21 22
8. 23 24 25 26 27 28 29
9. 30
10. None
11. >>>

Python Programs

There can be various python programs on many topics like basic python programming, conditions and loops, functions and native data types. A list of top python programs are given below which are widely asked by interviewer.

Basic Python programs

- [Python program to print "Hello Python"](#)
- [Python program to do arithmetical operations](#)
- [Python program to find the area of a triangle](#)
- [Python program to solve quadratic equation](#)
- [Python program to swap two variables](#)
- [Python program to generate a random number](#)
- [Python program to convert kilometers to miles](#)
- [Python program to convert Celsius to Fahrenheit](#)
- [Python program to display calendar](#)

Python programs with conditions and loops

- [Python Program to Check if a Number is Positive, Negative or Zero](#)
- [Python Program to Check if a Number is Odd or Even](#)
- [Python Program to Check Leap Year](#)
- [Python Program to Check Prime Number](#)
- [Python Program to Print all Prime Numbers in an Interval](#)
- [Python Program to Find the Factorial of a Number](#)
- [Python Program to Display the multiplication Table](#)
- [Python Program to Print the Fibonacci sequence](#)
- [Python Program to Check Armstrong Number](#)
- [Python Program to Find Armstrong Number in an Interval](#)
- [Python Program to Find the Sum of Natural Numbers](#)

Python Function Programs

- [Python Program to Find LCM](#)
- [Python Program to Find HCF](#)
- [Python Program to Convert Decimal to Binary, Octal and Hexadecimal](#)
- [Python Program To Find ASCII value of a character](#)
- [Python Program to Make a Simple Calculator](#)
- [Python Program to Display Calendar](#)
- [Python Program to Display Fibonacci Sequence Using Recursion](#)
- [Python Program to Find Factorial of Number Using Recursion](#)

Python Native Data Type Programs

- [Python Program to Add Two Matrices](#)
- [Python Program to Multiply Two Matrices](#)
- [Python Program to Transpose a Matrix](#)
- [Python Program to Sort Words in Alphabetic Order](#)
- [Python Program to Remove Punctuation From a String](#)

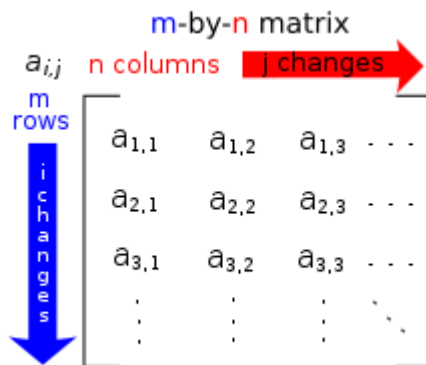
Python Program to Add Two Matrices

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 3x2 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. $[16,17,18]]$

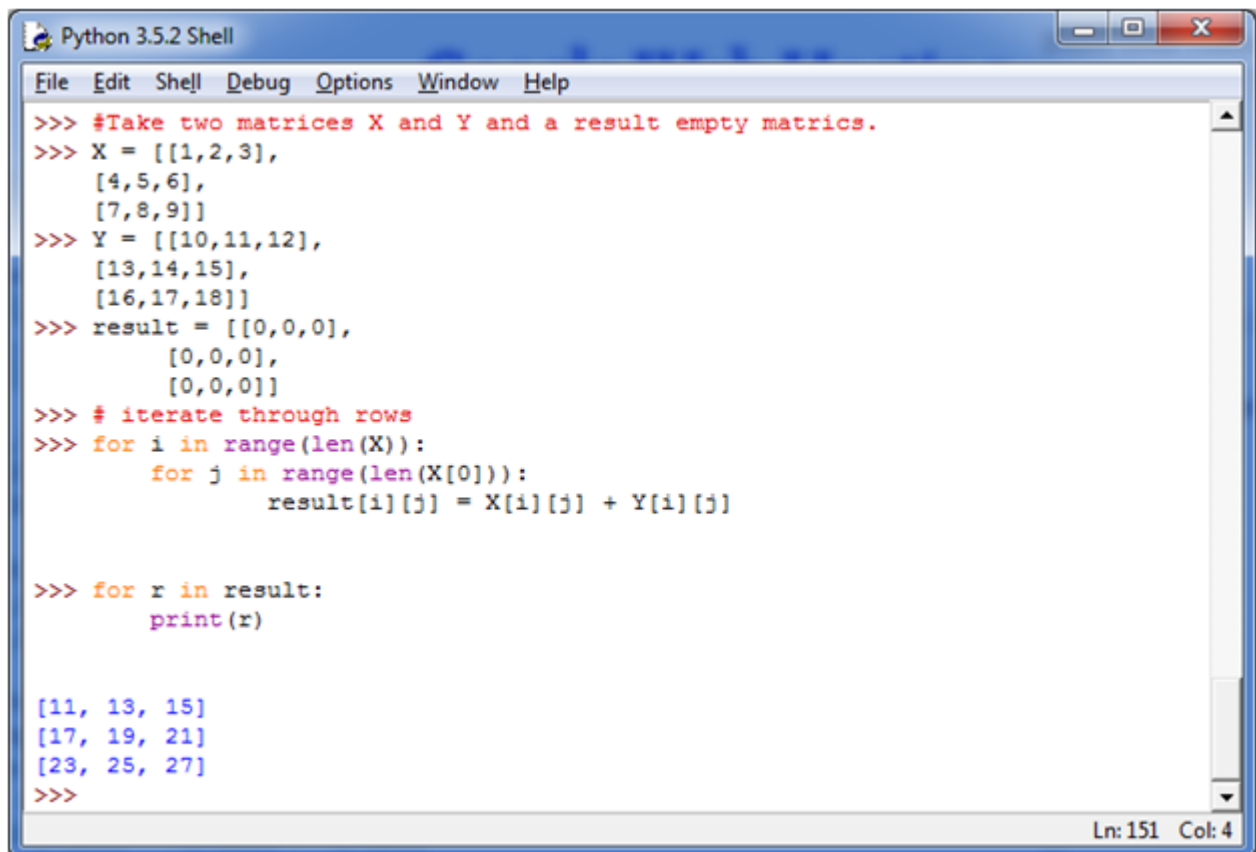
Create a new matrix result by adding them.

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 a Python 3.5.2 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The code entered 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)
```

The output of the program is:

```
[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.

