Python

Python History and Versions

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

Python Version List

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

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

Python Version	Released Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000

Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018

Python 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 offical 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 prvides 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 Applications

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

Here, we are specifing 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 delelop 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 Bussiness 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.

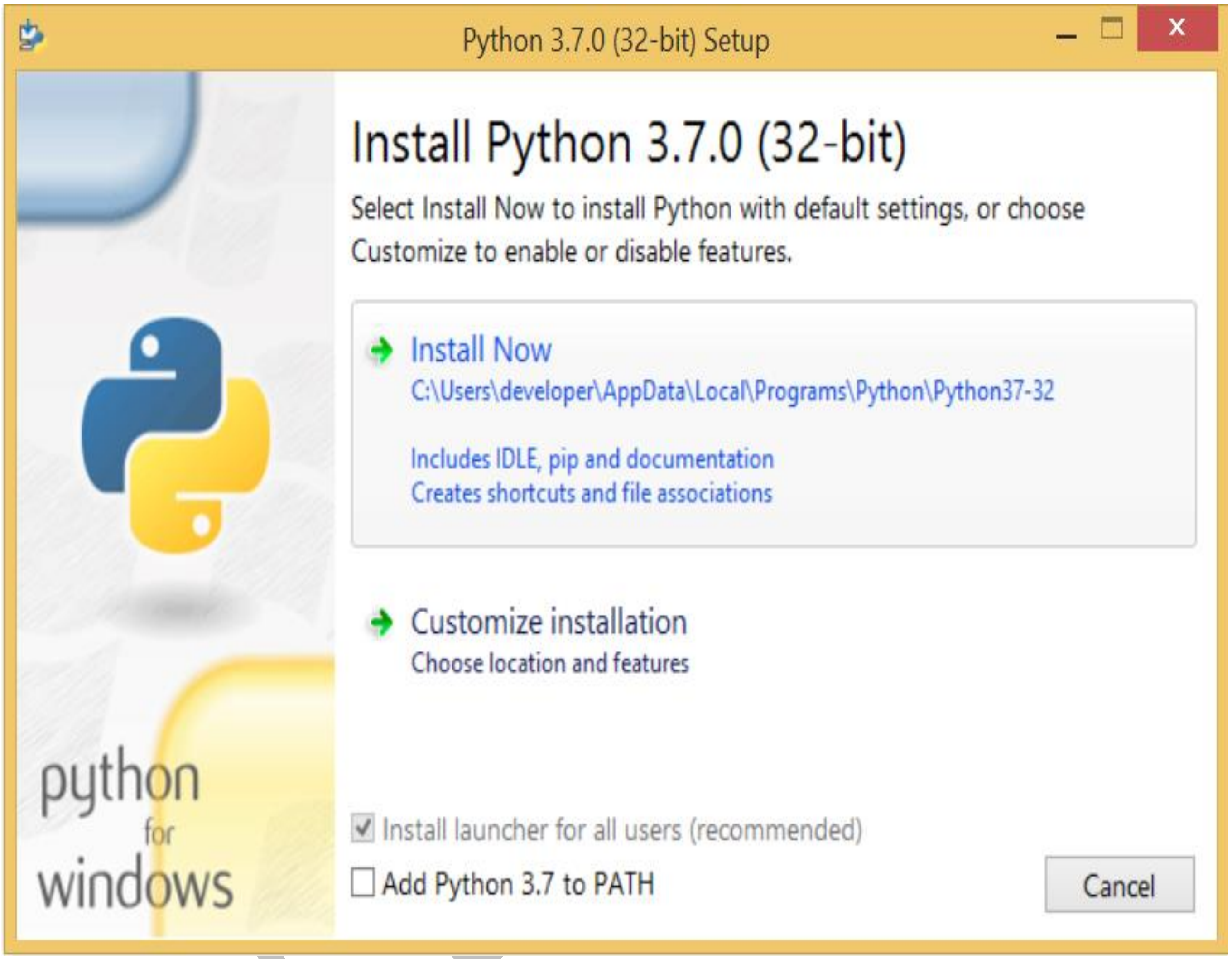
There are several such applications which can be developed using Python

How to Install Python (Environment Set-up)

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

Installation on Windows

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


Python 3.7.0 (32-bit) Setup

Install Python 3.7.0 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

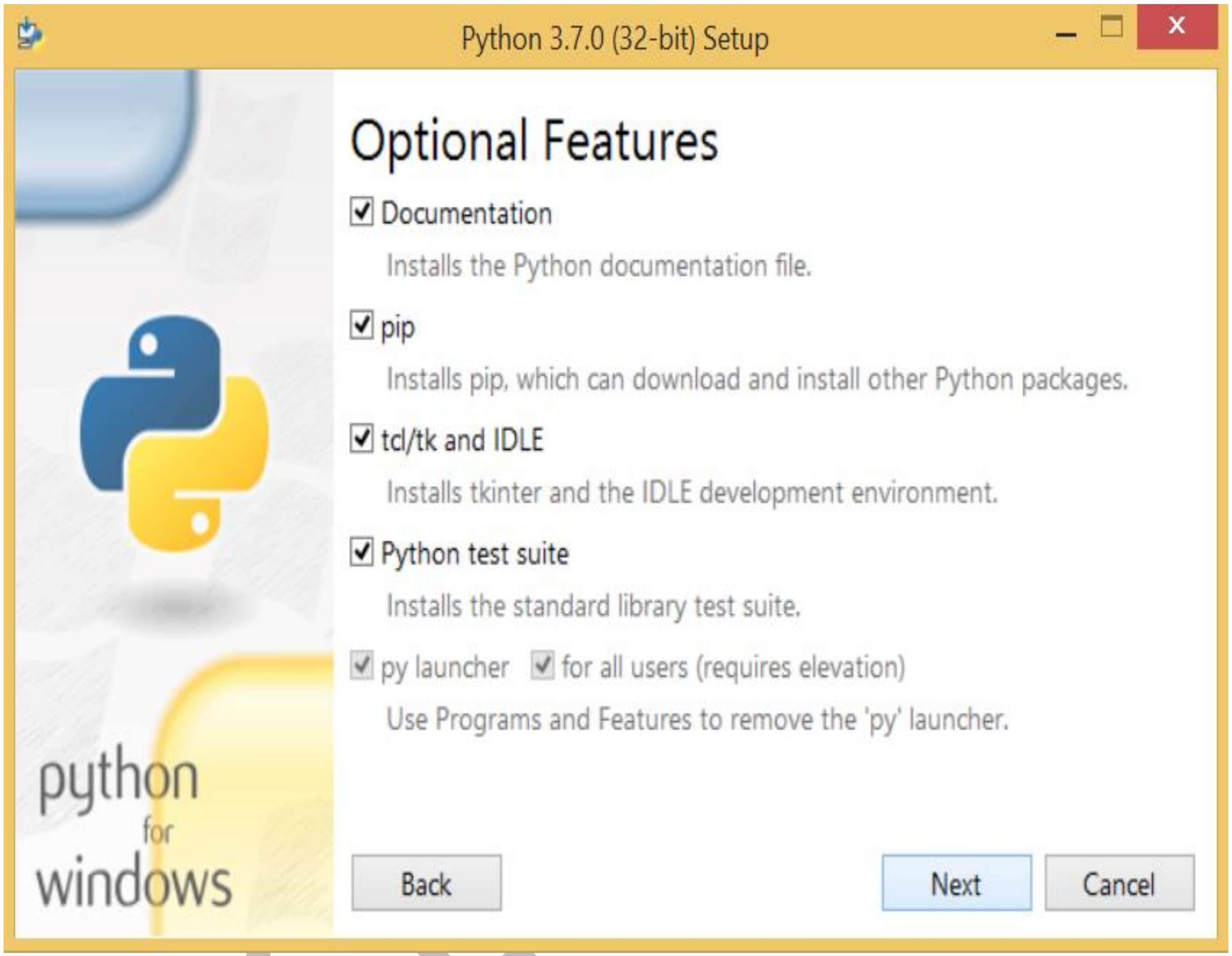
Install Now
C:\Users\developer\AppData\Local\Programs\Python\Python37-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.7 to PATH

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


Python 3.7.0 (32-bit) Setup

Optional Features

☑ Documentation
Installs the Python documentation file.

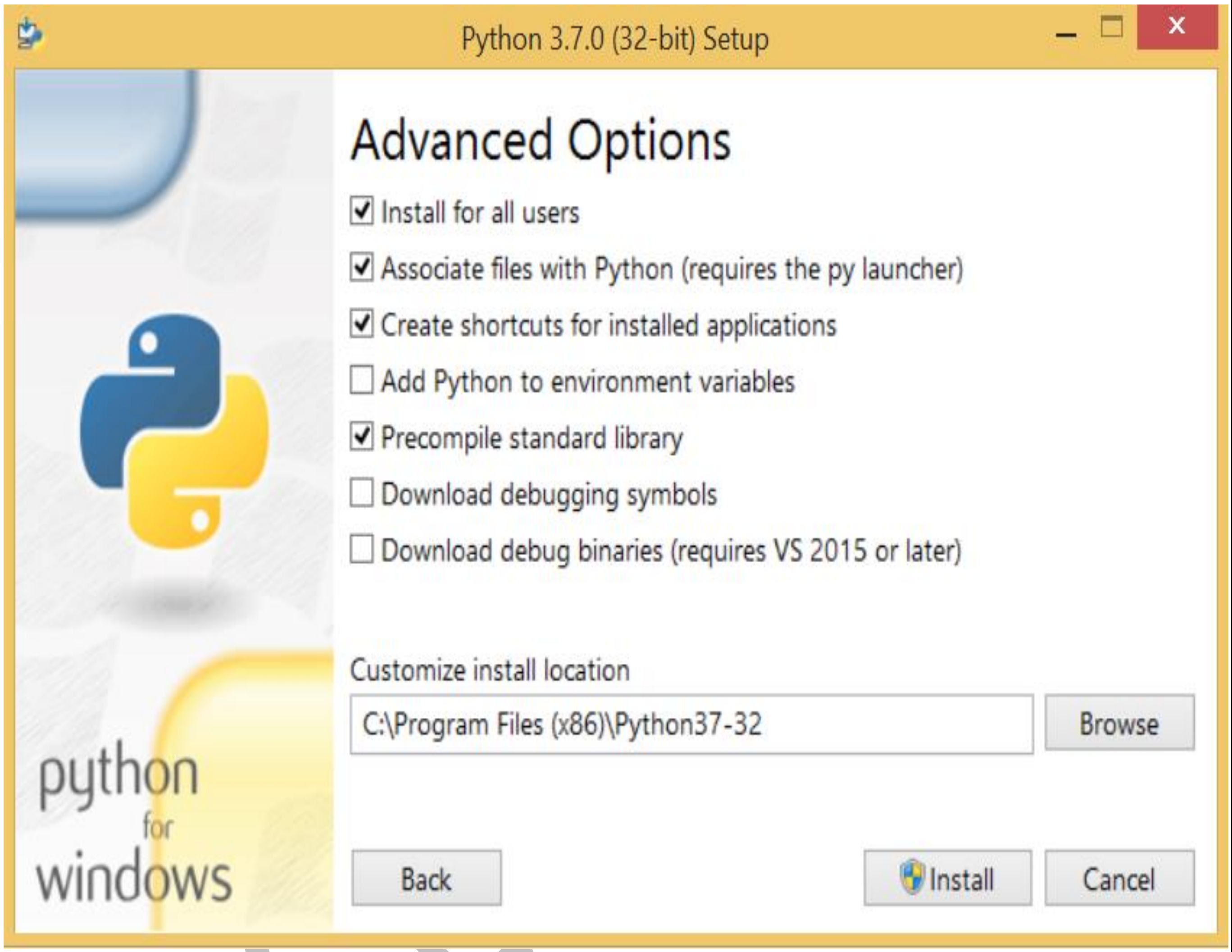
☑ pip
Installs pip, which can download and install other Python packages.

☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

☑ py launcher ☑ for all users (requires elevation)
Use Programs and Features to remove the 'py' launcher.

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



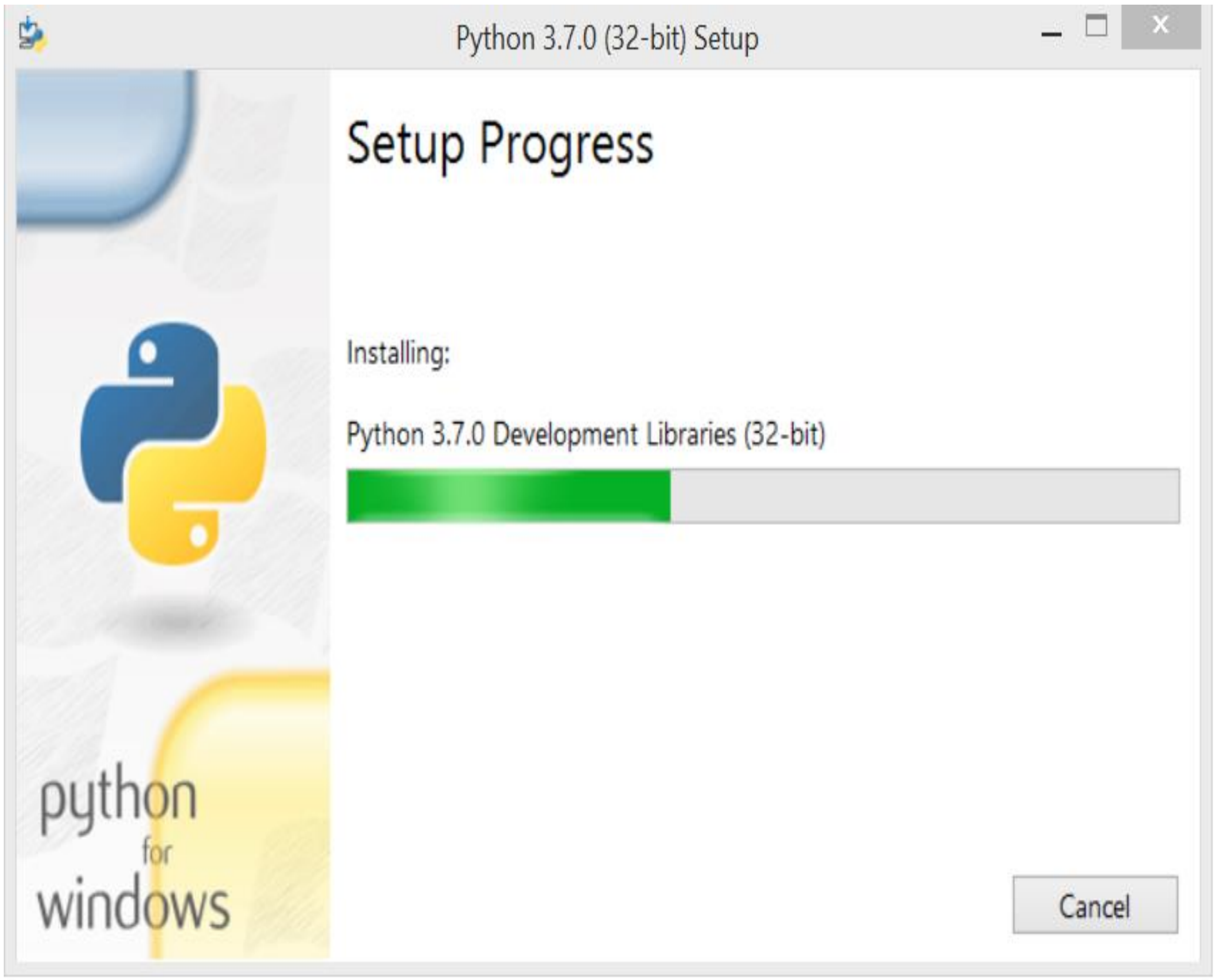
Advanced Options

- [✓] Install for all users
- [✓] Associate files with Python (requires the py launcher)
- [✓] Create shortcuts for installed applications
- [ ] Add Python to environment variables
- [✓] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2015 or later)

Customize install location

C:\Program Files (x86)\Python37-32

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



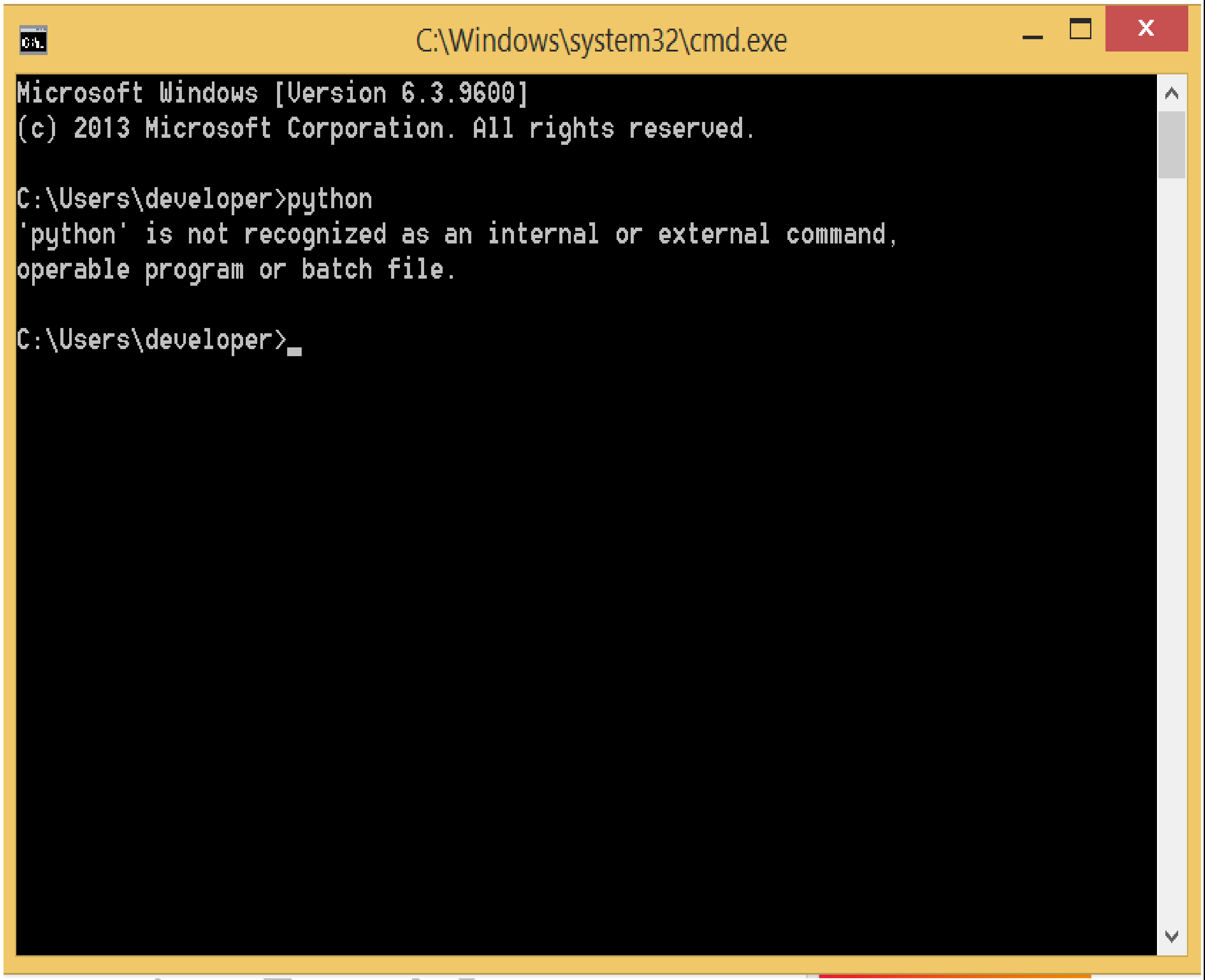
Python 3.7.0 (32-bit) Setup

Setup Progress

Installing:

Python 3.7.0 Development Libraries (32-bit)

