Skillmine
Technology ▪ Consulting ▪ Services

Python Documentation

BY

RANJAN KUMAR

Introduction

- Python is a general purpose high level programming language.
- Python was developed by Guido Van Rossam in 1989 while working at National Research Institute at Netherlands.
- But officially Python was made available to public in 1991. The official Date of Birth for Python is : Feb 20th 1991.

Example: To Print Sum of two numbers

Java:

```
1)  public class Add
2)  {
3)     public static void main(String[] args)
4)     {
5)        int a,b;
6)        a =10;
7)        b=20;
8)        System.out.println("The Sum:"+(a+b));
9)     }
10) }
```

C:

```
1)  #include <stdio.h>
2)
3)  void main()
4)  {
5)     int a,b;
6)     a =10;
7)     b=20;
8)     printf("The Sum:%d",(a+b));
9)  }
```

Python:

```
1)  a=10
2)  b=20
3)  print("The Sum:",(a+b))
```

Where we can use Python:

We can use everywhere. The most common important application areas are

1. For developing Desktop Applications

2. For developing web Applications

3. For developing database Applications

4. For Network Programming 5. For developing games

6. For Data Analysis Applications

7. For Machine Learning

8. For developing Artificial Intelligence Applications

9. For IOT

Note:

Internally Google and Youtube use Python coding NASA and Nework Stock Exchange Applications developed by Python. Top Software companies like Google, Microsoft, IBM, Yahoo using Python

Features of Python:

1. Simple and easy to learn:

Python is a simple programming language. When we read Python program, we can feel like reading english statements. The syntaxes are very simple and only 30+ kerywords are available. When compared with other languages, we can write programs with very less number of lines. Hence more readability and simplicity. We can reduce development and cost of the project.

2. Freeware and Open Source:

We can use Python software without any licence and it is freeware. Its source code is open so that we can we can customize based on our requirement.

Eg: Jython is customized version of Python to work with Java Applications.

3. High Level Programming language:

Python is high level programming language and hence it is programmer friendly language. Being a programmer we are not required to concentrate low level activities like memory management and security etc..

4. Platform Independent:

Once we write a Python program,it can run on any platform without rewriting once again. Internally PVM is responsible to convert into machine understandable form.

5. Portability:

Python programs are portable. ie we can migrate from one platform to another platform very easily. Python programs will provide same results on any paltform.

6. Dynamically Typed:

In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically.Hence Python is considered as dynamically typed language.

But Java, C etc are Statically Typed Languages b'z we have to provide type at the beginning only.

This dynamic typing nature will provide more flexibility to the programmer.

7. Both Procedure Oriented and Object Oriented:

Python language supports both Procedure oriented (like C, pascal etc) and object oriented (like C++,Java) features. Hence we can get benefits of both like security and reusability etc

8. Interpreted:

We are not required to compile Python programs explcitly. Internally Python interpreter will take care that compilation.

If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

9. Extensible:

We can use other language programs in Python. The main advantages of this approach are:

1. We can use already existing legacy non-Python code

2. We can improve performance of the application

10. Embedded:

We can use Python programs in any other language programs. i.e we can embedd Python programs anywhere.

11. Extensive Library:

Python has a rich inbuilt library. Being a programmer we can use this library directly and we are not responsible to implement the functionality.

Reserved Words In Python:

In Python some words are reserved to represent some meaning or functionality. Such type of words are called Reserved words.

There are 33 reserved words available in Python.

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

Note:

1. All Reserved words in Python contain only alphabet symbols.

2. Except the following 3 reserved words, all contain only lower case alphabet symbols.

- **True**
- **False**
- **None**

```
>>> import keyword

 >>> keyword.kwlist

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

Data Types In Python:

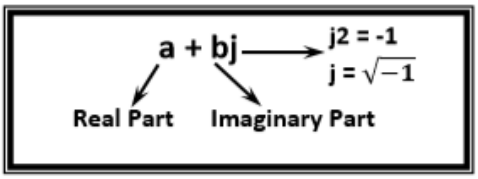
Data Type represent the type of data present inside a variable. In Python we are not required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence Python is Dynamically Typed Language.

Python contains the following inbuilt data types

- 1.int
- 2. float
- 3.complex
- 4.bool
- 5.str
- 6.bytes
- 7.bytearray
- 8.range
- 9.list
- 10.tuple
- 11.set
- 12.frozenset
- 13.dict
- 14.None

Complex Data Type:

A complex number is of the form



a and b contain intergers or floating point values

Eg:

3+5j
10+5.5j
0.5+0.1j

In the real part if we use int value then we can specify that either by decimal,octal,binary or hexa decimal form.
But imaginary part should be specified only by using decimal form.

```
1) >>> a=0B11+5j
2) >>> a
3) (3+5j)
4) >>> a=3+0B11j
5)   SyntaxError: invalid syntax
```

Even we can perform operations on complex type values.

Note: Complex data type has some inbuilt attributes to retrieve the real part and imaginary part

c=10.5+3.6j

c.real==>10.5

c.imag==>3.6

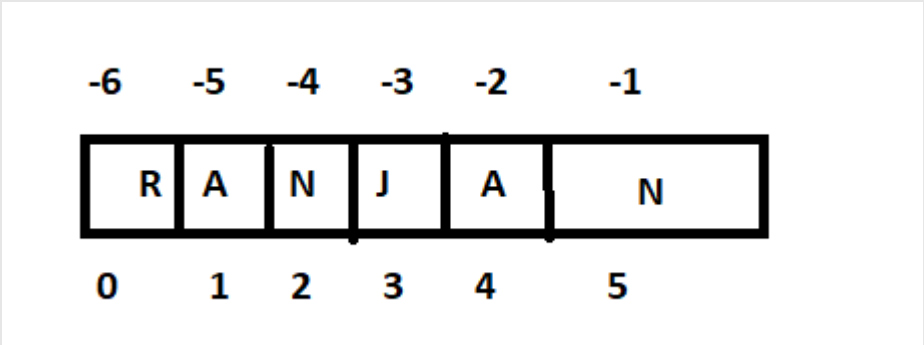
We can use complex type generally in scientific Applications and electrical engineering Applications.

Slicing of Strings:

slice means a piece [ ] operator is called slice operator, which can be used to retrieve parts of String. In Python Strings follows zero based index. The index can be either +ve or -ve. +ve index means forward direction from Left to Right -ve index means backward direction from Right to Left

```
>>> s="Ranjan"
>>> s[0]
'R'
>>> s[1]
'a'
>>> s[-1]
'n'
>>> s[40]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    s[40]
IndexError: string index out of range
>>>
```

```
>>> s[1:40]
'anjan'
>>> s[1:]
'anjan'
>>> s[:4]
'Ranj'
>>> s[:]
'Ranjan'
>>>
>>> s*4
'RanjanRanjanRanjanRanjan'
>>> len(s)
6
>>>
```



Note:

1. In Python the following data types are considered as Fundamental Data types

- **int**
- **float**
- **complex**
- **bool**
- **str**

2. In Python, we can represent char values also by using str type and explicitly char type is not available.

Example:

1) >>> c='a'

2) >>> type(c)

3) <class 'str'>

3. long Data Type is available in Python2 but not in Python3. In Python3 long values also we can represent by using int type only.

4. In Python we can present char Value also by using str Type and explicitly char Type is not available.

Type Casting

We can convert one type value to another type. This conversion is called Typecasting or Type cohesion.

The following are various inbuilt functions for type casting.

- **int()**
- **float()**
- **complex()**
- **bool()**
- **str()**

1. int():

We can use this function to convert values from other types to int

Eg:

```
1)  >>> int(123.987)
2)  123
3)  >>> int(10+5j)
4)  TypeError: can't convert complex to int
5)  >>> int(True)
6)  1
7)  >>> int(False)
8)  0
9)  >>> int("10")
10) 10
11) >>> int("10.5")
12) ValueError: invalid literal for int() with base 10: '10.5'
13) >>> int("ten")
14) ValueError: invalid literal for int() with base 10: 'ten'
15) >>> int("0B1111")
16) ValueError: invalid literal for int() with base 10: '0B1111'
```

Note:

1. We can convert from any type to int except complex type.

2. If we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10.

2. float():

We can use float() function to convert other type values to float type.

We can use float() function to convert other type values to float type.

```
1)  >>> float(10)
2)  10.0
3)  >>> float(10+5j)
4)  TypeError: can't convert complex to float
5)  >>> float(True)
6)  1.0
7)  >>> float(False)
8)  0.0
9)  >>> float("10")
10) 10.0
11) >>> float("10.5")
12) 10.5
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1111")
16) ValueError: could not convert string to float: '0B1111'
```

Note:

1. We can convert any type value to float type except complex type.

2. Whenever we are trying to convert str type to float type compulsary str should be either integral or floating point literal and should be specified only in base-10.

3.complex():

We can use complex() function to convert other types to complex type.

Form-1: complex(x) We can use this function to convert x into complex number with real part x and imaginary part 0.

Eg:

```
1)  complex(10)==>10+0j
2)  complex(10.5)===>10.5+0j
3)  complex(True)==>1+0j
4)  complex(False)==>0j
5)  complex("10")==>10+0j
6)  complex("10.5")==>10.5+0j
7)  complex("ten")
8)      ValueError: complex() arg is a malformed string
```

Form-2: complex(x,y)

We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

Eg: complex(10,-2)==>10-2j

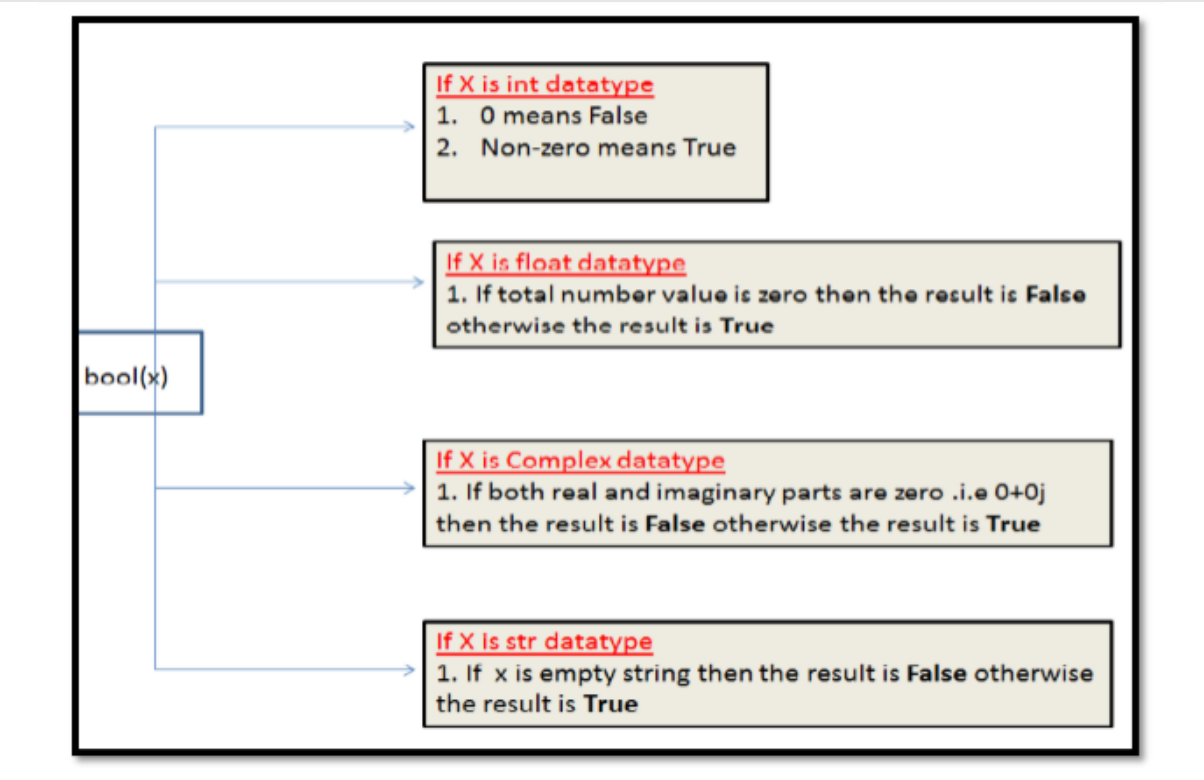
complex(True,False)==>1+0j

4. bool():

We can use this function to convert other type values to bool type.

Eg:

```
1)  bool(0)==>False
2)  bool(1)==>True
3)  bool(10)===>True
4)  bool(10.5)===>True
5)  bool(0.178)==>True
6)  bool(0.0)==>False
7)  bool(10-2j)==>True
8)  bool(0+1.5j)==>True
9)  bool(0+0j)==>False
10) bool("True")==>True
11) bool("False")==>True
12) bool("")==>False
```



5. str():

We can use this method to convert other type values to str type.

Eg:

```
1) >>> str(10)
2) '10'
3) >>> str(10.5)
4) '10.5'
5) >>> str(10+5j)
6) '(10+5j)'
7) >>> str(True)
8) 'True'
```

Fundamental Data Types vs Immutability:

All Fundamental Data types are immutable. i.e once we creates an object,we cannot perform any changes in that object. If we are trying to change then with those changes a new object will be created. This non-chageable behaviour is called immutability.

In Python if a new object is required, then PVM wont create object immediately. First it will check is any object available with the required content or not. If available then existing object will be reused. If it is not available then only a new object will be created. The advantage of this approach is memory utilization and performance will be improved.

But the problem in this approach is, several references pointing to the same object, by using one reference if we are allowed to change the content in the existing object then the remaining references will be effected. To prevent this immutability concept is required. According to this once creates an object we are not allowed to change content. If we are trying to change with those changes a new object will be created.

Eg:

```
1)  >>> a=10
2)  >>> b=10
3)  >>> a is b
4)  True
5)  >>> id(a)
6)  1572353952
7)  >>> id(b)
8)  1572353952
9)  >>>
```

```
>>> a=10
>>> b=10
>>> id(a)
1572353952
>>> id(b)
1572353952
>>> a is b
True
```

```
>>> a=10+5j
>>> b=10+5j
>>> a is b
False
>>> id(a)
15980256
>>> id(b)
15979944
```

```
>>> a=True
>>> b=True
>>> a is b
True
>>> id(a)
1572172624
>>> id(b)
1572172624
```

```
>>> a='durga'
>>> b='durga'
>>> a is b
True
>>> id(a)
16378848
>>> id(b)
16378848
```

bytes Data Type:

bytes data type represens a group of byte numbers just like an array.

Eg:

```
1)  x = [10,20,30,40]
2)      b = bytes(x)
3)      type(b)==>bytes
4)      print(b[0])==> 10
5)      print(b[-1])==> 40
6)      >>> for i in b : print(i)
7)
8)        10
9)        20
10)       30
11)       40
```

Conclusion 1:

The only allowed values for byte data type are 0 to 256. By mistake if we are trying to provide any other values then we will get value error.

Conclusion 2:

Once we creates bytes data type value, we cannot change its values, otherwise we will get TypeError.

Eg:

```
1)  >>> x=[10,20,30,40]
2)  >>> b=bytes(x)
3)  >>> b[0]=100
4)   TypeError: 'bytes' object does not support item assignment
```

bytearray Data type:

bytearray is exactly same as bytes data type except that its elements can be modified.

Eg 1:

```
1)  x=[10,20,30,40]
2)  b = bytearray(x)
3)  for i in b : print(i)
4)  10
5)  20
6)  30
7)  40
8)   b[0]=100
9)   for i in b: print(i)
10) 100
11) 20
12) 30
13) 40
```

Eg 2:

```
1)  >>> x =[10,256]
2)  >>> b = bytearray(x)
3)  ValueError: byte must be in range(0, 256)
```

list data type:

If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.

1. insertion order is preserved

2. heterogeneous objects are allowed

3. duplicates are allowed

4. Growable in nature

5. values should be enclosed within square brackets.

Example:1

```
>>> list=[10,20.5,'Ranjan',True,10]
>>> print(list)
[10, 20.5, 'Ranjan', True, 10]
>>>
```

Example:2

```
1)  list=[10,20,30,40]
2)  >>> list[0]
3)  10
4)  >>> list[-1]
5)  40
6)  >>> list[1:3]
7)  [20, 30]
8)  >>> list[0]=100
9)  >>> for i in list:print(i)
10) ...
11) 100
12) 20
13) 30
```

list is growable in nature. i.e based on our requirement we can increase or decrease the size.

```
>>> list=[100,200,300]
>>> list.append("Ranjan Kumar")
>>> list
[100, 200, 300, 'Ranjan Kumar']
>>> list.remove(300)
>>> list
[100, 200, 'Ranjan Kumar']
>>> list2=list*2
>>> list2
[100, 200, 'Ranjan Kumar', 100, 200, 'Ranjan Kumar']
>>>
```

Note: An ordered, mutable, heterogenous collection of elements is nothing but list, where duplicates also allowed.

tuple data type:

tuple data type is exactly same as list data type except that it is immutable. i.e we cannot chage values.

Tuple elements can be represented within parenthesis.

Eg:

```
1)  t=(10,20,30,40)
2)  type(t)
3)  <class 'tuple'>
4)  t[0]=100
5)  TypeError: 'tuple' object does not support item assignment
6)  >>> t.append("durga")
7)  AttributeError: 'tuple' object has no attribute 'append'
8)  >>> t.remove(10)
9)  AttributeError: 'tuple' object has no attribute 'remove'
```

Note: tuple is the read only version of list

range Data Type:

range Data Type represents a sequence of numbers. The elements present in range Data type are not modifiable. i.e range Data type is immutable.

Form-1: range(10) generate numbers from 0 to 9

Eg: r=range(10) for i in r : print(i) 0 to 9

Form-2: range(10,20)

generate numbers from 10 to 19

r = range(10,20) for i in r : print(i) 10 to 19

Form-3: range(10,20,2)

2 means increment value

r = range(10,20,2) for i in r : print(i) 10,12,14,16,18

We can access elements present in the range Data Type by using index. r=range(10,20) r[0]==>10 r[15]==> IndexError: range object index out of range

We cannot modify the values of range data type

Eg: r[0]=100 TypeError: 'range' object does not support item assignment

Eg:

1) >>> l = list(range(10))

2) >>> l

3) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

We can create a list of values with range data type

set Data Type:

If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type.

- **insertion order is not preserved**
- **duplicates are not allowed**
- **heterogeneous objects are allowed**
- **index concept is not applicable**
- **It is mutable collection 6. Growable in nature**

Example:

```
>>> s={100,0,10,200,10,'Ranjan'}
>>> s
{0, 100, 'Ranjan', 200, 10}
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s[0]
TypeError: 'set' object does not support indexing
>>> s.add(60)
>>> s
{0, 100, 'Ranjan', 200, 10, 60}
>>> s.remove(100)
>>> s
{0, 'Ranjan', 200, 10, 60}
>>>
```

frozenset Data Type:

It is exactly same as set except that it is immutable. Hence we cannot use add or remove functions.

```
1)  >>> s={10,20,30,40}
2)  >>> fs=frozenset(s)
3)  >>> type(fs)
4)  <class 'frozenset'>
5)  >>> fs
6)  frozenset({40, 10, 20, 30})
7)  >>> for i in fs:print(i)
8)  ...
9)  40
10) 10
11) 20
12) 30
13)
14) >>> fs.add(70)
15) AttributeError: 'frozenset' object has no attribute 'add'
16) >>> fs.remove(10)
17) AttributeError: 'frozenset' object has no attribute 'remove'
```

dict Data Type:

If we want to represent a group of values as key-value pairs then we should go for dict data type.

Eg: d={101:'Ranjan',102:'ravi',103:'shiva'}

Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

```
>>> d={101:'Ranjan',102:'Kumar',103:'Shiva'}
>>> d[101]='Anil'
>>> d
{101: 'Anil', 102: 'Kumar', 103: 'Shiva'}
>>> We can create empty dictionary as follows
SyntaxError: invalid syntax
>>> d={}
>>> We can add Key-Value pairs as follows
SyntaxError: invalid syntax
>>> d['a']='Apple'
>>> d['b']='Banana'
>>> print(d)
{'a': 'Apple', 'b': 'Banana'}
>>>
```

Note: dict is mutable and the order wont be preserved.

Note:

- **In general we can use bytes and bytearray data types to represent binary information like images,video files etc**
- **In Python2 long data type is available. But in Python3 it is not available and we can represent long values also by using int type only.**

- **In Python there is no char data type. Hence we can represent char values also by using str type.**

Summary of Datatypes in Python3:

Datatype	Description	Is Immutable	Example
Int	We can use to represent the whole/integral numbers	Immutable	>>> a=10 >>> type(a)
Float	We can use to represent the decimal/floating point numbers	Immutable	>>> b=10.5 >>> type(b)

Complex	We can use to represent the complex numbers	Immutable	>>> c=10+5j
			>>> type(c)
			<class 'complex'>
			>>> c.real
			10.0

Bool	We can use to represent the logical values(Only allowed values are True and False)	Immutable	>>> flag=True
			>>> flag=False
			>>> type(flag)

Str	To represent sequence of Characters	Immutable	>>> s='Ranjan'
			>>> type(s)
			<class 'str'>
			>>> s="Ranjan"
			>>> s='''Ranjan
			Kumar....'''
			>>> type(s)
			<class 'str'>

bytes	To represent a sequence of byte values from 0-255	Immutable	>>> list=[1,2,3,4]
			>>> b=bytes(list)
			>>> type(b)

bytearray	To represent a sequence of byte values from 0-255	Mutable	>>> list=[10,20,30]
			>>> ba=bytearray(list)
			>>> type(ba)

range	To represent a range of values	Immutable	>>> r=range(10) >>> r1=range(0,10)

list	To represent an ordered collection of objects	Mutable	>>> l=[10,11,12,13,14,15] >>> type(l)
tuple	To represent an ordered collections of objects	Immutable	>>> t=(1,2,3,4,5) >>> type(t)
set	To represent an unordered collection of unique objects	Mutable	>>> s={1,2,3,4,5,6} >>> type(s)

			<class 'set'>

frozenset	To represent an unordered collection of unique objects	Immutable	>>> s={11,2,3,'Ranjan',100,'Ramu'}
			>>> fs=frozenset(s)
			>>> type(fs)
			<class 'frozenset'>

dict	To represent a group of key value pairs	Mutable	>>> d={101:'Anil',102:'Bharat',103:'Chiiti'} >>> type(d) <class 'dict'>

None Data Type:

None means Nothing or No value associated. If the value is not available,then to handle such type of cases None introduced. It is something like null value in Java.

Eg:

```
def  m1():
a=10
print(m1())
```

None

The following are various important escape characters in Python

- **\n==>New Line**
- **\t===>Horizontal tab**
- **\r ==>Carriage Return**
- **\b===>Back space**
- **\f===>Form Feed**

- \v==>Vertical tab
- \'===>Single quote
- \"===>Double quote
- \\===>back slash symbol ....

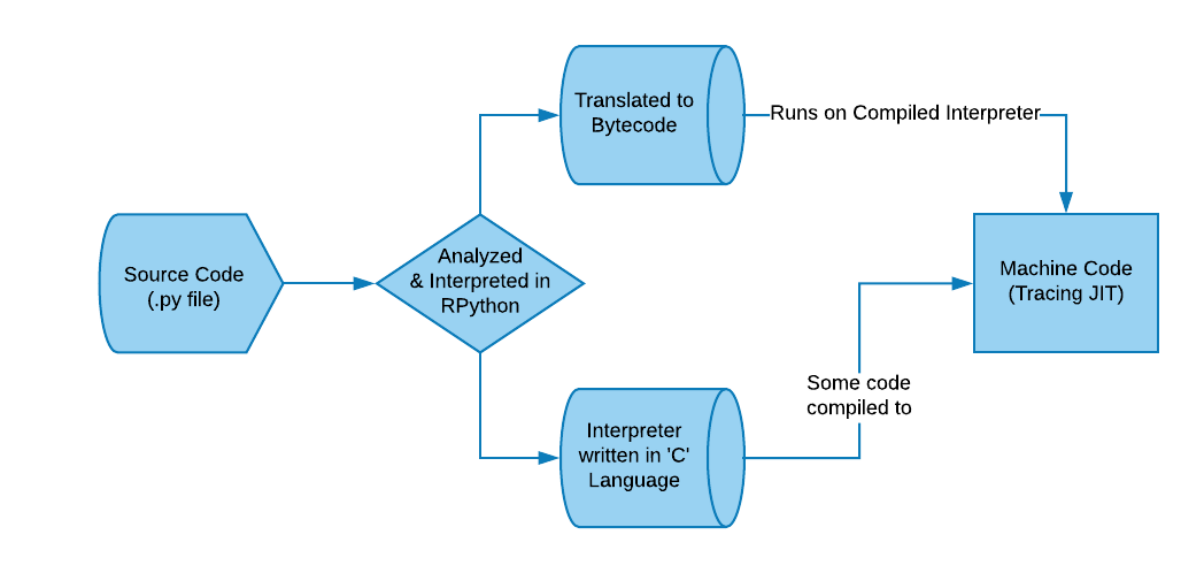
Constants:

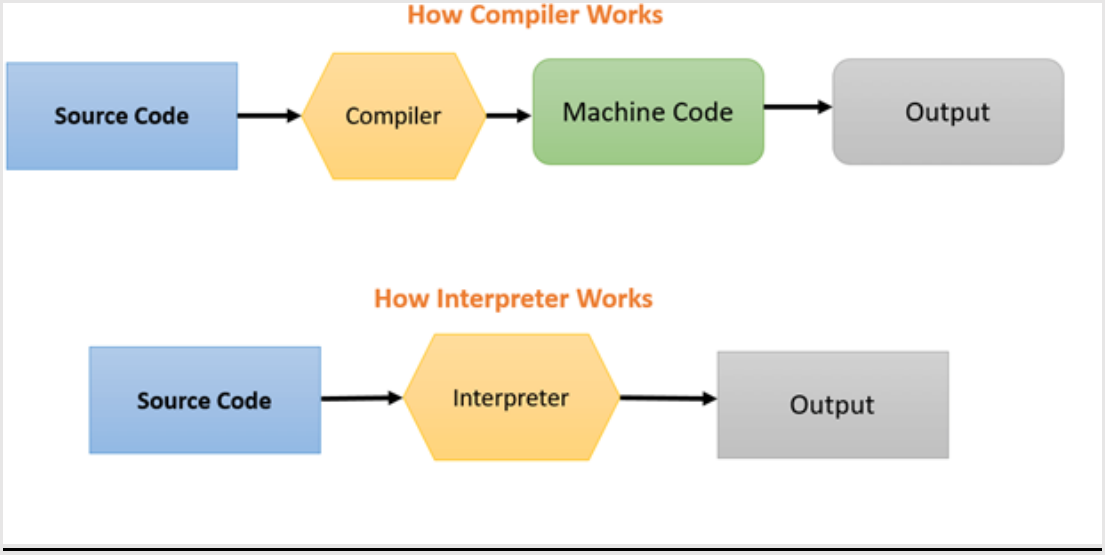
Constants concept is not applicable in Python. But it is convention to use only uppercase characters if we don't want to change value.

MAX_VALUE=10

It is just convention but we can change the value.

Internal Working Of PVM:





Input And Output Statements:

Reading dynamic input from the keyboard:

In Python 2 the following 2 functions are available to read dynamic input from the keyboard.

1. **raw_input()**
2. **input()**

1. raw_input():

This function always reads the data from the keyboard in the form of String Format. We have to convert that string type to our required type by using the corresponding type casting methods.

Eg: x=raw_input("Enter First Number:")

print(type(x)) It will always print str type only for any input type

2. input():

input() function can be used to read data directly in our required format.We are not required to perform type casting.

```
x=input("Enter Value)
type(x)
10 ===> int
"Ranjan"===>str
10.5===>float
True==>bool
```

*****Note:**

But in Python 3 we have only input() method and raw_input() method is not available. Python3 input() function behaviour exactly same as raw_input() method of Python2. i.e every input value is treated as str type only.

raw_input() function of Python 2 is renamed as input() function in Python3

Eg:

```
>>> type(input("Enter value:"))
Enter value:10
<class 'str'>

Enter value:10.5
<class 'str'>

Enter value:True
<class 'str'>
```

Q. Write a program to read 2 numbers from the keyboard and print sum.

```
x=input("Enter First Number:")
y=input("Enter Second Number:")
i = int(x)
j = int(y)
print("The Sum:",i+j)

Enter First Number:100
Enter Second Number:200
The Sum: 300
```

```
x=int(input("Enter First Number:"))
y=int(input("Enter Second Number:"))
print("The Sum:",x+y)
```

```
print("The Sum:",int(input("Enter First Number:"))+int(input("Enter Second Number:")))
```

Q. Write a program to read Employee data from the keyboard and print that data.

```
eno=int(input("Enter Employee No:"))
ename=input("Enter Employee Name:")
esal=float(input("Enter Employee Salary:"))
eaddr=input("Enter Employee Address:")
married=bool(input("Employee Married ?[True|False]:"))
print("Please Confirm Information")
print("Employee No :",eno)
print("Employee Name :",ename)
print("Employee Salary :",esal)
print("Employee Address :",eaddr)
print("Employee Married ? :",married)

```

- **13) D:\Python>py test.py**
- **14) Enter Employee No:100**
- **15) Enter Employee Name: Ranjan**
- **16) Enter Employee Salary:1000**

- 17) Enter Employee Address:Mumbai
- 18) Employee Married ?[True|False]:False
- 19) Please Confirm Information
- 20) Employee No : 100
- 21) Employee Name : Ranjan
- 22) Employee Salary : 1000.0
- 23) Employee Address : Mumbai
- 24) Employee Married ? : False

How to read multiple values from the keyboard in a single line:

```
1)  a,b= [int(x) for x in input("Enter 2 numbers :").split()]
2)  print("Product is :", a*b)
3)
4)  D:\Python_classes>py test.py
5)  Enter 2 numbers :10 20
6)  Product is : 200
```

Note: split() function can take space as seperator by default .But we can pass anything as seperator.

Q. Write a program to read 3 float numbers from the keyboard with , seperator and print their sum.

- a,b,c= [float(x) for x in input("Enter 3 float numbers :").split(',')]
- print("The Sum is :", a+b+c)
- D:\Python>py test.py
- Enter 3 float numbers :10.5,20.6,20.1
- The Sum is : 51.2

eval():

eval Function take a String and evaluate the Result.

Eg: x = eval(“10+20+30”)

print(x)

Output: 60

Eg: x = eval(input(“Enter Expression”))

Enter Expression: 10+2*3/4

Output11.5

eval() can evaluate the Input to list, tuple, set, etc based the provided Input.

Eg: Write a Program to accept list from the keynboard on the display

1) l = eval(input(“Enter List”)) 2) print (type(l)) 3) print(l)

output statements:

We can use print() function to display output.

Form-1: print() without any argument Just it prints new line character

Form-2:
1) print(String):
2) print("Hello World")
3) We can use escape characters also
4) print("Hello \n World")
5) print("Hello\tWorld")
6) We can use repetetion operator (*) in the string
7) print(10*"Hello")
8) print("Hello"*10)
9) We can use + operator also
10) print("Hello"+"World")

Note: If both arguments are String type then + operator acts as concatenation operator.
If one argument is string type and second is any other type like int then we will get Error
If both arguments are number type then + operator acts as arithmetic addition operator.

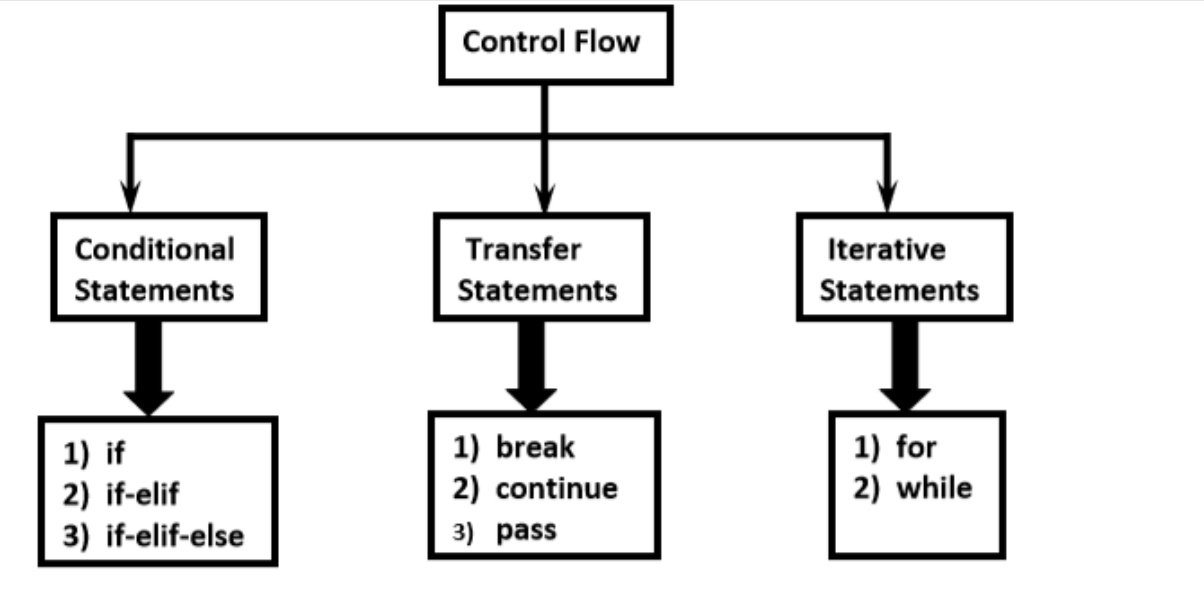
Note:
1) print("Hello"+"World")
2) print("Hello","World")
3)
4) HelloWorld
5) Hello World

Form-3: print() with variable number of arguments:

1. a,b,c=10,20,30
2. print("The Values are :",a,b,c)
3.
4. OutputThe Values are : 10 20 30

Flow Control:

Flow control describes the order in which statements will be executed at runtime.



I. Conditional Statements

1) if

if condition : statement
or
if condition :
statement-1
statement-2
statement-3
If condition is true then statements will be executed.

Eg:

```
1) name=input("Enter Name:")
2) if name=="Ranjan" :
3)     print("Hello Ranjan Good Morning")
4) print("How are you!!!")
5)
6) D:\Python>py test.py
7) Enter Name:Ranjan
8) Hello Ranjan Good Morning
9)
10)
```

11) D:\Python>py test.py
12) Enter Name:Anil
13) How are you!!!

2) if-else:

```
if condition :
        Action-1
else :
        Action-2
```

if condition is true then Action-1 will be executed otherwise Action-2 will be executed.
Eg:

```
1) name=input("Enter Name:")
2) if name=="Ranjan" :
3)     print("Hello Ranjan Good Morning")
4) else:
5)     print("Hello Guest Good Moring")
6) print("How are you!!!")
7)
8) D:\Python>py test.py
9) Enter Name:Ranjan
10) Hello Ranjan Good Morning
11)
12) D:\Python>py test.py
13) Enter Name:Anil
15) Hello Guest Good Moring
16) How are you!!!
```

3) if-elif-else:

Syntax:

```
if condition1:
        Action-1
elif condition2:
        Action-2
elif condition3:
        Action-3
elif condition4:
        Action-4    ...
else:    Default Action
```

Based condition the corresponding action will be executed.

Eg:

```
1) brand=input("Enter Your Favourite Color:")
2) if brand=="Orange" :
3)     print("It is Orange Color")
4) elif brand=="White":
5)     print("It is White Color")
6) elif brand=="Green":
7)     print("It is Green Color")
8) else :
9)     print("Other Colors not in Our Indian Flag")
10)
11)
12) D:\Python>py test.py
13) Enter Your Favourite Brand:Orange
14) It is Orange
15)
16) D:\Python>py test.py
17) Enter Your Favourite Brand:White
18) It is White Color
19)
20) D:\Python >py test.py
21) Enter Your Favourite Brand:Yellow
22) Other Colors not in Our Indian Flag
```

Note:

1. else part is always optional
2. Hence the following are various possible syntaxes.
   a. if b. if - else c. if-elif-else d.if-elif
3. There is no switch statement in Python

II. Iterative Statements

If we want to execute a group of statements multiple times then we should go for Iterative statements.

Python supports 2 types of iterative statements.

1. for loop

2. while loop

1) for loop:

If we want to execute some action for every element present in some sequence(it may be string or collection)then we should go for for loop.

Syntax:

for x in sequence : body

where sequence can be string or any collection. Body will be executed for every element present in the sequence.

Eg 1: To print characters present in string index wise:

- **1)s=input("Enter some String: ")**
- **2)i=0**
- **3)for x in s :**
- **4) print("The character present at ",i,"index is :",x)**
- **5) i=i+1**
- **6)**
- **7)**
- **8) D:\Python>py test.py**
- **9) Enter some String: Ranjan Kumar**
- **10) The character present at 0 index is : R**
- **11) The character present at 1 index is : a**
- **12) The character present at 2 index is : n**
- **13) The character present at 3 index is : j**
- **14) The character present at 4 index is : a**
- **15) The character present at 5 index is : n**
- **16) The character present at 6 index is :**
- **17) The character present at 7 index is : K**
- **18) The character present at 8 index is : u**
- **19) The character present at 9 index is : m**
- **20) The character present at 10 index is : a**
- **19) The character present at 11 index is : r**

2) while loop:

If we want to execute a group of statements iteratively until some condition false,then we should go for while loop.

Syntax:

while condition :

body

Eg: To print numbers from 1 to 10 by using while loop

```
1)      x=1
2)     while x <=10:
3)    print(x)
4)    x=x+1
```

Eg: To display the sum of first n numbers

```
1) n=int(input("Enter number:"))
2) sum=0
3) i=1
4) while i<=n:
5)    sum=sum+i
6)    i=i+1
7) print("The sum of first",n,"numbers is :",sum)
```

III. Transfer Statements

1) break:

We can use break statement inside loops to break loop execution based on some condition. Eg:

- **1)for i in range(10):**
- **2) if i==7:**
- **3) print("processing is enough..plz break")**
- **4) break**
- **5) print(i)**
- **6) D:\Python>py test.py**
- **7) 0**
- **8) 1**
- **9)2**
- **10) 3**
- **11) 4**
- **12) 5**
- **13) 6**
- **14) processing is enough..plz break**

2) continue:

We can use continue statement to skip current iteration and continue next iteration.

Eg 1: To print odd numbers in the range 0 to 9

```
1) for i in range(10):
2)     if i%2==0:
```

```
3)      continue
4)    print(i)
5)
6) D:\Python>py test.py
7) 1
8) 3
9) 5
10)7
11) 9
```

3) pass statement:

pass is a keyword in Python.

In our programming syntactically if block is required which won't do anything then we can define that empty block with pass keyword.

pass

|- It is an empty statement

|- It is null statement

|- It won't do anything

Eg:

```
1)for i in range(100):
2)    if i%9==0:
3)        print(i)
4)    else:pass
5)
6)D:\Python>py test.py
7)0
8)9
9)18
10)27
11)36
11)45
12)54
13)63
14)72
15)81
16)90
17)99
```

del statement:

del is a keyword in Python.

After using a variable, it is highly recommended to delete that variable if it is no longer required,so that the corresponding object is eligible for Garbage Collection. We can delete variable by using del keyword.

Eg:
1) x=10
2) print(x)
3) del x

After deleting a variable we cannot access that variable otherwise we will get NameError.

Eg:
1) x=10
2) del x
3) print(x)

NameError: name 'x' is not defined.

Note: We can delete variables which are pointing to immutable objects.But we cannot delete the elements present inside immutable object.

Eg:

1) s="Ranjan"
2) print(s)
3) del s==>valid
4) del s[0] ==>TypeError: 'str' object doesn't support item deletion

Difference between del and None:

In the case del, the variable will be removed and we cannot access that variable(unbind operation)

1) s="Ranjan"
2) del s
3) print(s) ==>NameError: name 's' is not defined.

But in the case of None assignment the variable won't be removed but the corresponding object is eligible for Garbage Collection(re bind operation). Hence after assigning with None value,we can access that variable.

1) s="Ranjan"
2) s=None
3) print(s) # None

Data Structure In Python

- **List Data Structure:**
- **Tuple Data Structure:**
- **Set Data Structure:**
- **Dictionary Data Structure:**

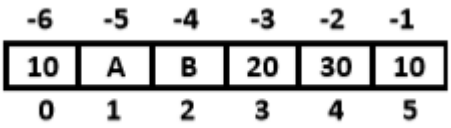
1) List Data Structure:

If we want to represent a group of individual objects as a single entity where insertion order preserved and duplicates are allowed, then we should go for List.

insertion order preserved. duplicate objects are allowed heterogeneous objects are allowed. List is dynamic because based on our requirement we can increase the size and decrease the size. In List the elements will be placed within square brackets and with comma seperator.

We can differentiate duplicate elements by using index and we can preserve insertion order by using index. Hence index will play very important role. Python supports both positive and negative indexes. +ve index means from left to right where as negative index means right to left

[10,"A","B",20, 30, 10]



List objects are mutable.i.e we can change the content.

Creation of List Objects:

1. **We can create empty list object as follows...**

```
1) list=[]
2) print(list)
3) print(type(list))
4)
5) []
6) <class 'list'>
```

2. **If we know elements already then we can create list as follows.**
   list=[10,20,30,40]

3. With dynamic input:

```
1) list=eval(input("Enter List:"))
2) print(list)
3) print(type(list))
4)
5)  D:\Python >py test.py
6) Enter List:[10,20,30,40]
7) [10, 20, 30, 40]
8) <class 'list'>
```

4. With list() function:

```
1) l=list(range(0,10,2))
2) print(l)
3) print(type(l))
4)
5) D:\Python>py test.py
6) [0, 2, 4, 6, 8]
7) <class 'list'>
```

Eg:

```
1) s="Ranjan"
2) l=list(s)
3) print(l)
4)
5) D:\Python >py test.py
6) ['R', 'a', 'n', 'j', 'a', 'n']
```

5. with split() function:

```
1) s="Learning Python is very very easy !!!"
2) l=s.split()
3) print(l)
4) print(type(l))
5)
6) D:\Python >py test.py
7) ['Learning', 'Python', 'is', 'very', 'very', 'easy', '!!!']
8) <class 'list'>
```

Note:

Sometimes we can take list inside another list, such type of lists are called nested lists. [10,20,[30,40]].

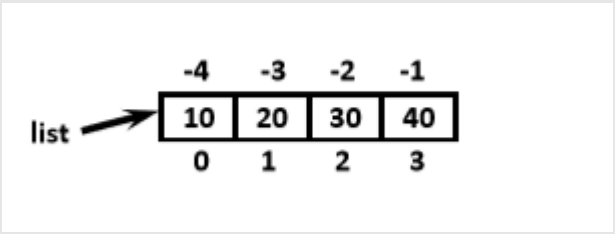
Accessing elements of List:

We can access elements of the list either by using index or by using slice operator(:)

1. By using index:

List follows zero based index. ie index of first element is zero. List supports both +ve and -ve indexes. +ve index meant for Left to Right -ve index meant for Right to Left.

list=[10,20,30,40]



print(list[0]) ==>10
print(list[-1]) ==>40
print(list[10]) ==>IndexError: list index out of range

2. By using slice operator:

Syntax:
list2= list1[start:stop:step]

start ==>it indicates the index where slice has to start
default value is 0

stop ===>It indicates the index where slice has to end
default value is max allowed index of list ie length of the list

step ==>increment value
default value is 1

Eg:

```
1) n=[1,2,3,4,5,6,7,8,9,10]
2) print(n[2:7:2])
3) print(n[4::2])
4) print(n[3:7])
5) print(n[8:2:-2])
6) print(n[4:100])
7)
```

8) Output
9) D:\Python >py test.py
10) [3, 5, 7]
11) [5, 7, 9]
12) [4, 5, 6, 7]
13) [9, 7, 5]
14) [5, 6, 7, 8, 9, 10]

List vs mutability:

Once we creates a List object, we can modify its content. Hence List objects are mutable.

Eg:

```
1) n=[10,20,30,40]
2) print(n)
3) n[1]=777
4) print(n)
5)
6) D:\Python >py test.py
7) [10, 20, 30, 40]
8) [10, 777, 30, 40]
```

Important functions of List:

I. To get information about list:

1. len():
   returns the number of elements present in the list
   Eg: n=[10,20,30,40]
   print(len(n))==>4

2. count():
   It returns the number of occurrences of specified item in the list

```
1) n=[1,2,2,2,2,3,3]
2) print(n.count(1))
3) print(n.count(2))
4) print(n.count(3))
5) print(n.count(4))
6)
7) Output:
8) D:\Python>py test.py
9) 1
10) 4
```

11) 2
12) 0

3. index() function:

returns the index of first occurrence of the specified item.

Eg:
1) n=[1,2,2,2,2,3,3]
2) print(n.index(1)) ==>0
3) print(n.index(2)) ==>1
4) print(n.index(3)) ==>5
5) print(n.index(4)) ==>ValueError: 4 is not in list

Note: If the specified element not present in the list then we will get ValueError.Hence before index() method we have to check whether item present in the list or not by using in operator.

print(4 in n)==>False

II. Manipulating elements of List:

1. append() function:

We can use append() function to add item at the end of the list.

Eg: To add all elements to list upto 100 which are divisible by 10

```
1) list=[]
2) for i in range(101):
3)     if i%10==0:
4)         list.append(i)
5) print(list)
6)
7)
8) D:\Python >py test.py
9) [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

2. insert() function:

To insert item at specified index position

Ex:

```
1) n=[1,2,3,4,5]
2) n.insert(1,888)
3) print(n)
4)
5) D:\Python>py test.py
6) [1, 888, 2, 3, 4, 5]
```

Note:

If the specified index is greater than max index then element will be inserted at last position. If the specified index is smaller than min index then element will be inserted at first position.

Differences between append() and insert()

append()	insert()
In List when we add any element it will come in last i.e. it will be last element.	In List we can insert any element in particular index number

3. extend() function:

To add all items of one list to another list

l1.extend(l2)
all items present in l2 will be added to l1

4. remove() function:

We can use this function to remove specified item from the list.If the item present multiple times then only first occurrence will be removed.

5. pop() function:

It removes and returns the last element of the list. This is only function which manipulates list and returns some element.

Note: If the list is empty then pop() function raises IndexError

Eg:
1) n=[]
2) print(n.pop()) ==> IndexError: pop from empty list

Note:

1. pop() is the only function which manipulates the list and returns some value
2. In general we can use append() and pop() functions to implement stack datastructure by using list,which follows LIFO(Last In First Out) order.

In general we can use pop() function to remove last element of the list. But we can use to remove elements based on index.

n.pop(index)===>To remove and return element present at specified index.
n.pop()==>To remove and return last element of the list

Ex:

1) n=[10,20,30,40,50,60]
2) print(n.pop()) #60
3) print(n.pop(1)) #20
4) print(n.pop(10)) ==>IndexError: pop index out of range

Differences between remove() and pop()

remove()	pop()
1) We can use to remove special element from the List.	1) We can use to remove last element from the List.
2) It can't return any value.	2) It returned removed element.
3) If special element not available then we get VALUE ERROR.	3) If List is empty then we get Error.

Note:
List objects are dynamic. i.e based on our requirement we can increase and decrease the size.

append(),insert() ,extend() ===>for increasing the size/growable nature
remove(), pop() ======>for decreasing the size /shrinking nature

III. Ordering elements of List:

1. **reverse():**
   We can use to reverse() order of elements of list.

Ex:

```
1) n=[10,20,30,40]
2) n.reverse()
3) print(n)
4)
5) D:\Python >py test.py
6) [40, 30, 20, 10]
```

2. sort() function:

In list by default insertion order is preserved. If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

For numbers ==>default natural sorting order is Ascending Order For Strings ==> default natural sorting order is Alphabetical Order

```
1) n=[20,5,15,10,0]
2) n.sort()
3) print(n)       #[0,5,10,15,20]
4)
5) s=["Dog","Banana","Cat","Apple"]
6) s.sort()
7) print(s)   #['Apple','Banana','Cat','Dog']
```

Note: To use sort() function, compulsory list should contain only homogeneous elements. otherwise we will get TypeError

Eg:

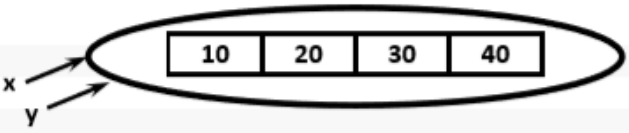
```
1) n=[20,10,"A","B"]
2) n.sort()
3) print(n)
4)
5)  TypeError: '<' not supported between instances of 'str' and 'int'
```

Aliasing and Cloning of List objects:

The process of giving another reference variable to the existing list is called aliasing.

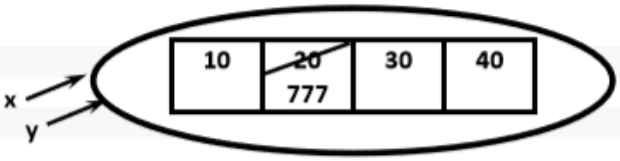
Eg:

```
1) x=[10,20,30,40]
2) y=x
3) print(id(x))
4) print(id(y))
```



The problem in this approach is by using one reference variable if we are changing content,then those changes will be reflected to the other reference variable.

```
1) x=[10,20,30,40]
2) y=x
3) y[1]=777
4) print(x) ==>[10,777,30,40]
```

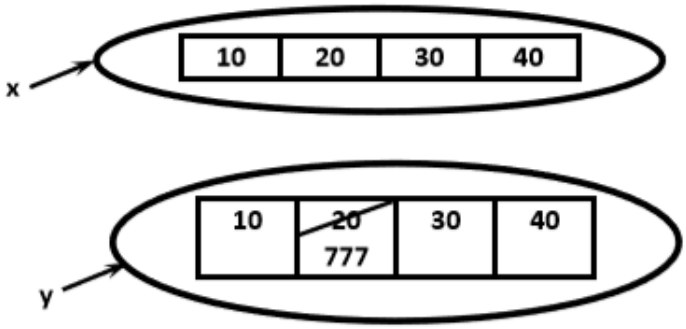


To overcome this problem we should go for cloning. The process of creating exactly duplicate independent object is called cloning.

We can implement cloning by using slice operator or by using copy() function.

1. **<u>By using slice operator:</u>**

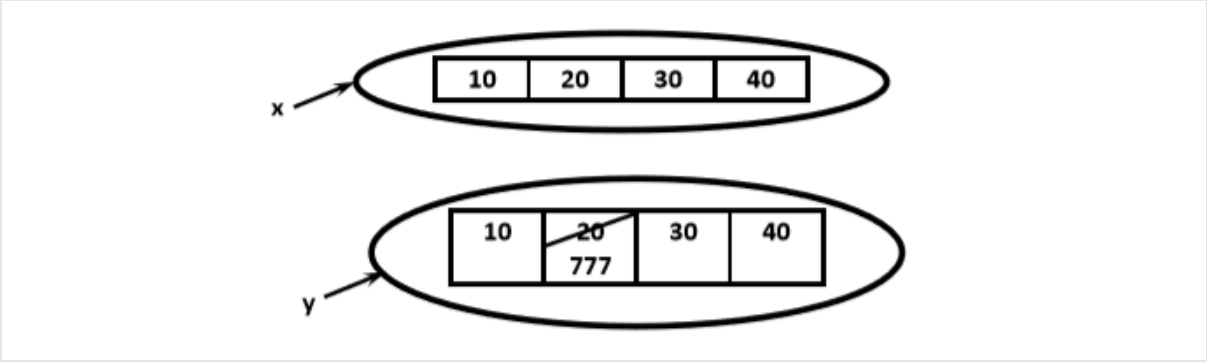
- **x=[10,20,30,40]**
- **y=x[:]**
- **y[1]=777**
- **print(x) ==>[10,20,30,40]**
- **print(y) ==>[10,777,30,40]**



2. **<u>By using copy() function:</u>**

1) x=[10,20,30,40]
2) y=x.copy()
3) y[1]=777
4) print(x) ==>[10,20,30,40]

5) print(y) ==>[10,777,30,40]



Q. Difference between = operator and copy() function

= operator meant for aliasing
copy() function meant for cloning

List Comprehensions:

It is very easy and compact way of creating list objects from any iterable objects(like list,tuple,dictionary,range etc) based on some condition.

Syntax: list=[expression for item in list if condition]

Eg:

```
1) s=[ x*x for x in range(1,11)]
2) print(s)
3) v=[2**x  for x in range(1,6)]
4) print(v)
5) m=[x for x in s if x%2==0]
6) print(m)
7)
8) Output
9) D:\Python>py test.py
10) [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
11) [2, 4, 8, 16, 32]
12) [4, 16, 36, 64, 100
```

2. Set Data Structure:

- **If we want to represent a group of unique values as a single entity then we should go for set.**
- **Duplicates are not allowed.**
- **Insertion order is not preserved.But we can sort the elements.**
- **Indexing and slicing not allowed for the set.**

- Heterogeneous elements are allowed.
- Set objects are mutable i.e once we creates set object we can perform any changes in that object based on our requirement.
- We can represent set elements within curly braces and with comma seperation
- We can apply mathematical operations like union,intersection,difference etc on set objects.

Creation of Set objects:

Eg:

```
1. s={10,20,30,40}
2. print(s)
3. print(type(s))
4.
5. Output
6. {40, 10, 20, 30}
7. <class 'set'>
```

We can create set objects by using set() function:

s=set(any sequence)

Eg 1:

```
1. l = [10,20,30,40,10,20,10]
2. s=set(l)
3. print(s) # {40, 10, 20, 30}
```

Eg 2:

```
1. s=set(range(5))
2. print(s) #{0, 1, 2, 3, 4}
```

Note: While creating empty set we have to take special care. Compulsory we should use set() function.

s={} ==>It is treated as dictionary but not empty set.

Important functions of set:

1. add(x):

Adds item x to the set

Eg:

1. s={10,20,30}

2. s.add(40)
3. print(s) #{40, 10, 20, 30}

2. update(x,y,z):

To add multiple items to the set. Arguments are not individual elements and these are Iterable objects like List,range etc. All elements present in the given Iterable objects will be added to the set.

Eg:

```
1. s={10,20,30}
2. l=[40,50,60,10]
3. s.update(l,range(5))
4. print(s)
5.
6. Output
7. {0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}
```

3. copy():

Returns copy of the set. It is cloned object.
s={10,20,30} s1=s.copy() print(s1)

4. pop():

It removes and returns some random element from the set.

Eg:

```
1. s={40,10,30,20}
2. print(s)
3. print(s.pop())
4. print(s)
5.
6. Output
7. {40, 10, 20, 30}
8. 40
9. {10, 20, 30}
```

5. remove(x):

It removes specified element from the set. If the specified element not present in the Set then we will get KeyError

s={40,10,30,20} s.remove(30) print(s) # {40, 10, 20} s.remove(50) ==>KeyError: 50

6. discard(x):

It removes the specified element from the set. If the specified element not present in the set then we won't get any error.

```
s={10,20,30}
s.discard(10)
print(s) ===>{20, 30}
s.discard(50)
print(s) ==>{20, 30}
```

7.clear():

To remove all elements from the Set.