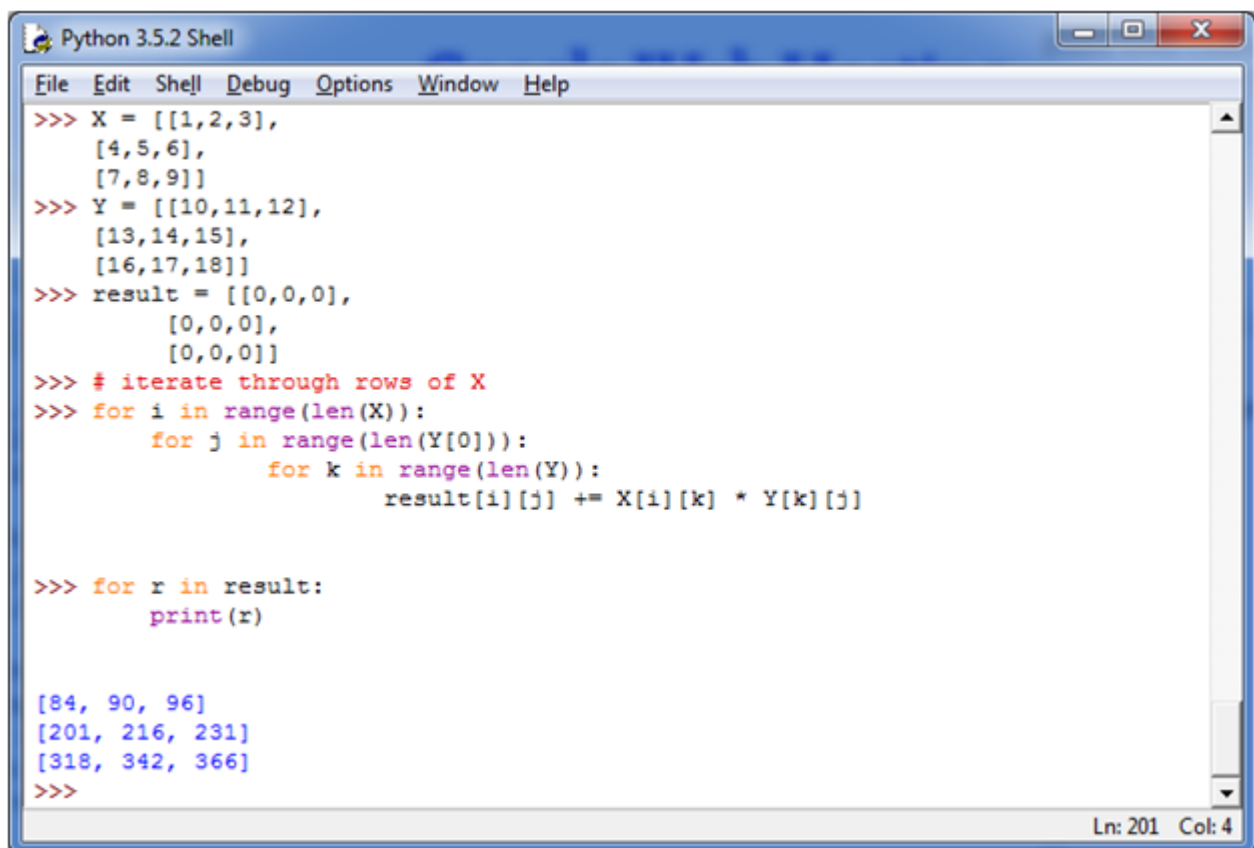
Matrix multiplication:

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:



A screenshot of a Python 3.5.2 Shell window. The window has a title bar with the text 'Python 3.5.2 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area contains a Python script that defines two 3x3 matrices, X and Y, and calculates their product. The script uses nested loops to iterate through the rows of X and the columns of Y. The output of the script is displayed at the bottom of the window.

```
>>> 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]
>>>
```

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^{th} row and j^{th} column in X will be placed at j^{th} row and i^{th} column in X' .

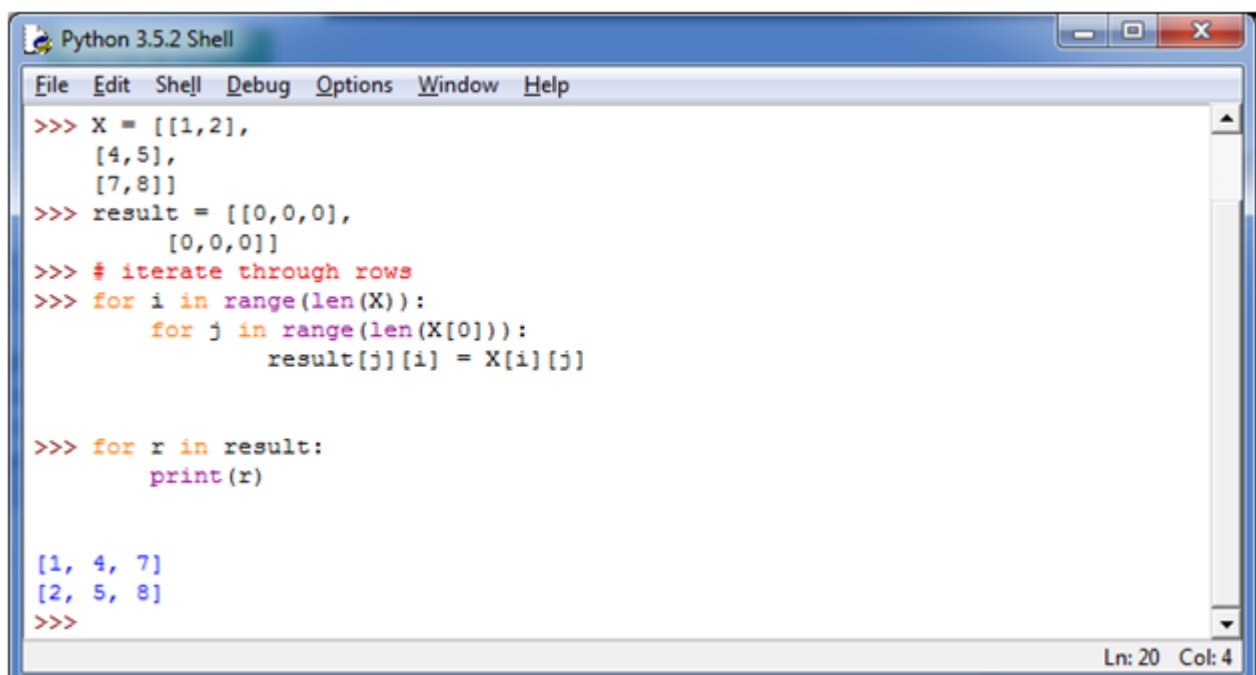
Let's take a matrix X , having the following elements:

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. $\text{Result} = [[0,0,0],$
6. $[0,0,0]]$
- 7.
8. *# iterate through rows*
9. **for** i **in** $\text{range}(\text{len}(X))$:
10. **for** j **in** $\text{range}(\text{len}(X[0]))$:
11. $\text{result}[j][i] = X[i][j]$
- 12.
13. **for** r **in** result :
14. **print**(r)