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

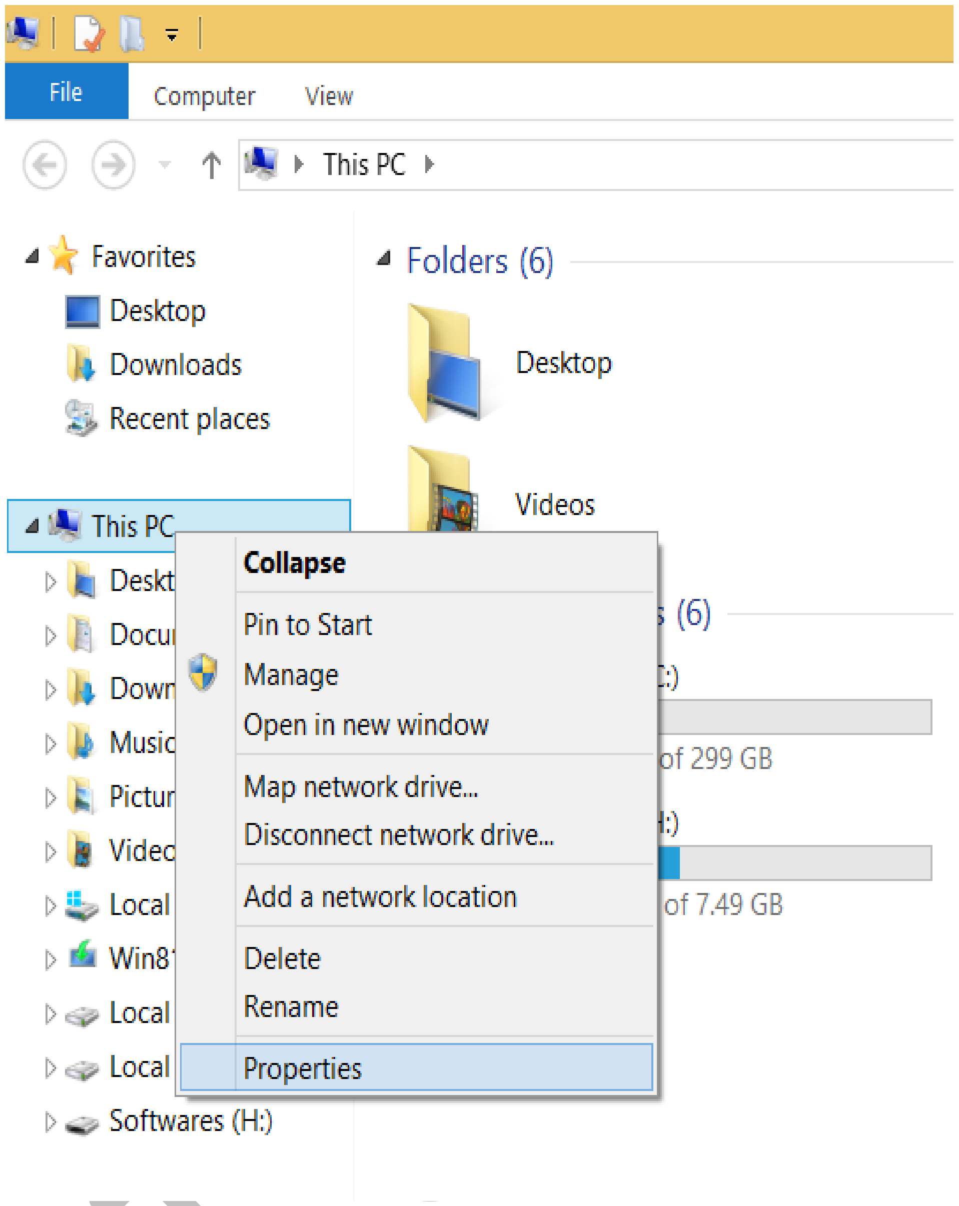


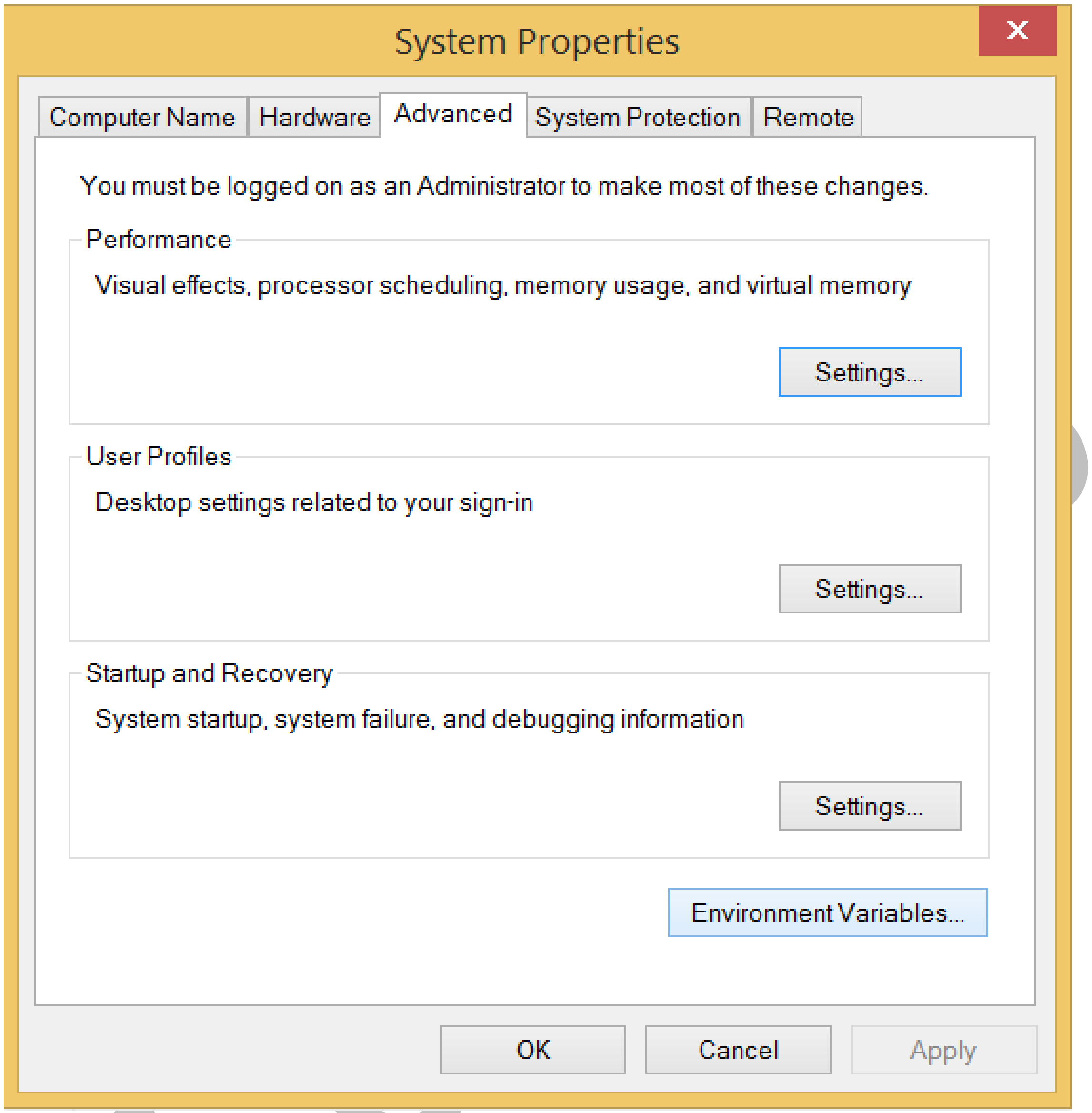
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

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

C:\Users\developer>_
```

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


File
Computer
View
This PC
Favorites
Desktop
Downloads
Recent places
This PC
Deskt
Docu
Dowr
Music
Pictur
Videc
Local
Win8
Local
Local
Softwares (H:)
Folders (6)
Desktop
Videos
(6)
:)
of 299 GB
l:)
of 7.49 GB
Collapse
Pin to Start
Manage
Open in new window
Map network drive...
Disconnect network drive...
Add a network location
Delete
Rename
Properties


System Properties

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

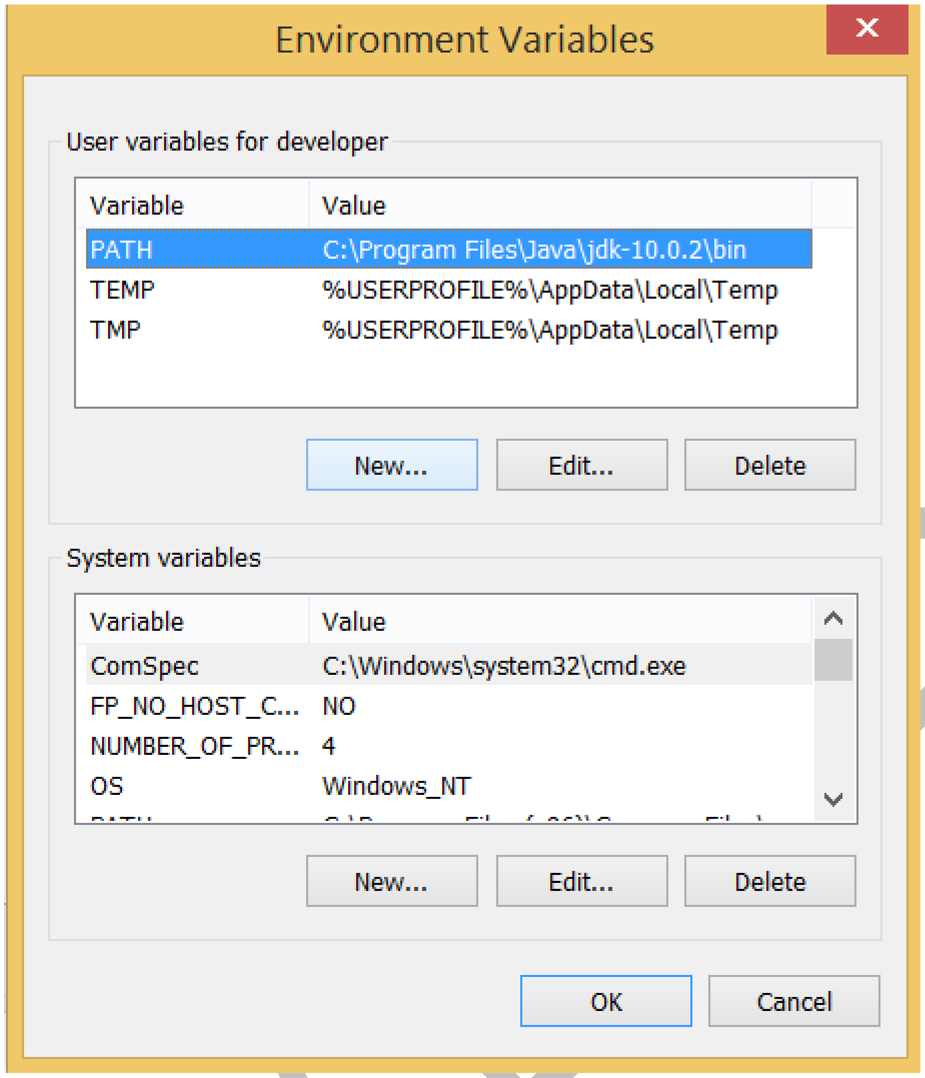
User Profiles

Desktop settings related to your sign-in

Startup and Recovery

System startup, system failure, and debugging information

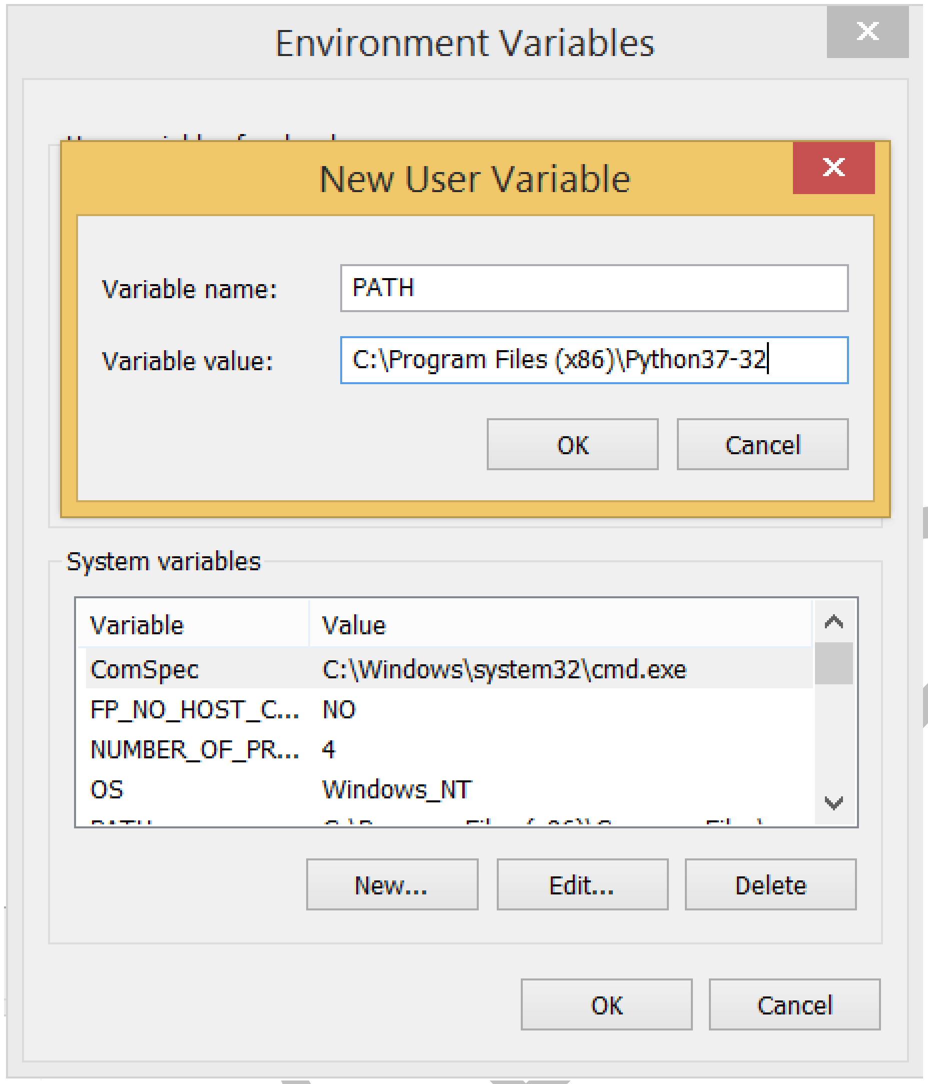
Add the new path variable in the user variable section.



Variable	Value
PATH	C:\Program Files\Java\jdk-10.0.2\bin
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_PR...	4
OS	Windows_NT

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



Variable name: PATH

Variable value: C:\Program Files (x86)\Python37-32

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_PR...	4
OS	Windows_NT

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

First Python Program

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

Python provides us the two ways to run a program:

- Using Interactive interpreter prompt
- Using a script file

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.

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

Identifier Naming

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

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore or digit (0-9).
- Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive for example my name, and MyName is not the same.
- Examples of valid identifiers : a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

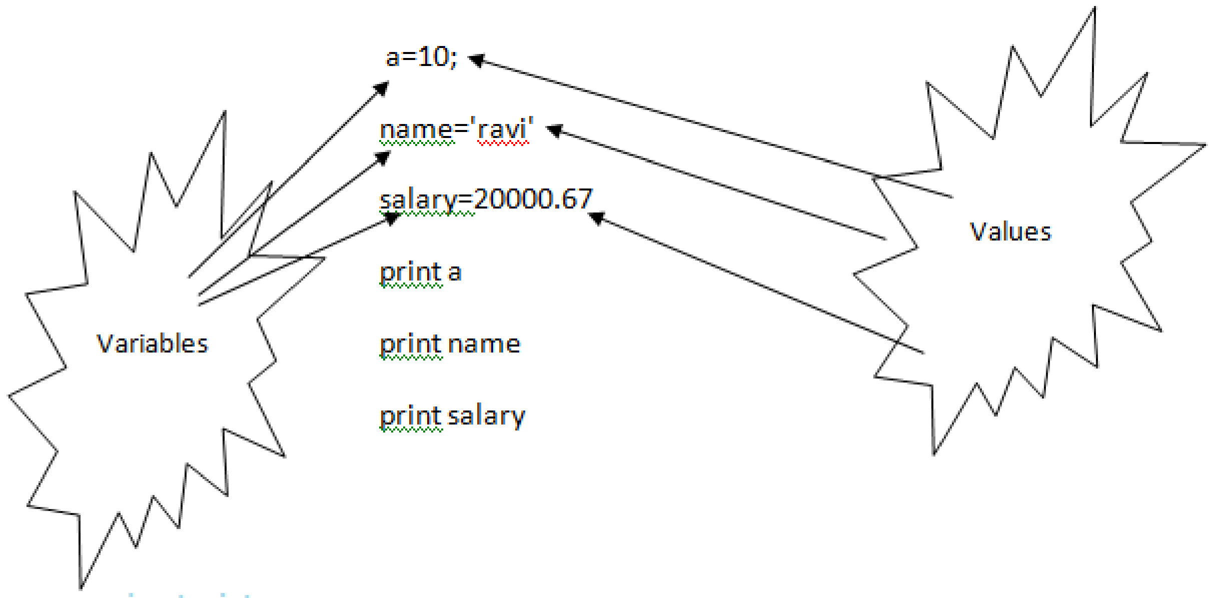
Declaring Variable and Assigning Values

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

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

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

Eg:



javatpoint.com

Output:

```
>>>
10
ravi
20000.67
>>>
```

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:

```
x=y=z=50
print iple
print y
print z
```

output:

```
>>>
50
50
50
>>>
```

2.Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
print a
print b
print c
```

Output:

```
>>>
5
10
15
>>>
```

There are following tokens in Python:

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

Python Data Types

```
A=10
b="Hi Python"
c = 10.5
print(type(a));
print(type(b));
print(type(c));
```

Output:

```
<type 'int'>
```

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

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:

```
>>> tuple=('rahul',100,60.4,'deepak')
>>> tuple1=('sanjay',10)
>>> tuple
('rahul', 100, 60.4, 'deepak')
>>> tuple[2:]
(60.4, 'deepak')
>>> tuple1[0]
'sanjay'
>>> tuple+tuple1
('rahul', 100, 60.4, 'deepak', 'sanjay', 10)
>>>
```

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:

```
>>> dictionary={'name':'charlie','id':100,'dept':'it'}
>>> dictionary
{'dept': 'it', 'name': 'charlie', 'id': 100}
>>> dictionary.keys()
['dept', 'name', 'id']
>>> dictionary.values()
['it', 'charlie', 100]
```

Python Data Types

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

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

```
A=10
b="Hi Python"
c = 10.5
print(type(a));
print(type(b));
print(type(c));
```

Standard data types

Python supports 4 types of numeric data.

int (signed integers like 10, 2, 29, etc.)

long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)

float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)

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

example, a person's name must be stored as a string whereas its id must be stored as an integer.

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

Numbers

String

List

Tuple

Dictionary

Numbers

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

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

Python supports 4 types of numeric data.

int (signed integers like 10, 2, 29, etc.)

long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)

float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)

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

String

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

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

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

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

The following example illustrates the string handling in python.

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

List

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

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

Consider the following example.

```
l  = [1, "hi", "python", 2]
print (l[3:]);
print (l[0:2]);
print (l);
print (l + l);
print (l * 3);
```

Tuple

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

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

Let's see a simple example of the tuple.

```
t  = ("hi", "python", 2)
print (t[1:]);
print (t[0:1]);
print (t);
print (t + t);
print (t * 3);
print (type(t))
t[2] = "hi";
```

Dictionary

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

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

Consider the following example.

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

Python Keywords

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

True	False	None	and	as
asset	def	class	continue	break
else	finally	elif	del	except
global	for	if	from	import
raise	try	or	return	pass
nonlocal	in	not	is	lambda

Python Literals

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

Python support the following literals:

Eg:

"Aman" , '12345'

Python Operators

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

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

- Identity Operators

Arithmetic operators

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

Comparison operator

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

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

Python assignment operators

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

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

Bitwise operator

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

For example,

```
if a = 7;
   b = 6;
then, binary (a) = 0111
    binary (b) = 0011

hence, a & b = 0011
       a | b = 0111
              a ^ b = 0100
        ~ a = 1000
```

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Logical Operators

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

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression **a** is true then not (a) will be false and vice versa.

Membership Operators

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

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

Identity Operators

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

Operator Precedence

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

Operator	Description

**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus and minus.
* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus and minus
>> <<	Left shift and right shift
&	Binary and.
^ |	Binary xor and or
<= < > >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python Comments

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

1) Single Line Comment:

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

Eg:

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

2) Multi Line Comment:

Multi lined comment can be given inside triple quotes.

eg:

```
""" This
    Is
    Multipline comment'''
```

Python If-else statements

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

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

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.

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

Indentation in Python

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

The elif statement

The syntax of the elif statement is given below.

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements


elif expression 3:



    # block of statements

else:
    # block of statements
```

Python Loops

Python for loop

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

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

python for loop example : printing the table of the given number

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

Nested for loop in python

Example 1

```
n = int(input("Enter the number of rows you want to print?"))
i,j=0,0
for i in range(0,n):
    print()
    for j in range(0,i+1):
        print("*",end="")
```

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

Python while loop

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

Example 1

```
i=1;
while i<=10:
   print(i);
   i=i+1;
```

Example 2

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

Infinite while loop

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

Using else with Python while loop

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

Example 2

```
i=1;
while i<=5:
   print(i)
```

```
    i=i+1;
    if(i==3):
        break;
else:print("The while loop exhausted");
```

Python break statement

The break is a keyword in python which is used to bring the program control out of the loop.

Example 1

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

Example 2

```
str = "python"
for i in str:
    if i == 'o':
        break
    print(i);
```

Output:

```
p
y
t
h
```

Example 3: break statement with while loop

```
i = 0;
while 1:
    print(i," ",end=""),
    i=i+1;
    if i == 10:
        break;
print("came out of while loop");
```

Python continue Statement

The continue statement in python is used to bring the program control to the beginning of the loop.

Example 1

```
i = 0;
while i!=10:
    print("%d"%i);
    continue;
    i=i+1;
```

Example 2

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

Pass Statement

The pass statement is a null operation since nothing happens when it is executed.

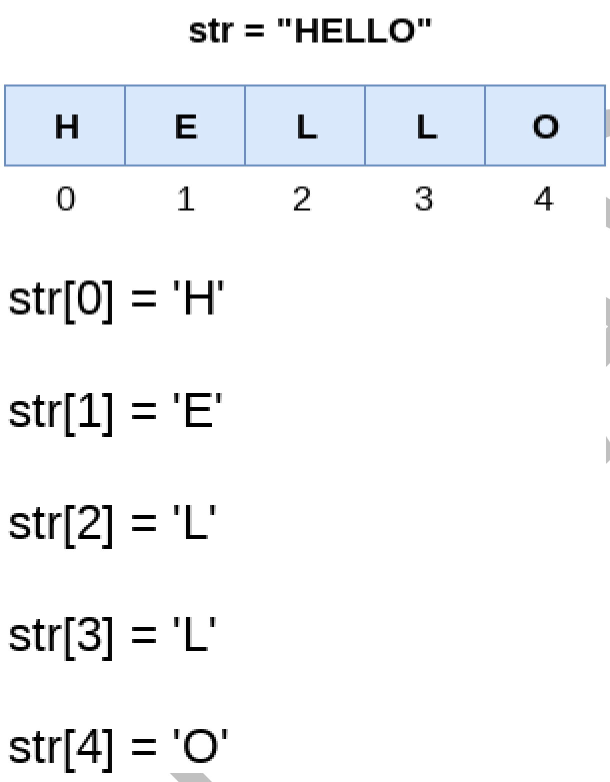
Example

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

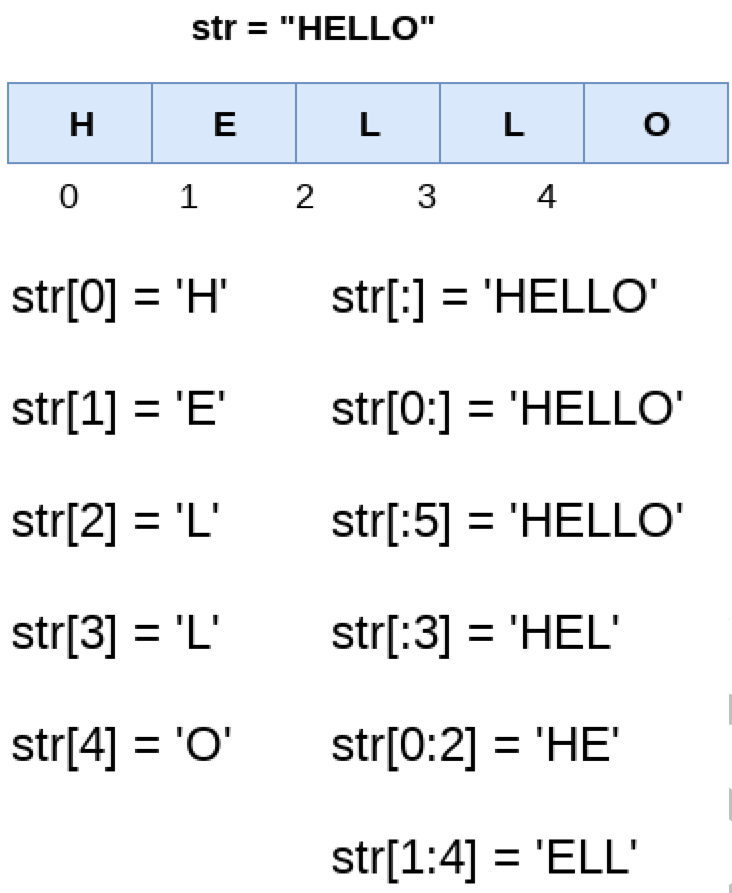
Python String

Strings indexing and splitting

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



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