Ex:

```
1. s={10,20,30}
2. print(s)
3. s.clear()
4. print(s)
5.
6. Output
7. {10, 20, 30}
8. set()
```

Mathematical operations on the Set:

1.union():

x.union(y) ==>We can use this function to return all elements present in both sets

x.union(y) or x|y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y)) #{10, 20, 30, 40, 50, 60}
```

print(x|y) #{10, 20, 30, 40, 50, 60}

2. intersection():

x.intersection(y) or x&y
Returns common elements present in both x and y

Eg:
x={10,20,30,40}
y={30,40,50,60}
print(x.intersection(y)) #{40, 30} print(x&y) #{40, 30}

3. difference():

x.difference(y) or x-y returns the elements present in x but not in y

Eg:
x={10,20,30,40}
y={30,40,50,60}
print(x.difference(y)) #{10, 20}
print(x-y) #{10, 20}
print(y-x) #{50, 60}
4.symmetric_difference():

x.symmetric_difference(y) or x^y

Returns elements present in either x or y but not in both

Eg:

x={10,20,30,40}
y={30,40,50,60}
print(x.symmetric_difference(y)) #{10, 50, 20, 60}
print(x^y) #{10, 50, 20, 60}

Membership operators: (in , not in)

Eg:

1. s=set("Ranjan")
2. print(s)
3. print('R' in s)
4. print('z' in s)
5.

6. Output
7. {'a', 'R', 'n', 'j', 'a', 'n'}
8. True
9. False

Set Comprehension:

Set comprehension is possible.

Ex1:

```
s={x*x  for x in range(5)}
print(s) #{0, 1, 4, 9, 16}
```

Ex2:

```
s={2**x  for x in range(2,10,2)}
print(s)    #{16, 256, 64, 4}
```

Note: Set objects won't support indexing and slicing:

```
Eg: s={10,20,30,40}
print(s[0])  ==>TypeError: 'set' object does not support indexing
print(s[1:3])  ==>TypeError: 'set' object is not subscriptable
```

Assignment:

Q. Write a program to eliminate duplicates present in the list?
Q. Write a program to print different vowels present in the given word?

3.Tuple Data Structure:

1. Tuple is exactly same as List except that it is immutable. i.e once we creates Tuple object, we cannot perform any changes in that object. Hence Tuple is Read Only version of List.
2. If our data is fixed and never changes then we should go for Tuple.
3. Insertion Order is preserved
4. Duplicates are allowed
5. Heterogeneous objects are allowed.
6. We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also. Tuple support both +ve and -ve index. +ve index means forward direction(from left to right) and -ve index means backward direction(from right to left)
7. We can represent Tuple elements within Parenthesis and with comma seperator. Parenethesis are optional but recommended to use.

Eg:

```
1. t=10,20,30,40
2. print(t)
3. print(type(t))
4.
5. Output
6. (10, 20, 30, 40)
7. <class 'tuple'>
8.
9.  t=()
10. print(type(t)) # tuple
```

Note: **We have to take special care about single valued tuple. compulsary the value should ends with comma, otherwise it is not treated as tuple.**

Eg:

```
1. t=(10)
2. print(t)
3. print(type(t))
4.
5. Output
6. 10
7. <class 'int'>
```

Eg:

```
1. t=(10,)
2. print(t)
3. print(type(t))
4.
5. Output
6. (10,)
7. <class 'tuple'>
```

Q. Which of the following are valid tuples?

1. t=()
2. t=10,20,30,40
3. t=10
4. t=10,
5. t=(10)
6. t=(10,)
7. t=(10,20,30,40)

Tuple creation:

1. **t=()**
   creation of empty tuple

2. t=(10,)

t=10,

creation of single valued tuple ,parenthesis are optional,should ends with comma

3. t=10,20,30

t=(10,20,30)

creation of multi values tuples & parenthesis are optional

4. By using tuple() function:

```
list=[10,20,30]
    t=tuple(list)
    print(t)

    t=tuple(range(10,20,2))
    print(t)
```

Accessing elements of tuple:

We can access either by index or by slice operator

1. By using index:

```
t=(10,20,30,40,50,60)
print(t[0]) #10
print(t[-1])  #60
print(t[100])    IndexError: tuple index out of range
```

2. By using slice operator:

```
t=(10,20,30,40,50,60)
print(t[2:5])
print(t[2:100])
print(t[::2])

Output
(30, 40, 50)
(30, 40, 50, 60)
(10, 30, 50)
```

Tuple vs immutability:

Once we creates tuple,we cannot change its content. Hence tuple objects are immutable.

Eg: t=(10,20,30,40)

t[1]=70 TypeError: 'tuple' object does not support item assignment

Mathematical operators for tuple:

We can apply + and * operators for tuple.

1. **Concatenation Operator(+):**

```
t1=(10,20,30)
t2=(40,50,60)
t3=t1+t2
print(t3)  # (10,20,30,40,50,60)
```

2. **Multiplication operator or repetition operator(*):**

```
t1=(10,20,30)
t2=t1*3
print(t2) #(10,20,30,10,20,30,10,20,30)
```

Important functions of Tuple:

1. **len():**
   To return number of elements present in the tuple
   Eg: t=(10,20,30,40)
   print(len(t)) #4

2. **count():**
   To return number of occurrences of given element in the tuple
   Eg: t=(10,20,10,10,20)
   print(t.count(10)) #3

3. **index():**
   returns index of first occurrence of the given element.
   If the specified element is not available then we will get ValueError.

   Eg: t=(10,20,10,10,20)
   print(t.index(10)) #0
   print(t.index(30)) ValueError: tuple.index(x): x not in tuple

4. **sorted():**
   To sort elements based on default natural sorting order.

```
t=(40,10,30,20)
t1=sorted(t)
print(t1)
print(t)

 Output
 [10, 20, 30, 40]
(40, 10, 30, 20)
```

We can sort according to reverse of default natural sorting order as follows.

Ex:

t1=sorted(t,reverse=True)
print(t1) [40, 30, 20, 10]

5. min() and max() functions:

These functions return min and max values according to default natural sorting order.

Eg:

```
t=(40,10,30,20)
print(min(t))  #10
print(max(t))  #40
```

6. cmp():

It compares the elements of both tuples. If both tuples are equal then returns 0 If the first tuple is less than second tuple then it returns -1 If the first tuple is greater than second tuple then it returns +1

Eg:

```
t1=(10,20,30)
t2=(40,50,60)
t3=(10,20,30)
print(cmp(t1,t2)) # -1
print(cmp(t1,t3)) # 0
print(cmp(t2,t3)) # +1
```

Note: cmp() function is available only in Python2 but not in Python 3

Tuple Comprehension:

Tuple Comprehension is not supported by Python.

```
t= ( x**2  for x in range(1,6))
```

Here we are not getting tuple object and we are getting generator object.

Ex:

```
1. t= ( x**2  for x in range(1,6))
2. print(type(t))
3. for x in t:
4.     print(x)
5.
6. Output
7. D:\Python>py test.py
8. <class 'generator'>
9. 1
10. 4
11. 9
12. 16
13. 25
```

Differences between List and Tuple:

List and Tuple are exactly same except small difference: List objects are mutable where as Tuple objects are immutable.

In both cases insertion order is preserved, duplicate objects are allowed, heterogenous objects are allowed, index and slicing are supported.

List	Tuple
1) List is a Group of Comma separeated Values within Square Brackets and Square Brackets are mandatory. Eg: i = [10, 20, 30, 40]	1) Tuple is a Group of Comma separeated Values within Parenthesis and Parenthesis are optional. Eg: t = (10, 20, 30, 40) t = 10, 20, 30, 40
2) List Objects are Mutable i.e. once we creates List Object we can perform any changes in that Object. Eg: i[1] = 70	2) Tuple Objeccts are Immutable i.e. once we creates Tuple Object we cannot change its content. t[1] = 70 ➔ ValueError: tuple object does not support item assignment.
3) If the Content is not fixed and keep on changing then we should go for List.	3) If the content is fixed and never changes then we should go for Tuple.
4) List Objects can not used as Keys for Dictionries because Keys should be Hashable and Immutable.	4) Tuple Objects can be used as Keys for Dictionries because Keys should be Hashable and Immutable.

Assignment:

Q. Write a program to take a tuple of numbers from the keyboard and print its sum and average?

4. Dictionary Data Structure:

We can use List, Tuple and Set to represent a group of individual objects as a single entity.

If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

Eg: rollno---name
phone number--address
ipaddress---domain name

Duplicate keys are not allowed but values can be duplicated. Hetrogeneous objects are allowed for both key and values. insertion order is not preserved Dictionaries are mutable Dictionaries are dynamic indexing and slicing concepts are not applicable

Note: In C++ and Java Dictionaries are known as "Map" where as in Perl and Ruby it is known as "Hash"

How to create Dictionary?

d={} or d=dict()

we are creating empty dictionary. We can add entries as follows

```
d[100]="Ranjan"
d[200]="Anil"
d[300]="Bhrat"
print(d)   #{100: 'Ranjan', 200: 'Anil', 300: 'Bhrat'}
```

If we know data in advance then we can create dictionary as follows

```
d={100:'Ranjan' ,200:'Anil', 300:'Bhrat'}
```

```
d={key:value, key:value}
```

How to access data from the dictionary?

We can access data by using keys.

Ex:

```
d={100:'Ranjan' ,200:'Anil', 300:'shiva'}
print(d[100])    #Ranjan
print(d[300])    #shiva
```

If the specified key is not available then we will get KeyError

```
print(d[400])  # KeyError: 400
```

We can prevent this by checking whether key is already available or not by using has_key() function or by using in operator.

d.has_key(400) ==> returns 1 if key is available otherwise returns 0

Note: But has_key() function is available only in Python 2 but not in Python 3. Hence compulsory we have to use in operator.

How to update dictionaries?

d[key]=value

If the key is not available then a new entry will be added to the dictionary with the specified key-value pair

If the key is already available then old value will be replaced with new value.

Eg:

1. d={100:"Ranjan",200:"Anil",300:"Bhrat"}

```
2. print(d)
3. d[400]="Fahad"
4. print(d)
5. d[100]="Kumar"
6. print(d)
7.
8. Output
9. {100:"Ranjan",200:"Anil",300:"Bhrat"}
10. {100:"Ranjan",200:"Anil",300:"Bhrat", 400: 'Fahad'}
11. {100: 'Kumar', 200: 'Anil', 300: 'Bhrat', 400: 'Fahad'}
```

How to delete elements from dictionary?

del d[key]:

It deletes entry associated with the specified key.
If the key is not available then we will get KeyError.

Eg:

```
1. d={100:"Ranjan",200:"Fahad",300:"JP"}
2. print(d)
3. del d[100]
4. print(d)
5. del d[400]
6.
7. Output
8. {100: 'Ranjan', 200: 'Fahad', 300: 'JP'}
9. {200: 'Fahad', 300: 'JP'}
10. KeyError: 400
```

d.clear():

To remove all entries from the dictionary

Eg:

```
1.  d={100:"Ranjan",200:"Fahad",300:"JP"}
2. print(d)
3. d.clear()
4. print(d)
5.
6. Output
7. {100: 'Ranjan', 200: 'Fahad', 300: 'JP'}
8. {}
```

del d:

To delete total dictionary.Now we cannot access d

Eg:
1. d={100:"Ranjan",200:"Fahad",300:"JP"}
2. print(d)
3. del d
4. print(d)
5.
6. Output
7.{100:"Ranjan",200:"Fahad",300:"JP"}
8. NameError: name 'd' is not defined

Important functions of dictionary:

1. dict():
To create a dictionary

d=dict() ===>It creates empty dictionary
d=dict({100:"Ranjan",200:"Kumar"}) ==>It creates dictionary with specified elements
d=dict([(100,"Anil"),(200,"Bhrat"),(300,"Chintu")])==>It creates dictionary with the given list of tuple elements

2. len():
Returns the number of items in the dictionary

3. clear():
To remove all elements from the dictionary

4. get():
To get the value associated with the key

d.get(key)

If the key is available then returns the corresponding value otherwise returns None.
It wont raise any error.

d.get(key,defaultvalue)

If the key is available then returns the corresponding value otherwise returns default value.

Eg:

```
d={100:"Ranjan",200:"Fahad",300:"JP"}
print(d[100]) ==>Ranjan
print(d[400]) ==>KeyError:400
print(d.get(100)) ==Ranjan
print(d.get(400)) ==>None
print(d.get(100,"Guest")) ==Ranjan
print(d.get(400,"Guest")) ==>Guest
```

3. pop():

d.pop(key)

It removes the entry associated with the specified key and returns the corresponding value
If the specified key is not available then we will get KeyError.

Ex:

```
1) d={100:"Ranjan",200:"Anil",300:"Bhrat"}
2) print(d.pop(100))
3) print(d)
4) print(d.pop(400))
5)
6) Output
7) Ranjan
8) {200: 'Anil', 300: 'Bhrat'}
9) KeyError: 400
```

4. popitem():

It removes an arbitrary item(key-value) from the dictionaty and returns it.

5. keys():

It returns all keys associated eith dictionary.

6. values():

It returns all values associated with the dictionary.

7. items():

It returns list of tuples representing key-value pairs.

[(k,v),(k,v),(k,v)]

8. copy():

To create exactly duplicate dictionary(cloned copy)

d1=d.copy();

9. setdefault():

d.setdefault(k,v)
If the key is already available then this function returns the corresponding value.
If the key is not available then the specified key-value will be added as new item to the dictionary.

10. update():

d.update(x) All items present in the dictionary x will be added to dictionary d v.

Dictionary Comprehension:

Comprehension concept applicable for dictionaries also.

```
1. squares={x:x*x for x in range(1,6)}
2. print(squares)
3. doubles={x:2*x for x in range(1,6)}
4. print(doubles)
5.
6. Output
7. {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
8. {1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Assignment:

Q. Write a program to take dictionary from the keyboard and print the sum of values?
Q. Write a program to find number of occurrences of each letter present in the given string?

Q. Write a program to find number of occurrences of each vowel present in the given string?

Q. Write a program to accept student name and marks from the keyboard and creates a dictionary. Also display student marks by taking student name as input?

Function:

If a group of statements is repeatedly required then it is not recommended to write these statements everytime seperately. We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

The main advantage of functions is code Reusability.

Note: In other languages functions are known as methods, procedures, subroutines etc.

Python supports 2 types of functions:

1. Built in Functions

2. User Defined Functions

1. Built in Functions:

The functions which are coming along with Python software automatically, are called built in functions or pre defined functions

Eg:

- **id()**
- **type()**
- **input()**
- **eval()**
- **etc..**

2. User Defined Functions:

The functions which are developed by programmer explicitly according to business requirements ,are called user defined functions.

Syntax to create user defined functions:

```
def function_name(parameters) :
        """ doc string"""
          ----    -----
 return value
```

Note:

While creating functions we can use 2 keywords

1. def (mandatory) 2. return (optional)

Eg 1: Write a function to print Hello

test.py:

```
1) def wish():
2)     print("Hello Good Morning")
3) wish()
4) wish()
5) wish()
```

Return Statement:

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

Ex:

Q. Write a function to accept 2 numbers as input and return sum.

```
1. def add(x,y):
2.     return x+y
3. result=add(10,20)
4. print("The sum is",result)
5. print("The sum is",add(100,200))
6.
```

Output:

- **The sum is 30**
- **The sum is 300**

If we are not writing return statement then default return value is None.
Ex:

```
1. def  f1():
2.     print("Hello")
3. f1()
4. print(f1())
5.
6. Output
7. Hello
8. Hello
9. None
```

Returning multiple values from a function:

In other languages like C,C++ and Java, function can return atmost one value. But in Python, a function can return any number of values.

Ex:

```
1)  def sum_sub(a,b):
2)      sum=a+b
3)      sub=a-b
4)      return sum,sub
5)  x,y=sum_sub(100,50)
6)  print("The Sum is :",x)
7)  print("The Subtraction is :",y)
8)
9)  Output
10) The Sum is : 150
11) The Subtraction is : 50
```

Types of Variables

Python supports 2 types of variables.

1. Global Variables
2. Local Variables

1. Global Variables:

The variables which are declared outside of function are called global variables. These variables can be accessed in all functions of that module.

Ex:

```
1)  a=10  # global variable
2)  def f1():
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
8)  f1()
9)  f2()
10)
11) Output
12) 10
13) 10
```

2. Local Variables:

The variables which are declared inside a function are called local variables. Local variables are available only for the function in which we declared it. i.e from outside of function we cannot access.

Ex:

```
1)  def f1():
2)      a=10
3)      print(a)   # valid
4)
5)  def f2():
6)      print(a)    #invalid
7)
8)  f1()
9)  f2()
10)
11) NameError: name 'a' is not defined
```

global keyword:

We can use global keyword for the following 2 purposes:

1. To declare global variable inside function
2. To make global variable available to the function so that we can perform required modifications.

Ex:

```
1)  def f1():
2)      global a
3)      a=10
4)      print(a)
5)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
11)
12) Output
13) 10
14) 10
```

Note: If global variable and local variable having the same name then we can access global variable inside a function as follows.

lambda Function:

We can define by using lambda keyword

Syntax of lambda Function:
lambda argument_list : expression

Note:

By using Lambda Functions we can write very concise code so that readability of the program will be improved.

Ex:

```
1)  s=lambda a,b:a if a>b else b
2)  print("The Biggest of 10,20 is:",s(10,20))
3)  print("The Biggest of 100,200 is:",s(100,200))
4)
5)  Output
6)  The Biggest of 10,20 is: 20
7)  The Biggest of 100,200 is: 200
```

Note:

Lambda Function internally returns expression value and we are not required to write return statement explicitly.

Note:

Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(),map() and reduce() functions,b'z these functions expect function as argument.

filter() function:

We can use filter() function to filter values from the given sequence based on some condition.

filter(function,sequence)

Where function argument is responsible to perform conditional check sequence can be list or tuple or string.

Ex: Program to filter only even numbers from the list by using filter() function?

```
1)  l=[0,5,10,15,20,25,30]
2)  l1=list(filter(lambda x:x%2==0,l))
3)  print(l1)  #[0,10,20,30]
4)  l2=list(filter(lambda x:x%2!=0,l))
5)  print(l2) #[5,15,25]
```

map() function:

For every element present in the given sequence,apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Syntax: **map(function,sequence)**

The function can be applied on each element of sequence and generates new sequence.

Ex: For every element present in the list perform double and generate new list of doubles.

```
1) l=[1,2,3,4,5]
2) l1=list(map(lambda x:2*x,l))
3) print(l1)   #[2, 4, 6, 8, 10]
```

We can apply map() function on multiple lists also.But make sure all list should have same length.

Syntax: **map(lambda x,y:x*y,l1,l2))**
x is from l1 and y is from l2

Ex:

```
1. l1=[1,2,3,4]
2. l2=[2,3,4,5]
3. l3=list(map(lambda x,y:x*y,l1,l2))
4. print(l3)     #[2, 6, 12, 20]
```

reduce() function:

reduce() function reduces sequence of elements into a single element by applying the specified function.

Syntax:
reduce(function,sequence)

reduce() function present in functools module and hence we should write import statement.

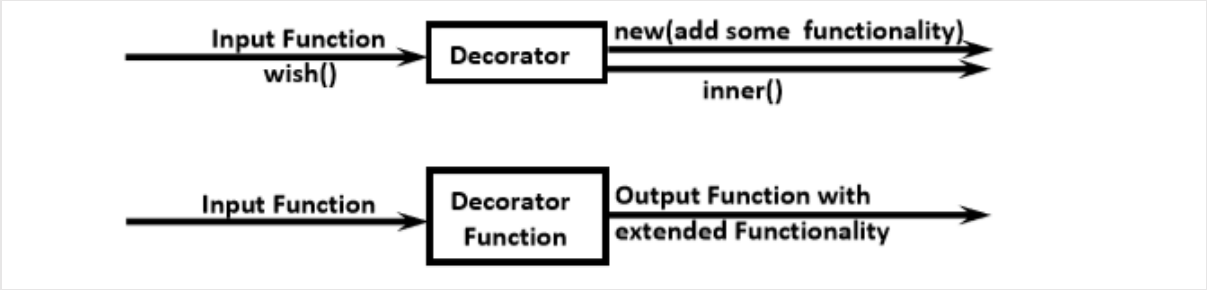
Ex:

```
1) result=reduce(lambda x,y:x*y,l)
2) print(result)  #12000000
```

Note:
In Python every thing is treated as object.
Even functions also internally treated as objects only.

Function Decorators:

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.



The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

Ex:

```
1)  def wish(name):
2)      print("Hello",name,"Good Morning")
```

This function can always print same output for any name

Hello Ranjan Good Morning
Hello Fahad Good Morning
Hello JP Good Morning

But we want to modify this function to provide different message if name is Sunny. We can do this without touching wish() function by using decorator.

Ex:

```
1) def decor(func):
2)     def inner(name):
3)         if name=="Ranjan":
4)             print("Welcome to SkillMine Family")
5)         else:
6)             func(name)
7)     return inner
8)
9) @decor
10) def wish(name):
11)     print("Hello",name,"Good Morning")
12)
13) wish("Ranjan")
```

14) wish("Anil")
15) wish("Bharat")
16)
17) Output
18) Welcome to Skillmine Family
19) Hello Anil Good Morning
20) Hello Bharat Good Morning

Note: In the above program whenever we call wish() function automatically decor function will be executed.

Python's Object Oriented Programming (OOPs):

What is Class :

⚽ In Python every thing is an object. To create objects we required some Model or Plan or Blue print, which is nothing but class.
⚽ We can write a class to represent properties (attributes) and actions (behaviour) of object.
⚽ Properties can be represented by variables.
⚽ Actions can be represented by Methods.
⚽ Hence class contains both variables and methods.

How to Define a class?

We can define a class by using class keyword.

Syntax: class className:
''' documentation string '''
variables: instance variables, static and local variables
methods: instance methods, static methods, class methods

Documentation string represents description of the class. Within the class doc string is always optional. We can get doc string by using the following 2 ways.