Output:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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]
>>>
```

Ln: 20 Col: 4

Python Program to Sort Words in Alphabetic Order

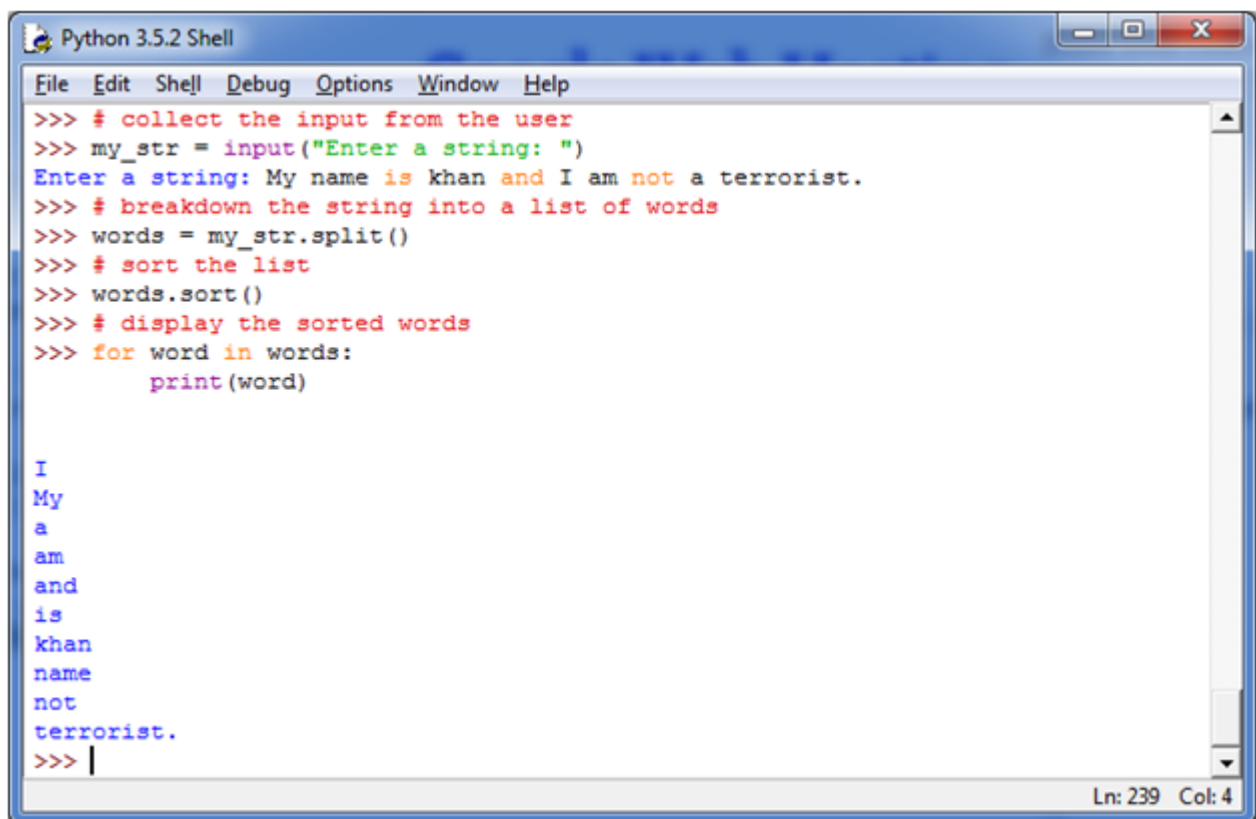
Sorting:

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> # 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)

I
My
a
am
and
is
khan
name
not
terrorist.
>>> |
```

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:

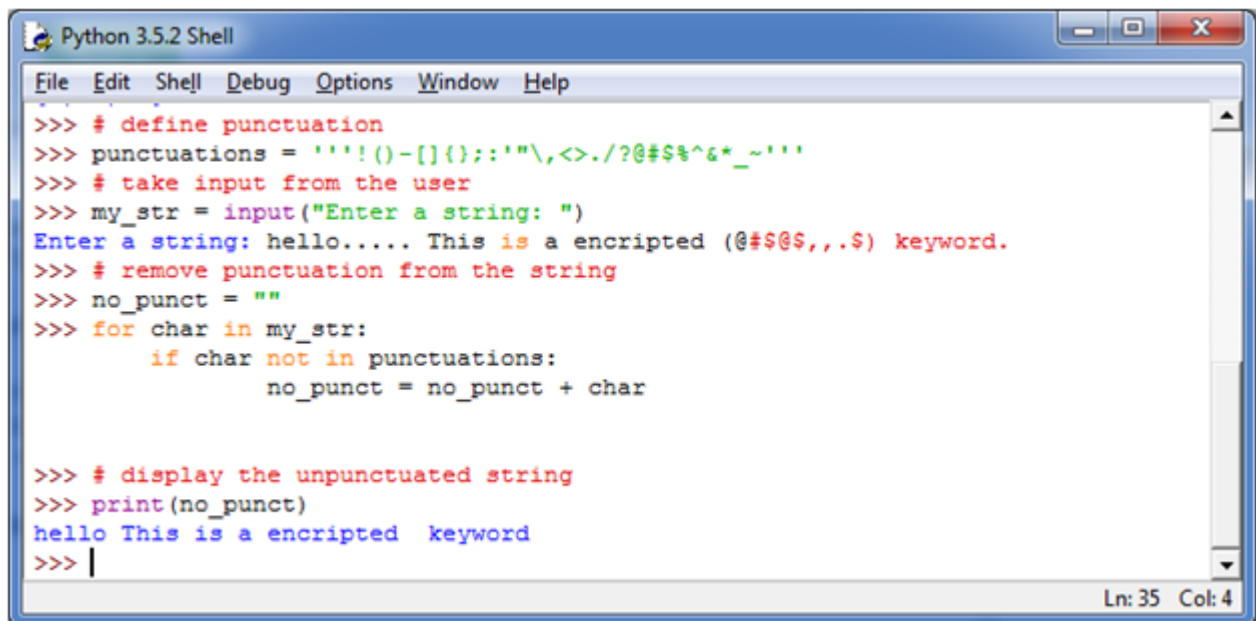
- "Woman, without her man, is nothing" (the sentence boasting about men's importance.)
- "Woman: without her, man is nothing" (the sentence boasting about women's importance.)