```
str[0] = 'H'        str[:] = 'HELLO'
str[1] = 'E'        str[0:] = 'HELLO'
str[2] = 'L'        str[:5] = 'HELLO'
str[3] = 'L'        str[:3] = 'HEL'
str[4] = 'O'        str[0:2] = 'HE'
                    str[1:4] = 'ELL'
```

Reassigning strings

```
str = "HELLO"
print(str)
str = "hello"
print(str)
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.

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

Example

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

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

Python Formatting operator

```
Integer = 10;
Float = 1.290
String = "Ayush"
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... M
y value is %s"%(Integer,Float,String));
```

Built-in String functions

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

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

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

islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.
isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.
istitle()	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case.
isupper()	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
join(seq)	It merges the strings representation of the given sequence.
len(string)	It returns the length of a string.
ljust(width[,fillchar])	It returns the space padded strings with the original string left justified to the given width.
lower()	It converts all the characters of a string to Lower case.
lstrip()	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.

partition()	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
maketrans()	It returns a translation table to be used in translate function.
replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
rfind(str,beg=0,end=len(str))	It is similar to find but it traverses the string in backward direction.
rindex(str,beg=0,end=len(str))	It is same as index but it traverses the string in backward direction.
rjust(width,[,fillchar])	Returns a space padded string having original string right justified to the number of characters specified.
rstrip()	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.
rsplit(sep=None, maxsplit = -1)	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
split(str,num=string.count(str))	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.

splitlines(num=string.count('\n'))	It returns the list of strings at each line with newline removed.
startswith(str,beg=0,end=len(str))	It returns a Boolean value if the string starts with given str between begin and end.
strip([chars])	It is used to perform lstrip() and rstrip() on the string.
swapcase()	It inverts case of all characters in a string.
title()	It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut.
translate(table,deletechars = '')	It translates the string according to the translation table passed in the function .
upper()	It converts all the characters of a string to Upper Case.
zfill(width)	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
rpartition()	

Python List

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

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

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

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10];
Dep2 = ["IT",11];
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]
print("printing employee data...");
print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
print("printing departments...");
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],D
ep2[1],Dep2[0],Dep2[1]));
print("HOD Details ....");
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]));
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]));
print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT));
```

Output:

```
printing employee data...
Name : John, ID: 102, Country: USA
printing departments...
Department 1:
Name: CS, ID: 11
Department 2:
Name: IT, ID: 11
HOD Details ....
CS HOD Name: Mr. Holding, Id: 10
IT HOD Name: Mr. Bewon, Id: 11
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>
```

List indexing and splitting

Consider the following example.

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

0	**1**	**2**	**3**	**4**	**5**

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

Updating List values

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

Iterating a List

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

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

```
    print(i);
```

Output:

```
John
David
James
Jonathan
```

Adding elements to the list

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

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

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

Removing elements from the list

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

Python List Built-in functions

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

SN	Function	Description
1	cmp(list1, list2)	It compares the elements of both the lists.
2	len(list)	It is used to calculate the length of the list.
3	max(list)	It returns the maximum element of the list.
4	min(list)	It returns the minimum element of the list.
5	list(seq)	It converts any sequence to the list.

Python List built-in methods

SN	Function	Description
1	list.append(obj)	The element represented by the object obj is added to the list.
2	list.clear()	It removes all the elements from the list.
3	List.copy()	It returns a shallow copy of the list.
4	list.count(obj)	It returns the number of occurrences of the specified object in the list.
5	list.extend(seq)	The sequence represented by the object seq is extended to the list.
6	list.index(obj)	It returns the lowest index in the list that object appears.

7	list.insert(index, obj)	The object is inserted into the list at the specified index.
8	list.pop(obj=list[-1])	It removes and returns the last object of the list.
9	list.remove(obj)	It removes the specified object from the list.
10	list.reverse()	It reverses the list.
11	list.sort([func])	It sorts the list by using the specified compare function if given

Python Tuple

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

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

```
T1 = (101, "Ayush", 22)
T2 = ("Apple", "Banana", "Orange")
```

Example

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

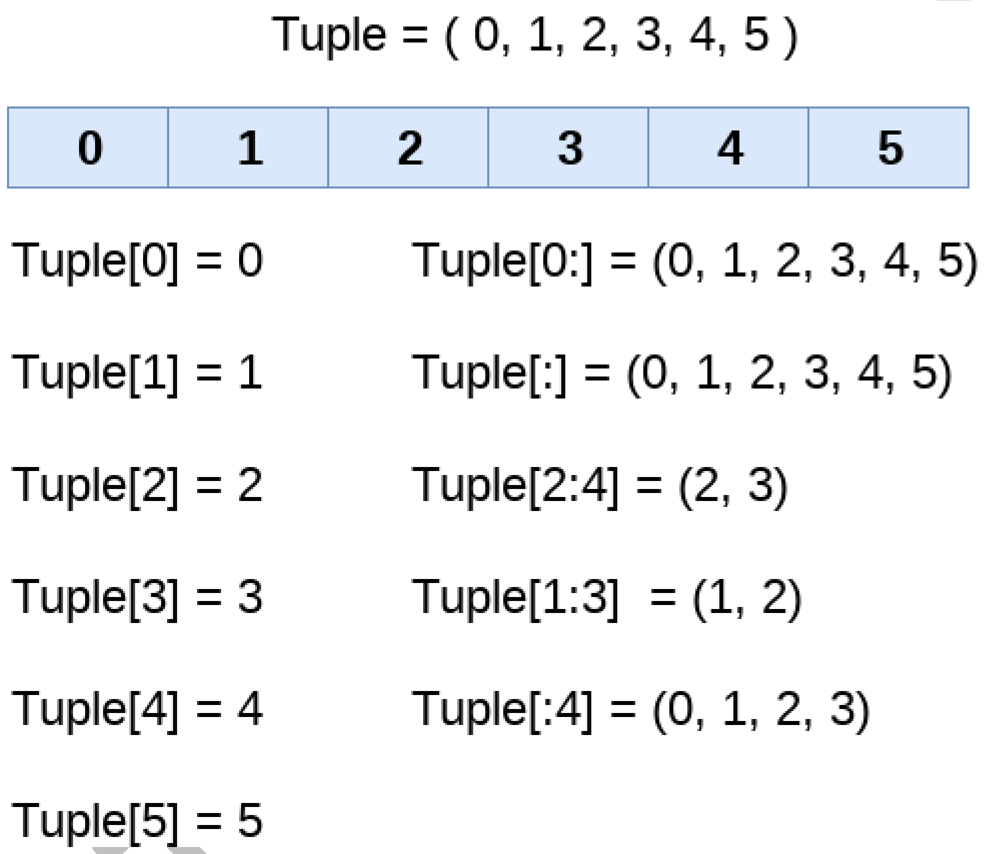
Example 2

```
tuple1 = tuple(input("Enter the tuple elements ..."))
print(tuple1)
```

```
count = 0
for i in tuple1:
   print("tuple1[%d] = %s"%(count, i));
```

Tuple indexing and splitting

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



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

0	1	2	3	4	5

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

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

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

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

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

Tuple[5] = 5
```

Python Functions

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

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

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

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

Advantage of functions in python

There are the following advantages of C functions.

By using functions, we can avoid rewriting same logic/code again and again in a program.

We can call python functions any number of times in a program and from any place in a program.

We can track a large python program easily when it is divided into multiple functions.

Reusability is the main achievement of python functions.

However, Function calling is always overhead in a python program.

Creating a function

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

```
def my_function():
    function-suite
    return <expression>
```

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

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

Function calling

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

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

```
def hello_world():
   print("hello world")

hello_world()
```

Output:

```
hello world
```

Parameters in function

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

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

Example 1

```
#defining the function
def func (name):
   print("Hi ",name);

#calling the function
func("Ayush")
```

Example 2

```
#python function to calculate the sum of two variables
#defining the function
def sum (a,b):
   return a+b;
```

```
#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

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

Call by reference in Python

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

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

Example 1 Passing Immutable Object (List)

```
#defining the function
def change_list(list1):
    list1.append(20);
    list1.append(30);
    print("list inside function = ",list1)

#defining the list
list1 = [10,30,40,50]

#calling the function
change_list(list1);
print("list outside function = ",list1);
```

Output:

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

Example 2 Passing Mutable Object (String)

```
#defining the function
def change_string (str):
    str = str + " Hows you";
    print("printing the string inside function :",str);

string1 = "Hi I am there"

#calling the function
change_string(string1)

print("printing the string outside function :",string1)
```

Output:

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

Types of arguments

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

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required Arguments

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

Consider the following example.

Example 1

```
#the argument name is the required argument to the function func
def func(name):
    message = "Hi "+name;
    return message;
name = input("Enter the name?")
print(func(name))
```

Output:

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

Example 2

```
#the function simple_interest accepts three arguments and returns the simple interest accordin
gly
def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output:

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

Example 3

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

Output:

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

Keyword arguments

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

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

Consider the following example.

Example 1

```
#function func is called with the name and message as the keyword arguments
def func(name,message):
   print("printing the message with",name,"and ",message)
func(name = "John",message="hello") #name and message is copied with the values John and
 hello respectively
```

Output:

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

Example 2 providing the values in different order at the calling

```
#The function simple_interest(p, t, r) is called with the keyword arguments the order of argum
ents doesn't matter in this case
def simple_interest(p,t,r):
   return (p*t*r)/100
print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Output:

```
Simple Interest:  1900.0
```

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

Consider the following example.

Example 3

```
#The function simple_interest(p, t, r) is called with the keyword arguments.
def simple_interest(p,t,r):
   return (p*t*r)/100

print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900)) # doesn't find the
 exact match of the name of the arguments (keywords)
```

Output:

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

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

Consider the following example.

Example 4

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

Output:

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

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

Example 5

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

Output:

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

Default Arguments

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

Example 1

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

Output:

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

Example 2

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

Output:

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

Variable length Arguments

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

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

Consider the following example.

Example

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

Output:

```
type of passed argument is  <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
```

Scope of variables

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

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

Global variables

Local variables

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

Consider the following example.

Example 1

```
def print_message():
   message = "hello !! I am going to print a message." # the variable message is local to the fu
nction itself
   print(message)
print_message()
print(message) # this will cause an error since a local variable cannot be accessible here.
```

Output:

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

Example 2

```
def calculate(*args):
   sum=0
   for arg in args:
      sum = sum +arg
   print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed
```

Output:

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

Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

```
#  integer number
integer = -20
print('Absolute value of -20 is:', abs(integer))

#  floating number
floating = -20.83
print('Absolute value of -20.83 is:', abs(floating))
```

Output:

```
Absolute value of -20 is: 20
Absolute value of -20.83 is: 20.83
```

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

```
# all values true
k = [1, 3, 4, 6]
print(all(k))

# all values false
k = [0, False]
print(all(k))

# one false value
k = [1, 3, 7, 0]
print(all(k))

# one true value
k = [0, False, 5]
print(all(k))

# empty iterable
k = []
print(all(k))
```

Output:

```
True
False
False
False
True
```

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

```
x =  10
```

```
y =  bin(x)
print (y)
```

Output:

```
0b1010
```

Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

Python bool() Example

```
test1 = []
print(test1,'is',bool(test1))
test1 = [0]
print(test1,'is',bool(test1))
test1 = 0.0
print(test1,'is',bool(test1))
test1 = None
print(test1,'is',bool(test1))
test1 = True
print(test1,'is',bool(test1))
test1 = 'Easy string'
print(test1,'is',bool(test1))
```

Output:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
```

Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

Python bytes() Example

```
string = "Hello World."
array = bytes(string, 'utf-8')
print(array)
```

Output:

```
b ' Hello World.'
```

Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

Python callable() Function Example

```
x = 8
print(callable(x))
```

Output:

```
False
```

Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

Python compile() Function Example

```
# compile string source to code
code_str = 'x=5\ny=10\nprint("sum =",x+y)'
code = compile(code_str, 'sum.py', 'exec')
print(type(code))
exec(code)
exec(x)
```

Output:

```
<class 'code'>
sum = 15
```

Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

Python exec() Function Example

```
x = 8
exec('print(x==8)')
exec('print(x+4)')
```

Output:

```
True
12
```

Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

Python sum() Function Example

```
s = sum([1, 2,4 ])
print(s)

s = sum([1, 2, 4], 10)
print(s)
```

Output:

```
7
17
```

Python any() Function

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

Python any() Function Example

```
l = [4, 3, 2, 0]
print(any(l))
```

```
l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

Python ascii() Function Example

```
normalText = 'Python is interesting'
print(ascii(normalText))

otherText = 'Pythön is interesting'
print(ascii(otherText))

print('Pyth\xf6n is interesting')
```

Output:

```
'Python is interesting'
'Pyth\xf6n is interesting'
Pythön is interesting
```

Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

Python bytearray() Example

```
string = "Python is a programming language."

# string with encoding 'utf-8'
arr = bytearray(string, 'utf-8')
print(arr)
```

Output:

```
bytearray(b'Python is a programming language.')
```

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

```
x = 8
print(eval('x + 1'))
```

Output:

```
9
```

Python float()

The python **float()** function returns a floating-point number from a number or string.

Python float() Example

```
# for integers
print(float(9))

# for floats
print(float(8.19))

# for string floats
print(float("-24.27"))

# for string floats with whitespaces
print(float("     -17.19\n"))
```

```
# string float error
print(float("xyz"))
```

Output:

```
9.0
8.19
-24.27
-17.19
ValueError: could not convert string to float: 'xyz'
```

Python format() Function

The python **format()** function returns a formatted representation of the given value.

Python format() Function Example

```
# d, f and b are a type

# integer
print(format(123, "d"))

# float arguments
print(format(123.4567898, "f"))

# binary format
print(format(12, "b"))
```

Output:

```
123
123.456790
1100
```

Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

Python frozenset() Example

```
# tuple of letters
letters = ('m', 'r', 'o', 't', 's')
```

```
fSet = frozenset(letters)
print('Frozen set is:', fSet)
print('Empty frozen set is:', frozenset())
```

Output:

```
Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})
Empty frozen set is: frozenset()
```

Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Python getattr() Function Example

```
class Details:
    age = 22
    name = "Phill"

details = Details()
print('The age is:', getattr(details, "age"))
print('The age is:', details.age)
```

Output:

```
The age is: 22
The age is: 22
```

Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python globals() Function Example

```
age = 22

globals()['age'] = 22
print('The age is:', age)
```

Output:

```
The age is: 22
```

Python hasattr() Function

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

Python hasattr() Function Example

```
l = [4, 3, 2, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

Python iter() Function Example

```
# list of numbers
list = [1,2,3,4,5]

listIter = iter(list)

# prints '1'
print(next(listIter))
```

```
# prints '2'
print(next(listIter))

# prints '3'
print(next(listIter))

# prints '4'
print(next(listIter))

# prints '5'
print(next(listIter))
```

Output:

```
1
2
3
4
5
```

Python len() Function

The python **len()** function is used to return the length (the number of items) of an object.

Python len() Function Example

```
strA = 'Python'
print(len(strA))
```

Output:

```
6
```

Python list()

The python **list()** creates a list in python.

Python list() Example

```
# empty list
print(list())

# string
String = 'abcde'
```

```
print(list(String))

# tuple
Tuple = (1,2,3,4,5)
print(list(Tuple))
# list
List = [1,2,3,4,5]
print(list(List))
```

Output:

```
[]
['a', 'b', 'c', 'd', 'e']
[1,2,3,4,5]
[1,2,3,4,5]
```

Python locals() Function

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python locals() Function Example

```
def localsAbsent():
    return locals()

def localsPresent():
    present = True
    return locals()

print('localsNotPresent:', localsAbsent())
print('localsPresent:', localsPresent())
```

Output:

```
localsAbsent: {}
localsPresent: {'present': True}
```

Python map() Function

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

Python map() Function Example

```
def calculateAddition(n):
  return n+n

numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(result)

# converting map object to set
numbersAddition = set(result)
print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Python memoryview() Function

The python **memoryview()** function returns a memoryview object of the given argument.

Python memoryview () Function Example

```
#A random bytearray
randomByteArray = bytearray('ABC', 'utf-8')

mv = memoryview(randomByteArray)

# access the memory view's zeroth index
print(mv[0])

# It create byte from memory view
print(bytes(mv[0:2]))

# It create list from memory view
print(list(mv[0:3]))
```

Output:

```
65
b'AB'
[65, 66, 67]
```

Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

Python object() Example

```
python = object()

print(type(python))
print(dir(python))
```

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__',
'__str__', '__subclasshook__']
```

Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

Python open() Function Example

```
# opens python.text file of the current directory
f = open("python.txt")
# specifying full path
f = open("C:/Python33/README.txt")
```

Output:

```
Since the mode is omitted, the file is opened in 'r' mode; opens for reading.
```

Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

Python chr() Function Example

```
# Calling function
result = chr(102) # It returns string representation of a char
result2 = chr(112)
# Displaying result
print(result)
print(result2)
# Verify, is it string type?
print("is it string type:", type(result) is str)
```

Output:

```
ValueError: chr() arg not in range(0x110000)
```

Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

Python complex() Example

```
# Python complex() function example
# Calling function
a = complex(1) # Passing single parameter
b = complex(1,2) # Passing both parameters
# Displaying result
print(a)
print(b)
```

Output:

```
(1.5+0j)
(1.5+2.2j)
```

Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Python delattr() Function Example

```
class Student:
    id = 101
    name = "Pranshu"
    email = "pranshu@abc.com"
# Declaring function
    def getinfo(self):
        print(self.id, self.name, self.email)
s = Student()
s.getinfo()
delattr(Student,'course') # Removing attribute which is not available
s.getinfo() # error: throws an error
```

Output:

```
101 Pranshu pranshu@abc.com
AttributeError: course
```

Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named __dir__(), this method will be called and must return the list of attributes. It takes a single object type argument.

Python dir() Function Example

```
# Calling function
att = dir()
# Displaying result
print(att)
```

Output:

```
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__',
'__name__', '__package__', '__spec__']
```

Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

Python divmod() Function Example

```
# Python divmod() function example
# Calling function
result = divmod(10,2)
# Displaying result
print(result)
```

Output:

```
(5, 0)
```

Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

Python enumerate() Function Example

```
# Calling function
result = enumerate([1,2,3])
# Displaying result
print(result)
print(list(result))
```

Output:

```
<enumerate object at 0x7ff641093d80>
[(0, 1), (1, 2), (2, 3)]
```

Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

- If no argument is passed, it creates an empty dictionary.

- If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
- If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

Python dict() Example

```
# Calling function
result = dict() # returns an empty dictionary
result2 = dict(a=1,b=2)
# Displaying result
print(result)
print(result2)
```

Output:

```
{}
{'a': 1, 'b': 2}
```

Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

Python filter() Function Example

```
# Python filter() function example
def filterdata(x):
    if x>5:
        return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
print(list(result))
```

Output:

```
[6]
```

Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

Hashable types: * bool * int * long * float * string * Unicode * tuple * code object.