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:

A screenshot of a Python 3.5.2 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area contains the following Python code:

```
>>> # define punctuation
>>> punctuations = '!"()-[]{};:'"\<>./?@$%^&*~'
>>> # take input from the user
>>> my_str = input("Enter a string: ")
Enter a string: hello..... This is a encrypted (@#$%$,,$) 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 encrypted keyword
>>> |
```

The status bar at the bottom right shows 'Ln: 35 Col: 4'.

Python Program to Find Armstrong Number between an Interval

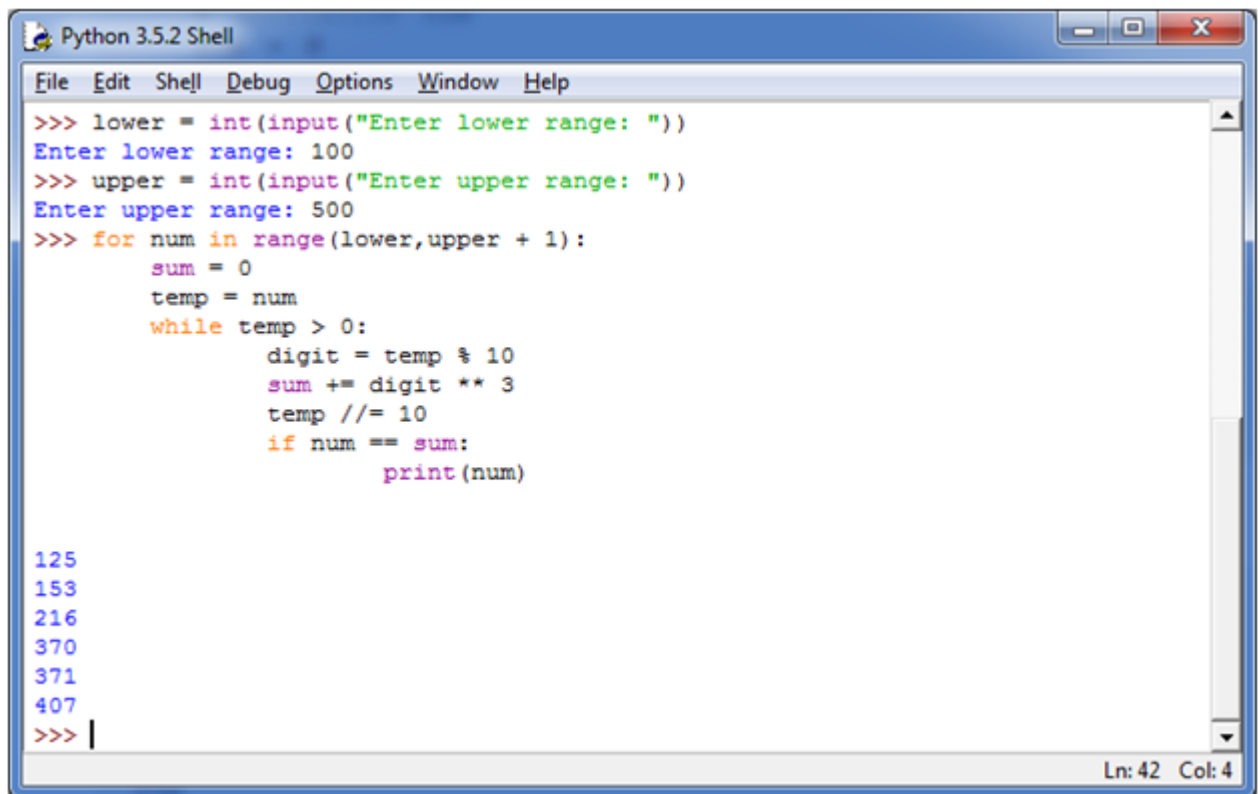
We have already read the concept of Armstrong numbers in the previous program. Here, we print the Armstrong numbers within a specific given interval.

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:



A screenshot of a Python 3.5.2 Shell window. The window has a blue title bar with the text 'Python 3.5.2 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area is a text editor with a white background and a vertical scrollbar on the right. It contains Python code for finding Armstrong numbers. The code prompts the user for a lower and upper range, then iterates through numbers in that range, checking if they are Armstrong numbers by summing the cubes of their digits. The output shows the numbers 125, 153, 216, 370, 371, and 407. The status bar at the bottom right indicates 'Ln: 42 Col: 4'.

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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
>>> |
```

Ln: 42 Col: 4

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?

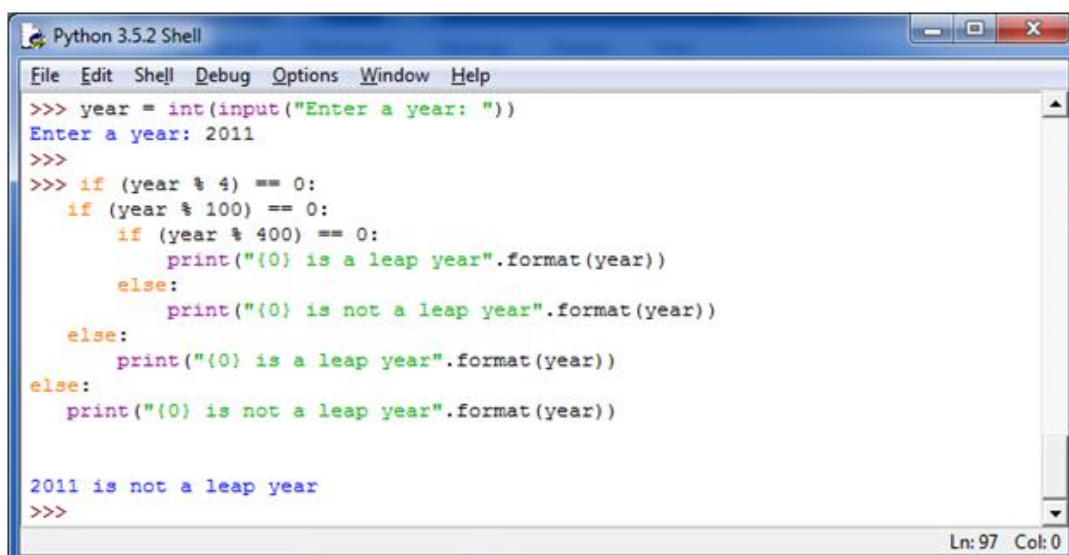
You should follow the following steps to determine whether a year is a leap year or not.

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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
>>>
```