Python hash() Function Example

```
# Calling function
result = hash(21) # integer value
result2 = hash(22.2) # decimal value
# Displaying result
print(result)
print(result2)
```

Output:

```
21
461168601842737174
```

Python help() Function

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Python help() Function Example

```
# Calling function
info = help() # No argument
# Displaying result
print(info)
```

Output:

```
Welcome to Python 3.5's help utility!
```

Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

Python min() Function Example

```
# Calling function
small = min(2225,325,2025) # returns smallest element
small2 = min(1000.25,2025.35,5625.36,10052.50)
# Displaying result
print(small)
print(small2)
```

Output:

```
325
1000.25
```

Python set() Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

Python set() Function Example

```
# Calling function
result = set() # empty set
result2 = set('12')
result3 = set('javatpoint')
# Displaying result
print(result)
print(result2)
print(result3)
```

Output:

```
set()
{'1', '2'}
{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}
```

Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

Python hex() Function Example

```
# Calling function
result = hex(1)
# integer value
result2 = hex(342)
# Displaying result
print(result)
print(result2)
```

Output:

```
0x1
0x156
```

Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

Python id() Function Example

```
# Calling function
val = id("Javatpoint") # string object
val2 = id(1200) # integer object
val3 = id([25,336,95,236,92,3225]) # List object
# Displaying result
print(val)
print(val2)
print(val3)
```

Output:

```
139963782059696
139963805666864
139963781994504
```

Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

Python setattr() Function Example

```
class Student:
    id = 0
    name = ""

    def __init__(self, id, name):
        self.id = id
        self.name = name

student = Student(102,"Sohan")
print(student.id)
print(student.name)
#print(student.email) product error
setattr(student, 'email','sohan@abc.com') # adding new attribute
print(student.email)
```

Output:

```
102
Sohan
sohan@abc.com
```

Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

Python slice() Function Example

```
# Calling function
result = slice(5) # returns slice object
result2 = slice(0,5,3) # returns slice object
# Displaying result
print(result)
print(result2)
```

Output:

```
slice(None, 5, None)
slice(0, 5, 3)
```

Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

Python sorted() Function Example

```
str = "javatpoint" # declaring string
# Calling function
sorted1 = sorted(str) # sorting string
# Displaying result
print(sorted1)
```

Output:

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

Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

Python next() Function Example

```
number = iter([256, 32, 82]) # Creating iterator
# Calling function
item = next(number)
# Displaying result
print(item)
# second item
item = next(number)
print(item)
# third item
```

```
item = next(number)
print(item)
```

Output:

```
256
32
82
```

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

Python input() Function Example

```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Output:

```
Enter a value: 45
You entered: 45
```

Python int() Function

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

Python int() Function Example

```
# Calling function
val = int(10) # integer value
val2 = int(10.52) # float value
val3 = int('10') # string value
# Displaying result
print("integer values :",val, val2, val3)
```

Output:

```
integer values : 10 10 10
```

Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

Python isinstance() function Example

```
class Student:
    id = 101
    name = "John"
    def __init__(self, id, name):
        self.id=id
        self.name=name

student = Student(1010,"John")
lst = [12,34,5,6,767]
# Calling function
print(isinstance(student, Student)) # isinstance of Student class
print(isinstance(lst, Student))
```

Output:

```
True
False
```

Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

Python oct() function Example

```
# Calling function
val = oct(10)
# Displaying result
```

```
print("Octal value of 10:",val)
```

Output:

```
Octal value of 10: 0o12
```

Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

Python ord() function Example

```
# Code point of an integer
print(ord('8'))

# Code point of an alphabet
print(ord('R'))

# Code point of a character
print(ord('&'))
```

Output:

```
56
82
38
```

Python pow() Function

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

Python pow() function Example

```
# positive x, positive y (x**y)
print(pow(4, 2))

# negative x, positive y
print(pow(-4, 2))

# positive x, negative y (x**-y)
```

```
print(pow(4, -2))

# negative x, negative y
print(pow(-4, -2))
```

Output:

```
16
16
0.0625
0.0625
```

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

Python print() function Example

```
print("Python is programming language.")

x = 7
# Two objects passed
print("x =", x)

y = x
# Three objects passed
print('x =', x, '= y')
```

Output:

```
Python is programming language.
x = 7
x = 7 = y
```

Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

Python range() function Example

```
# empty range
print(list(range(0)))
```

```
# using the range(stop)
print(list(range(4)))

# using the range(start, stop)
print(list(range(1,7 )))
```

Output:

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

Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

Python reversed() function Example

```
# for string
String = 'Java'
print(list(reversed(String)))

# for tuple
Tuple = ('J', 'a', 'v', 'a')
print(list(reversed(Tuple)))

# for range
Range = range(8, 12)
print(list(reversed(Range)))

# for list
List = [1, 2, 7, 5]
print(list(reversed(List)))
```

Output:

```
['a', 'v', 'a', 'J']
['a', 'v', 'a', 'J']
[11, 10, 9, 8]
[5, 7, 2, 1]
```

Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

Python round() Function Example

```
#  for integers
print(round(10))

#  for floating point
print(round(10.8))

#  even choice
print(round(6.6))
```

Output:

```
10
11
7
```

Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

Python issubclass() Function Example

```
class Rectangle:
  def __init__(rectangleType):
    print('Rectangle is a ', rectangleType)

class Square(Rectangle):
  def __init__(self):
    Rectangle.__init__('square')

print(issubclass(Square, Rectangle))
print(issubclass(Square, list))
print(issubclass(Square, (list, Rectangle)))
print(issubclass(Rectangle, (list, Rectangle)))
```

Output:

```
True
False
True
True
```

Python str

The python **str()** converts a specified value into a string.

Python str() Function Example

```
str('4')
```

Output:

```
'4'
```

Python tuple() Function

The python **tuple()** function is used to create a tuple object.

Python tuple() Function Example

```
t1 = tuple()
print('t1=', t1)

# creating a tuple from a list
t2 = tuple([1, 6, 9])
print('t2=', t2)

# creating a tuple from a string
t1 = tuple('Java')
print('t1=',t1)

# creating a tuple from a dictionary
t1 = tuple({4: 'four', 5: 'five'})
print('t1=',t1)
```

Output:

```
t1= ()
t2= (1, 6, 9)
t1= ('J', 'a', 'v', 'a')
t1= (4, 5)
```

Python type()

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

Python type() Function Example

```
List = [4, 5]
print(type(List))

Dict = {4: 'four', 5: 'five'}
print(type(Dict))

class Python:
    a = 0

InstanceOfPython = Python()
print(type(InstanceOfPython))
```

Output:

```
<class 'list'>
<class 'dict'>
<class '__main__.Python'>
```

Python vars() function

The python **vars()** function returns the __dict__ attribute of the given object.

Python vars() Function Example

```
class Python:
  def __init__(self, x = 7, y = 9):
    self.x = x
    self.y = y

InstanceOfPython = Python()
print(vars(InstanceOfPython))
```

Output:

```
{'y': 9, 'x': 7}
```

Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

Python zip() Function Example

```
numList = [4,5, 6]
strList = ['four', 'five', 'six']

# No iterables are passed
result = zip()

# Converting itertor to list
resultList = list(result)
print(resultList)

# Two iterables are passed
result = zip(numList, strList)

# Converting itertor to set
resultSet = set(result)
print(resultSet)
```

Output:

```
[]
{(5, 'five'), (4, 'four'), (6, 'six')}
```

Python Lambda Functions

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

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

The syntax to define an Anonymous function is given below.

```
lambda arguments : expression
```

Example 1

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

Output:

```
sum =  30
```

Example 2

Multiple arguments to Lambda function

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

Output:

```
sum =  30
```

Why use lambda functions?

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

Example 1

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

Output:

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

Example 2

Use of lambda function with filter

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

Output:

```
[3, 123]
```

Example 3

Use of lambda function with map

```
#program to triple each number of the list using map
```

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

Output:

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

Python File Handling

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

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

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

Opening a file

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

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

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

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

SN	Access mode	Description
1	r	It opens the file to read-only. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read only in binary format. The file pointer exists at the beginning of the file.

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

Example

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")
```

Output:

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

The close() method

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

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

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

```
fileobject.close()
```

Consider the following example.

Example

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")

#closes the opened file
fileptr.close()
```

Reading the file

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

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

```
fileobj.read(<count>)
```

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

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file.txt","r");

#stores all the data of the file into the variable content
content = fileptr.read(9);

# prints the type of the data stored in the file
print(type(content))

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

Output:

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

Read Lines of the file

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

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

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readline();

# prints the type of the data stored in the file
print(type(content))

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

Output:

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

Looping through the file

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

Example

```
#open the file.txt in read mode. causes an error if no such file exists.


fileptr = open("file.txt","r");

#running a for loop
for i in fileptr:
   print(i) # i contains each line of the file
```

Output:

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

Writing the file

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

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

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

Consider the following example.

Example 1

```
#open the file.txt in append mode. Creates a new file if no such file exists.
fileptr = open("file.txt","a");

#appending the content to the file
fileptr.write("Python is the modern day language. It makes things so simple.")


#closing the opened file
fileptr.close();
```

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

File.txt:

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

Example 2

```
#open the file.txt in write mode.
fileptr = open("file.txt","w");

#overwriting the content of the file
fileptr.write("Python is the modern day language. It makes things so simple.")
```

```
#closing the opened file
fileptr.close();
```

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

File.txt:

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

Creating a new file

The new file can be created by using one of the following access modes with the function open(). **x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

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

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

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","x");

print(fileptr)

if fileptr:
   print("File created successfully");
```

Output:

```
 File created successfully
```

Using with statement with files

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

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

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

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

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

Consider the following example.

Example

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

Output:

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

File Pointer positions

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

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file
content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

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

Python Modules

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

Example

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

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

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

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

Loading the module in our python code

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

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

The import statement

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

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

The syntax to use the import statement is given below.

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

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

Example:

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

Output:

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

The from-import statement

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

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

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

calculation.py:

```
#place the code in the calculation.py
def summation(a,b):
   return a+b
def multiplication(a,b):
   return a*b;
def divide(a,b):
   return a/b;
```

Main.py:

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

Output:

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

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

Consider the following syntax.

```
from <module> import *
```

Renaming a module

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

The syntax to rename a module is given below.

```
import <module-name> as <specific-name>
```

Example

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

Output:

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

Using dir() function

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

Consider the following example.

Example

```
import json

List = dir(json)

print(List)
```

Output:

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

The reload() function

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

1. reload(<module-name>)

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

1. reload(calculation)

Scope of variables

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

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

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

Consider the following example.

Example

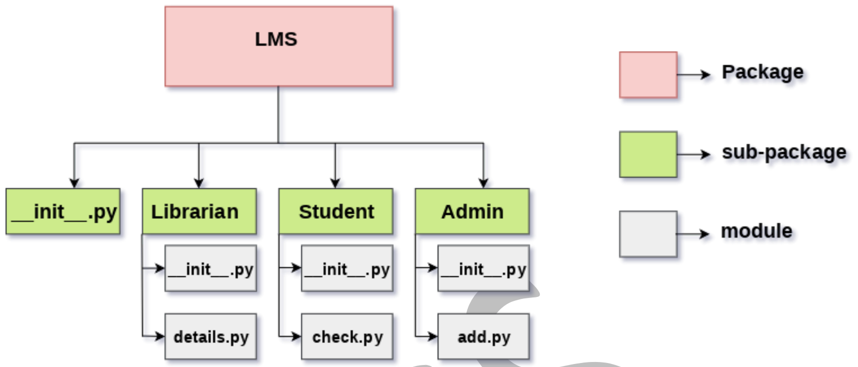
```
name = "john"
def print_name(name):
   print("Hi",name) #prints the name that is local to this function only.
name = input("Enter the name?")
print_name(name)
```

Output:

```
Hi David
```

Python packages

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



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

1. Create a directory with name Employees on path /**home**.

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

ITEmployees.py

```
def getITNames():
    List = ["John", "David", "Nick",    "Martin"]
    return List;
```

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is __init__.py which contains the import statements of the modules defined in this directory.

__init__.py

```
from ITEmployees import getITNames
from BPOEmployees import getBPONames
```

5. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create __init__.py inside a directory to convert this directory to a package.

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

Test.py

```
import Employees
print(Employees.getNames())
```

Output:

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

Python Date and time

In the real world applications, there are the scenarios where we need to work with the date and time. There are the examples in python where we have to schedule the script to run at some particular timings.

In python, the date is not a data type, but we can work with the date objects by importing the module named with datetime, time, and calendar.

Tick

In python, the time instants are counted since 12 AM, 1st January 1970. The function time() of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Consider the following example.

Example

```
import time;

#prints the number of ticks spent since 12 AM, 1st January 1970

print(time.time())
```

Output:

```
1545124460.9151757
```

How to get the current time?

The localtime() functions of the time module are used to get the current time tuple. Consider the following example.

Example

```
import time;

#returns a time tuple
```

```
print(time.localtime(time.time()))
```

Output:

```
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=18, tm_hour=15, tm_min=1,
tm_sec=32, tm_wday=1, tm_yday=352, tm_isdst=0)
```

Time tuple

The time is treated as the tuple of 9 numbers. Let's look at the members of the time tuple.

Index	Attribute	Values
0	Year	4 digit (for example 2018)
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 60
6	Day of weak	0 to 6
7	Day of year	1 to 366
8	Daylight savings	-1, 0, 1 , or -1

Getting formatted time

The time can be formatted by using the asctime() function of time module. It returns the formatted time for the time tuple being passed.

Example

```
import time;

#returns the formatted time

print(time.asctime(time.localtime(time.time())))
```

Output:

```
Tue Dec 18 15:31:39 2018
```

Python sleep time

The sleep() method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds given as float.

Consider the following example.

Example

```
import time
for i in range(0,5):
   print(i)
   #Each element will be printed after 1 second
   time.sleep(1)
```

Output:

```
0
1
2
3
4
```

The datetime Module

The datetime module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import datetime module into the python source code.

Consider the following example to get the datetime object representation for the current time.

Example

```
import datetime;

#returns the current datetime object

print(datetime.datetime.now())
```

Output:

```
2018-12-18 16:16:45.462778
```

Creating date objects

We can create the date objects by passing the desired date in the datetime constructor for which the date objects are to be created.

Consider the following example.

Example

```
import datetime;

#returns the datetime object for the specified date

print(datetime.datetime(2018,12,10))
```

Output:

```
2018-12-10 00:00:00
```

We can also specify the time along with the date to create the datetime object. Consider the following example.

Example

```
import datetime;

#returns the datetime object for the specified time

print(datetime.datetime(2018,12,10,14,15,10))
```

Output:

```
2018-12-10 14:15:10
```

Comparison of two dates

We can compare two dates by using the comparison operators like >, >=, <, and <=.

Consider the following example.

Example

```
from datetime import datetime as dt
#Compares the time. If the time is in between 8AM and 4PM, then it prints working hours other wise it prints fun hours
if dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):
    print("Working hours....")
else:
    print("fun hours")
```

Output:

```
fun hours
```

The calendar module

Python provides a calendar object that contains various methods to work with the calendars.

Consider the following example to print the Calendar of the last month of 2018.

Example

```
import calendar;
cal = calendar.month(2018,12)
#printing the calendar of December 2018
```

Python OOPs Concepts

Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

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

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

Object

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

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

Class

The class can be defined as a blue print 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

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

Method

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

Inheritance

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

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

It provides re-usability of the code.

Polymorphism

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

Encapsulation

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

Data Abstraction

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

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

Python Class and Objects

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

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

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

Creating classes in python

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

Syntax

```
1.  class ClassName:
2.      #statement_suite
```

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

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

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

Example

```
class Employee:
    id = 10;
    name = "ayush"
    def display (self):
        print(self.id,self.name)
```

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

Creating an instance of the class

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

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

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

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

Example

```
class Employee:
    id = 10;
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: ayush
```

Python Constructor

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

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

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

Creating the constructor in python

In python, the method **__init__** simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon **__init__** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the Employee class attributes.

Example

```
class Employee:
    def __init__(self,name,id):
        self.id = id;
        self.name = name;
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
emp1 = Employee("John",101)
emp2 = Employee("David",102)
#accessing display() method to print employee 1 information

emp1.display();

#accessing display() method to print employee 2 information
emp2.display();
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Example: Counting the number of objects of a class

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor Example

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

Output:

```
This is non parametrized constructor
Hello John
```

Python Parameterized Constructor Example

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

Output:

```
This is parametrized constructor
Hello John
```

Python In-built class functions

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

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.

3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Example

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age

#creates the object of the class Student
s = Student("John",101,22)

#prints the attribute name of the object s
print(getattr(s,'name'))

# reset the value of attribute age to 23
setattr(s,"age",23)

# prints the modified value of age
print(getattr(s,'age'))

# prints true if the student contains the attribute with name id

print(hasattr(s,'id'))
# deletes the attribute age
delattr(s,'age')

# this will give an error since the attribute age has been deleted
print(s.age)
```

Output:

```
John
23
True
AttributeError: 'Student' object has no attribute 'age'
```

Built-in class attributes

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

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

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Example

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))
s = Student("John",101,22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Output:

```
None
{'name': 'John', 'id': 101, 'age': 22}
```

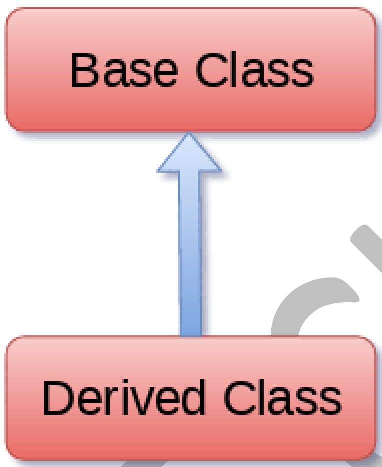
```
__main__
```

Python Inheritance

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

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

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



Syntax

```
class derived-class(base class):
   <class-suite>
```

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

Syntax

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

Example 1

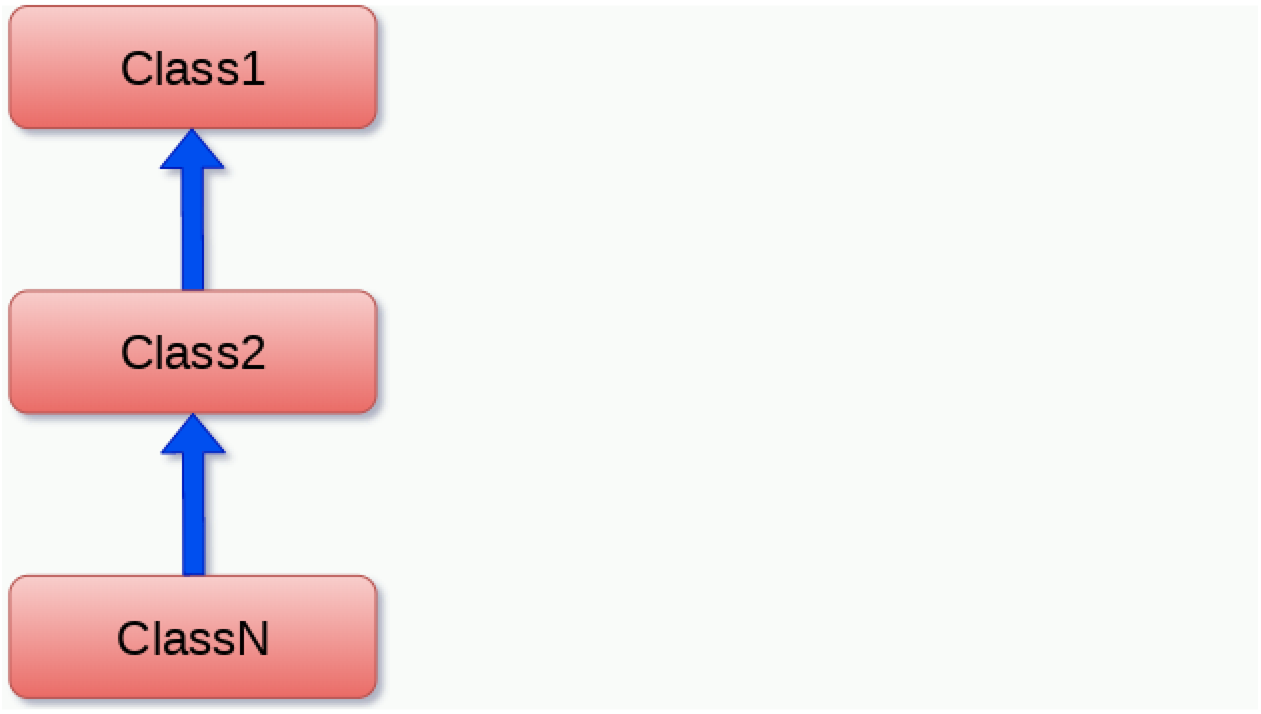
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

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



The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```

Example

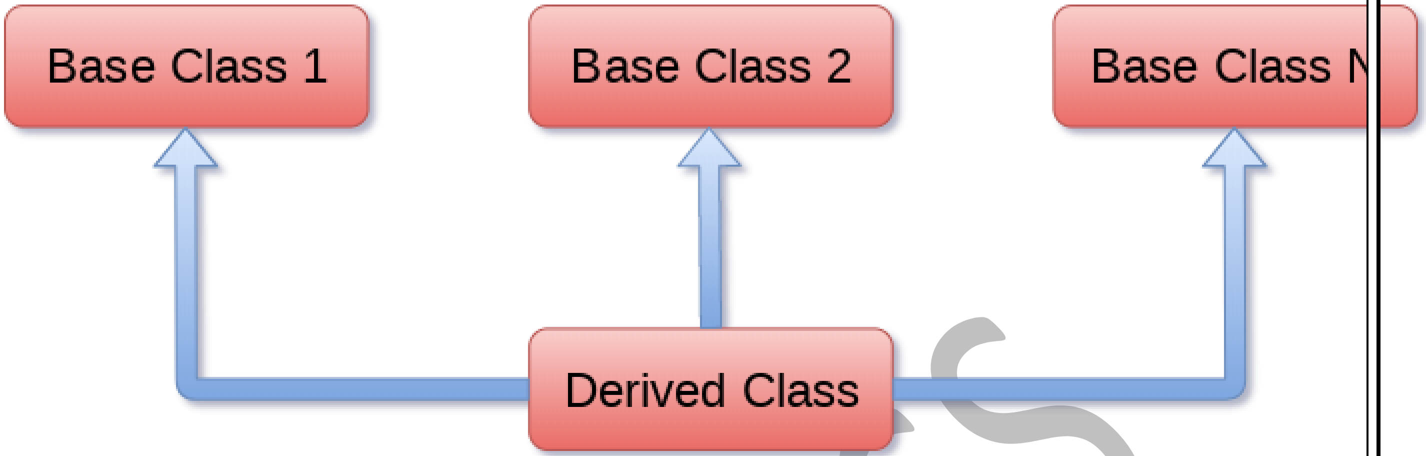
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

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



The syntax to perform multiple inheritance is given below.

The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ...... BaseN):
    <class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
```

```
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

The issubclass(sub,sup) method

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

Consider the following example.

Example

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

Output:

```
True
False
```

The isinstance (obj, class) method

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

Consider the following example.

Example

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(isinstance(d,Derived))
```

Output:

```
True
```

Method Overriding

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

Consider the following example to perform method overriding in python.

Example

```
class Animal:
   def speak(self):
      print("speaking")
class Dog(Animal):
   def speak(self):
      print("Barking")
d = Dog()
d.speak()
```

Output:

```
Barking
```

Real Life Example of method overriding

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

Data abstraction in python

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

Consider the following example.

Example

```
class Employee:
    __count = 0;
    def __init__(self):
        Employee.__count = Employee.__count+1
    def display(self):
        print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
try:
    print(emp.__count)
finally:
    emp.display()
```

Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```