Python Program to Print all Prime Numbers between an Interval

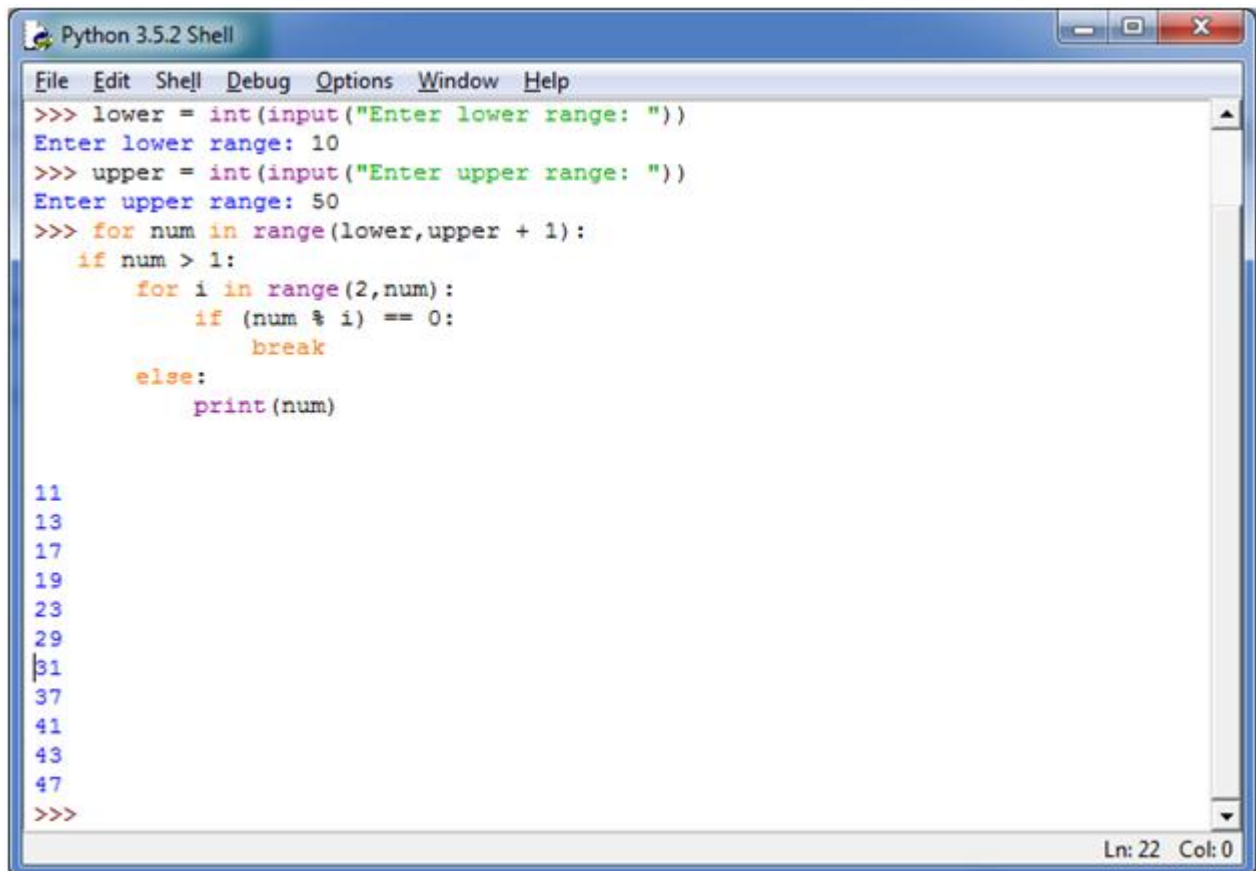
We have already read the concept of prime numbers in the previous program. Here, we are going to print the prime numbers between given interval.

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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
>>>
```

Ln: 22 Col: 0

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.

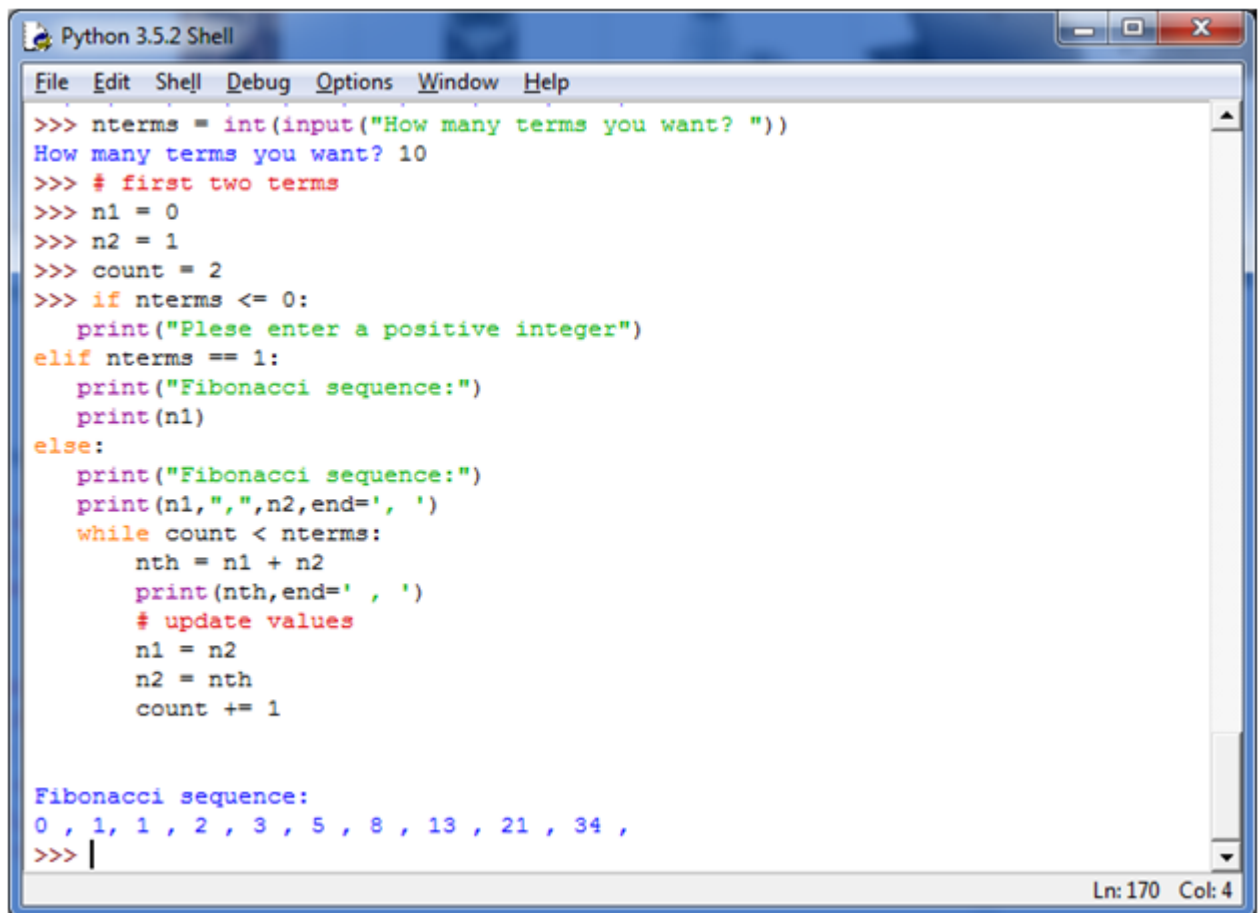
For example:

- 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("Plese 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:



A screenshot of a Python 3.5.2 Shell window. The window has a title bar with the text "Python 3.5.2 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Window, and Help. The main area of the window contains a Python script for generating a Fibonacci sequence. The script starts by asking the user for the number of terms, which is 10. It then initializes variables for the first two terms (n1=0, n2=1) and a counter (count=2). A conditional statement checks if the number of terms is less than or equal to 0, or if it is 1. If so, it prints a message or the first term. Otherwise, it prints the Fibonacci sequence. The sequence is generated using a while loop that calculates the next term (nth = n1 + n2), prints it, and updates the values of n1 and n2. The output shows the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. The status bar at the bottom right indicates "Ln: 170 Col: 4".

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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("Plese 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 ,
>>> |
```

Ln: 170 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 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....

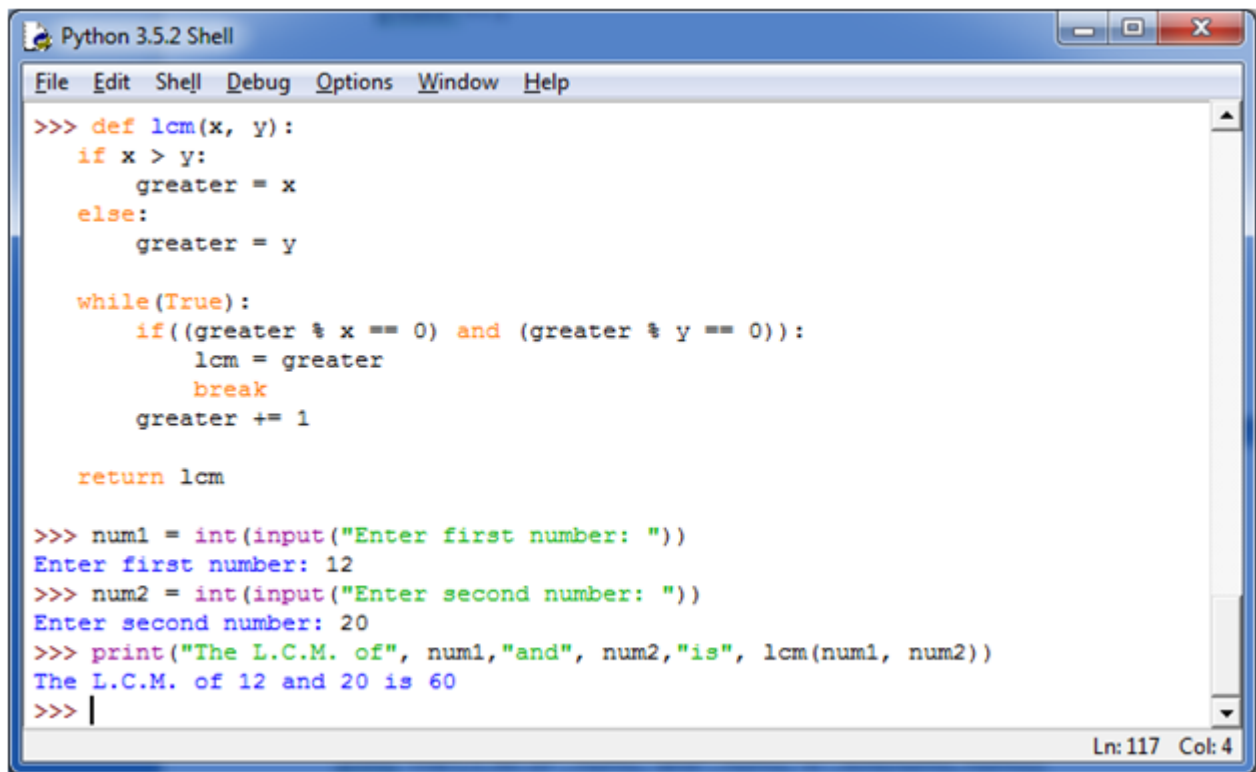
LCM is the lowest common multiplier so it is 12.

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:



A screenshot of a Python 3.5.2 Shell window. The window has a blue title bar with the text 'Python 3.5.2 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area is a text editor with a white background and a vertical scrollbar on the right. It contains Python code for calculating the Least Common Multiple (LCM) of two numbers. The code defines a function 'lcm(x, y)' that finds the greater of the two numbers and then increments it until it is divisible by both. The code is executed in a REPL style, showing user input and program output. The output shows the LCM of 12 and 20 is 60. The status bar at the bottom right indicates 'Ln: 117 Col: 4'.

```
>>> 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
>>> |
```

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

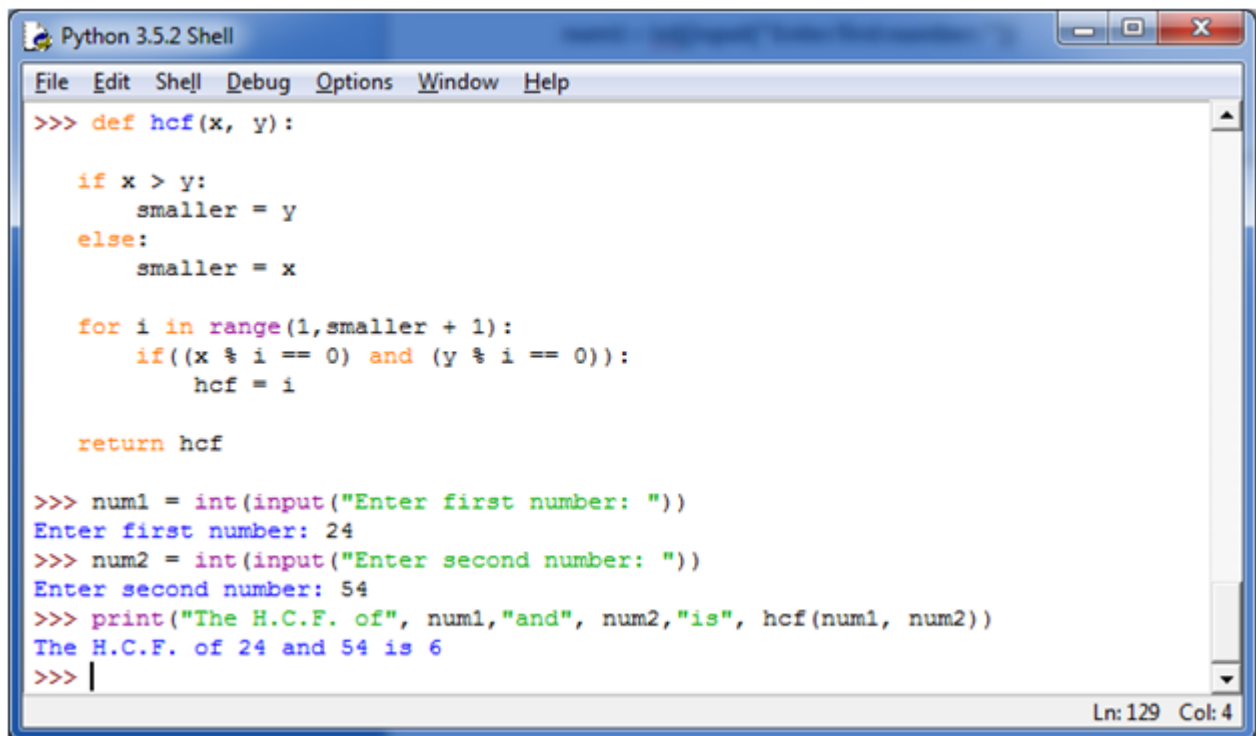
HCF /GCD is the greatest common divisor. So HCF of 8 and 12 are 4.

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:



A screenshot of a Python 3.5.2 Shell window. The window has a title bar with the text 'Python 3.5.2 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area contains Python code for calculating the Highest Common Factor (HCF) of two numbers. The code defines a function 'hcf(x, y)' that finds the smaller of the two numbers and then iterates from 1 to that number to find the largest common divisor. It then takes user input for two numbers, 24 and 54, and prints the result: 'The H.C.F. of 24 and 54 is 6'. The status bar at the bottom right shows 'Ln: 129 Col: 4'.

```
>>> 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  
>>> |
```

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.

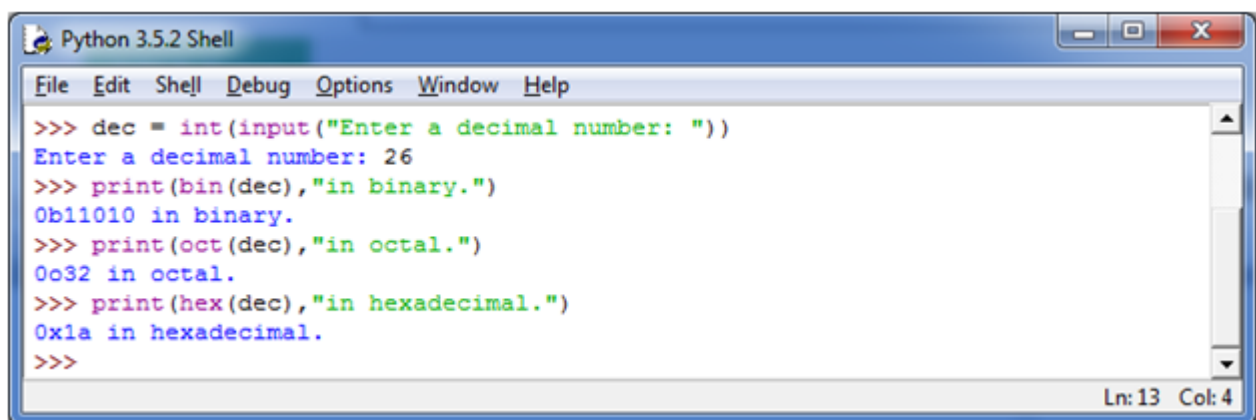
Hexadecimal System: Hexadecimal system is base 16 number system.

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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.
>>>
```

Ln:13 Col:4

Python Program To Find ASCII value of a character

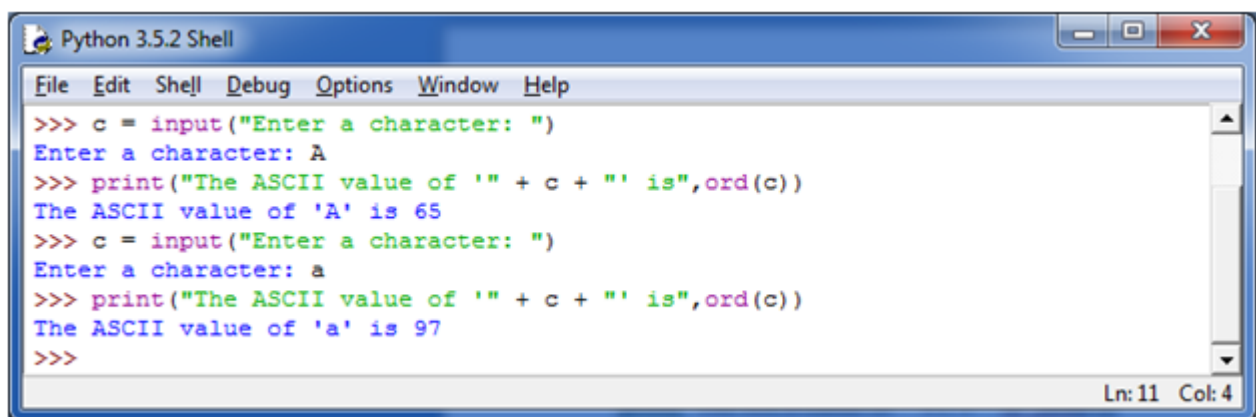
ASCII: ASCII is an acronym stands for **American Standard Code for Information Interchange**. In ASCII, a specific numerical value is given to different characters and symbols, for computers to store and manipulate.

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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
>>>
```

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

Ln: 52 Col: 0
```

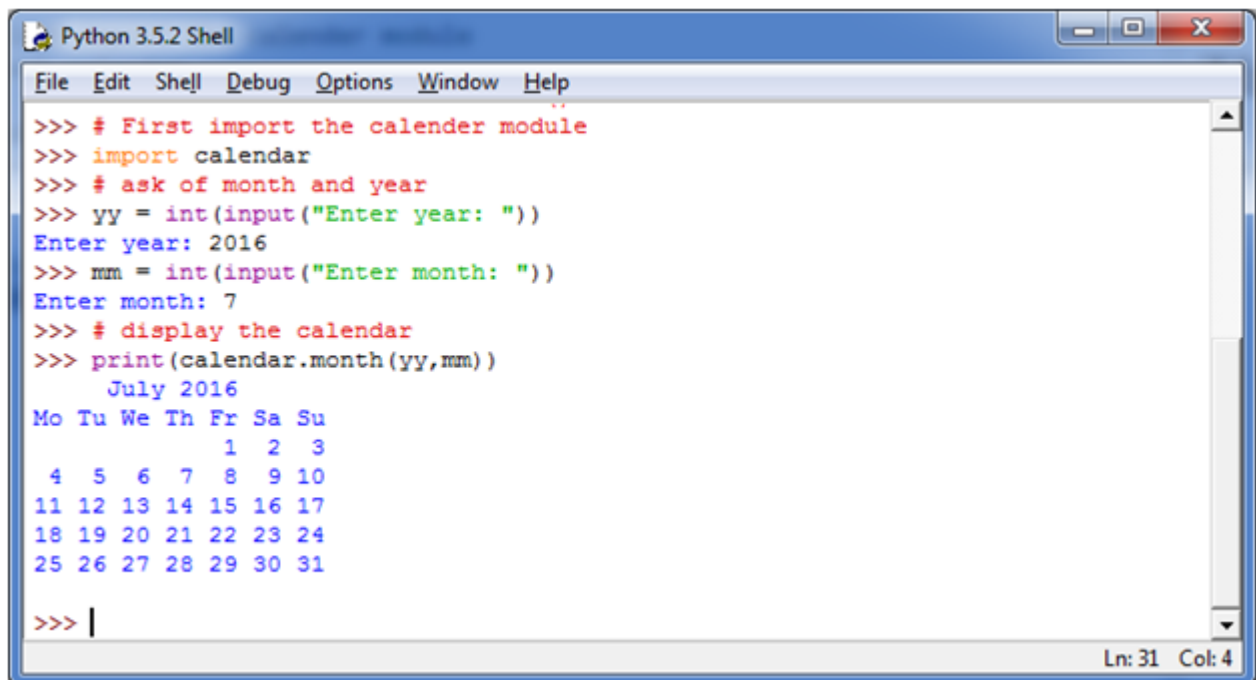
Python Function to Display Calendar

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:



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> # First import the calendar 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

>>> |
```

Ln: 31 Col: 4

Python Program to Display Fibonacci Sequence Using Recursion

Fibonacci sequence:

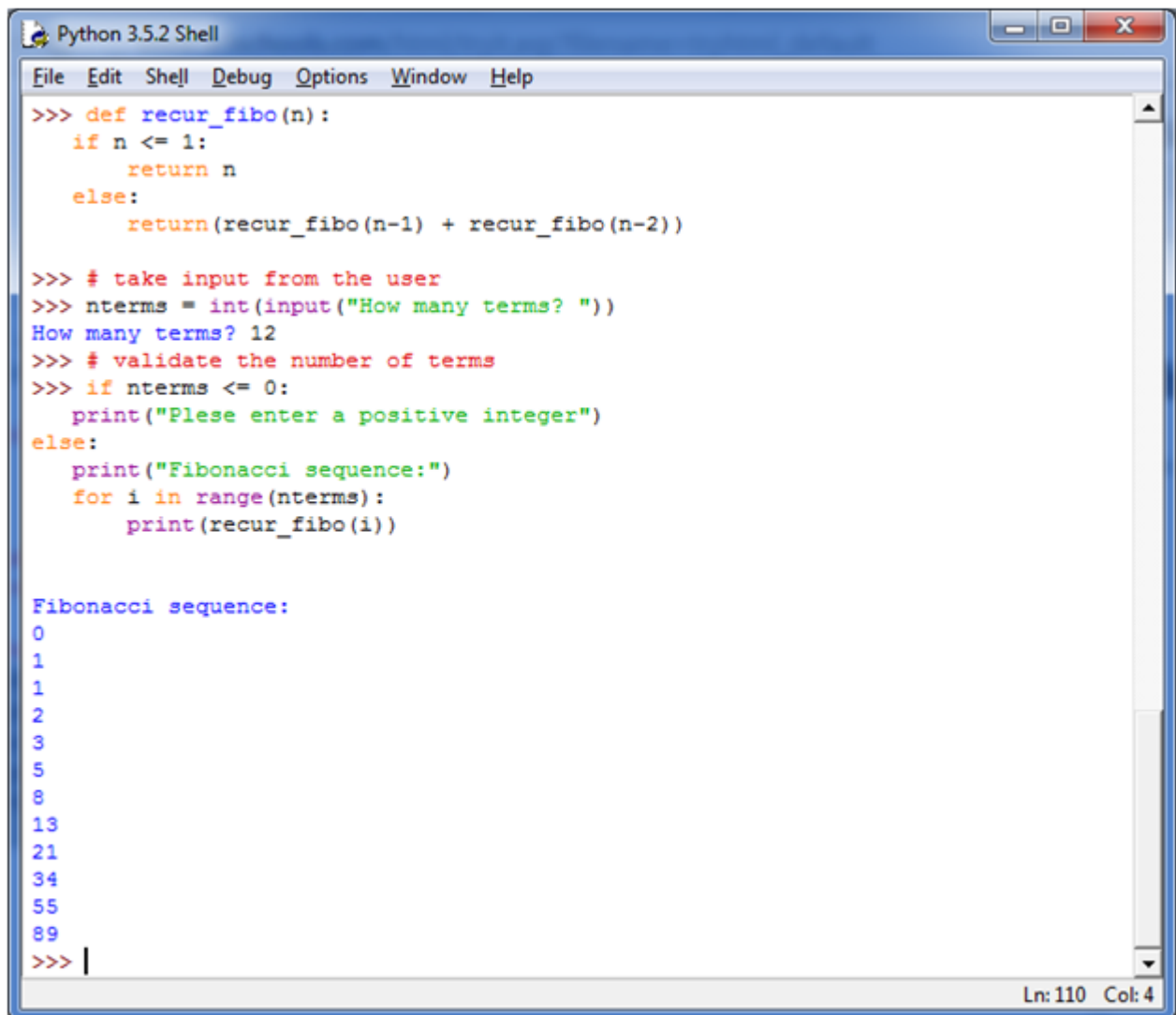
A Fibonacci sequence is a sequence of integers which first two terms are 0 and 1 and all other terms of the sequence are obtained by adding their preceding two numbers.

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("Plese enter a positive integer")
11. else:
12.    print("Fibonacci sequence:")
13.    for i in range(nterms):
14.        print(recur_fibo(i))
```

Output:



The image shows a screenshot of a Python 3.5.2 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area contains Python code for a recursive Fibonacci function and its execution. The code defines a function `recur_fibo(n)` that returns the nth Fibonacci number. It then takes user input for the number of terms (12) and prints the Fibonacci sequence. The output shows the first 12 terms of the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. The status bar at the bottom right indicates 'Ln: 110 Col: 4'.

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 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("Plese 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
>>> |
```

Ln: 110 Col: 4

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