1. print(classname.__doc__)
2. help(classname)

Example:

```
1)  class Student:
2)      """ This is student class with required data"""
3)  print(Student.__doc__)
4)  help(Student)
```

Within the Python class we can represent data by using variables.
There are 3 types of variables are allowed in Python.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

Within the Python class, we can represent operations by using methods. The following are various types of allowed methods.

Example for class:

```
1)  class Student:
2)      """Developed by Ranjan for python demo"""
3)      def __init__(self):
4)          self.name='Ranjan'
5)          self.age=40
6)          self.marks=80
7)
8)      def talk(self):
9)          print("Hello I am :",self.name)
10)         print("My Age is:",self.age)

    print("My Marks are:",self.marks)
```

What is Object:

Physical existence of a class is nothing but object. We can create any number of objects for a class.

Syntax to create object: referencevariable = classname()

Example: s = Student()

What is Reference Variable:

The variable which can be used to refer object is called reference variable.
By using reference variable, we can access properties and methods of object.

Program: Write a Python program to create a Student class and Creates an object to it. Call the method talk() to display student details

```
1)  class Student:
2)
3)      def __init__(self,name,rollno,marks):
```

```
4)          self.name=name
5)          self.rollno=rollno
6)          self.marks=marks
7)
8)      def talk(self):
9)          print("Hello My Name is:",self.name)
10)         print("My Rollno is:",self.rollno)
11)         print("My Marks are:",self.marks)
12)
13) s1=Student("Ranjan",101,90)
14) s1.talk()
```

Output:

```
D:\Python>py test.py
Hello My Name is: Ranjan
My Rollno is: 101
My Marks are: 90
```

Self variable:

self is the default variable which is always pointing to current object (like this keyword in Java)

By using self we can access instance variables and instance methods of object.

Note:

1. self should be first parameter inside constructor
   def __init__(self):

2. self should be first parameter inside instance methods
   def talk(self):

Constructor Concept:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be exeucted only once.
- Constructor can take atleast one argument(atleast self)

- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

```
1)  def __init__(self,name,rollno,marks):
2)          self.name=name
```

```
3)          self.rollno=rollno
4)          self.marks=marks
```

Program to demonstrate constructor will execute only once per object:

```
1)  class Test:
2)
3)      def __init__(self):
4)          print("Constructor exeuction...")
5)
6)      def m1(self):
7)          print("Method execution...")
8)
9)  t1=Test()
10) t2=Test()
11) t3=Test()
12) t1.m1()
```

Output

```
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method execution...
```

Differences between Methods and Constructors:

Method	Constructor
1. Name of method can be any name	1. Constructor name should be always __init__
2. Method will be executed if we call that method	2. Constructor will be executed automatically at the time of object creation.
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once
4. Inside method we can write business logic	4. Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

1. Instance Variables:

If the value of a variable is varied from object to object, then such type of variables are called instance variables.

For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:

1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

1. Inside Constructor by using self variable:

We can declare instance variables inside a constructor by using self keyword. Once we creates object, automatically these variables will be added to the object.

Example:

```
1)  class Employee:
2)
3)    def __init__(self):
4)       self.eno=100
5)       self.ename='Ranjan'
6)       self.esal=10000
7)
8)  e=Employee()
9)  print(e.__dict__)
```

Output: {'eno': 100, 'ename': 'Ranjan', 'esal': 10000}

2. Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable. If any instance variable declared inside instance method, that instance variable will be added once we call taht method.

Example:

```
1)  class Test:
2)
3)     def __init__(self):
4)        self.a=10
5)        self.b=20
6)
7)     def m1(self):
8)        self.c=30
9)
10) t=Test()
11) t.m1()
```

```
12) print(t.__dict__)
```

Output

```
{'a': 10, 'b': 20, 'c': 30}
```

How to access Instance variables:

We can access instance variables with in the class by using self variable and outside of the class by using object reference.

```
1)  class Test:
2)
3)      def __init__(self):
4)          self.a=10
5)          self.b=20
6)
7)      def display(self):
8)          print(self.a)
9)          print(self.b)
10)
11) t=Test()
12) t.display()
13) print(t.a,t.b)
```

Output

```
10
20
10 20
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

```
del self.variableName
```

2. From outside of class we can delete instance variables as follows

```
del objectreference.variableName
```

Example:

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)          self.c=30
6)          self.d=40
7)      def m1(self):
8)          del self.d
```

```
9)
10) t=Test()
11) print(t.__dict__)
12) t.m1()
13) print(t.__dict__)
14) del t.c
15) print(t.__dict__)
```

Output

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Note: The instance variables which are deleted from one object, will not be deleted from other objects.

3. Outside of the class by using object reference variable:

We can also add instance variables outside of a class to a particular object.

```
1)  class Test:
2)
3)      def __init__(self):
4)          self.a=10
5)          self.b=20
6)
7)      def m1(self):
8)          self.c=30
9)
10) t=Test()
11) t.m1()
12) t.d=40
13) print(t.__dict__)
```

Output {'a': 10, 'b': 20, 'c': 30, 'd': 40}

How to access Instance variables:

We can access instance variables with in the class by using self variable and outside of the class by using object reference.

```
14) class Test:
15)
16)     def __init__(self):
17)         self.a=10
18)         self.b=20
```

```
19)
20)     def display(self):
21)         print(self.a)
22)         print(self.b)
23)
24) t=Test()
25) t.display()
26) print(t.a,t.b)
```

Output

```
10
20
10 20
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

```
del self.variableName
```

2. From outside of class we can delete instance variables as follows

```
del objectreference.variableName
```

Example:

```
16) class Test:
17)     def __init__(self):
18)         self.a=10
19)         self.b=20
20)         self.c=30
21)         self.d=40
22)     def m1(self):
23)         del self.d
24)
25) t=Test()
26) print(t.__dict__)
27) t.m1()
28) print(t.__dict__)
29) del t.c
30) print(t.__dict__)
```

Output

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Note: The instance variables which are deleted from one object,will not be deleted from other objects.

Example:

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)          self.c=30
6)          self.d=40
7)
8)
9)  t1=Test()
10) t2=Test()
11) del t1.a
12) print(t1.__dict__)
13) print(t2.__dict__)
```

Output

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

Example:

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)
6)  t1=Test()
7)  t1.a=888
8)  t1.b=999
9)  t2=Test()
10) print('t1:',t1.a,t1.b)
11) print('t2:',t2.a,t2.b)
```

Output

```
t1: 888 999
t2: 10 20
```

1. Static variables:

If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.

For total class only one copy of static variable will be created and shared by all objects of that class.

We can access static variables either by class name or by object reference. But recommended to use class name.

Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside classmethod by using either class name or cls variable
5. Inside static method by using class name

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          Test.b=20
5)      def m1(self):
6)          Test.c=30
7)      @classmethod
8)      def m2(cls):
9)          cls.d1=40
10)         Test.d2=400
11)     @staticmethod
12)     def m3():
13)         Test.e=50
14) print(Test.__dict__)
15) t=Test()
16) print(Test.__dict__)
17) t.m1()
18) print(Test.__dict__)
19) Test.m2()
20) print(Test.__dict__)
21) Test.m3()
22) print(Test.__dict__)
23) Test.f=60
24) print(Test.__dict__)
25)
```

How to access static variables:

1. inside constructor: by using either self or classname
2. inside instance method: by using either self or classname
3. inside class method: by using either cls variable or classname
4. inside static method: by using classname
5. From outside of class: by using either object reference or classnmae

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          print(self.a)
```

```
5)          print(Test.a)
6)      def m1(self):
7)          print(self.a)
8)          print(Test.a)
9)      @classmethod
10)     def m2(cls):
11)         print(cls.a)
12)         print(Test.a)
13)     @staticmethod
14)     def m3():
15)         print(Test.a)
16) t=Test()
17) print(Test.a)
18) print(t.a)
19) t.m1()
20) t.m2()
21) t.m3()
```

Where we can modify the value of static variable:

Anywhere either with in the class or outside of class we can modify by using classname.
But inside class method, by using cls variable.

Example:

```
1)  class Test:
2)      a=777
3)      @classmethod
4)      def m1(cls):
5)          cls.a=888
6)      @staticmethod
7)      def m2():
8)          Test.a=999
9)  print(Test.a)
10) Test.m1()
11) print(Test.a)
12) Test.m2()
13) print(Test.a)
```

Output
777
888
999

If we change the value of static variable by using either self or object reference variable:

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed,just a new instance variable with that name will be added to that particular object.

Example 1:

```
1)  class Test:
2)      a=10
3)      def  m1(self):
4)          self.a=888
5)  t1=Test()
6)  t1.m1()
7)  print(Test.a)
8)  print(t1.a)
```

Output

10
888

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

```
del   classname.variablename
```

But inside classmethod we can also use cls variable

```
del   cls.variablename
```

```
1)  class Test:
2)      a=10
3)      @classmethod
4)      def m1(cls):
5)          del cls.a
6)  Test.m1()
7)  print(Test.__dict__)
```

Example:

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          Test.b=20
5)          del Test.a
6)      def m1(self):
7)          Test.c=30
8)          del Test.b
9)      @classmethod
10)     def m2(cls):
11)         cls.d=40
```

```
12)         del Test.c
13)     @staticmethod
14)     def m3():
15)         Test.e=50
16)         del Test.d
17) print(Test.__dict__)
18) t=Test()
19) print(Test.__dict__)
20) t.m1()
21) print(Test.__dict__)
22) Test.m2()
23) print(Test.__dict__)
24) Test.m3()
25) print(Test.__dict__)
26) Test.f=60
27) print(Test.__dict__)
28) del Test.e
29) print(Test.__dict__)
```

****

Note: By using object reference variable/self we can read static variables, but we cannot modify or delete.

If we are trying to modify, then a new instance variable will be added to that particular object.
t1.a = 70

If we are trying to delete then we will get error.

Example:

```
1)  class Test:
2)      a=10
3)
4)  t1=Test()
5)  del t1.a      ===>AttributeError: a
```

We can modify or delete static variables only by using classname or cls variable.

If we change the value of static variable by using either self or object reference variable:

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed,just a new instance variable with that name will be added to that particular object.

Example 1:

```
1)  class Test:
2)      a=10
3)      def  m1(self):
```

```
        self.a=888
t1=Test()
t1.m1()
print(Test.a)
print(t1.a)
```

Output

```
10
888
```

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

```
del   classname.variablename
```

But inside classmethod we can also use cls variable

```
del   cls.variablename
```

```
class Test:
    a=10
    @classmethod
    def m1(cls):
        del cls.a
Test.m1()
print(Test.__dict__)
```

****

Note: By using object reference variable/self we can read static variables, but we cannot modify or delete.

If we are trying to modify, then a new instance variable will be added to that particular object.
t1.a = 70

If we are trying to delete then we will get error.

Example:

```
class Test:
    a=10

t1=Test()
del t1.a     ===>AttributeError: a
```

We can modify or delete static variables only by using classname or cls variable.

Local variables:

Sometimes to meet temporary requirements of programmer,we can declare variables inside a method directly,such type of variables are called local variable or temporary variables.

Local variables will be created at the time of method execution and destroyed once method completes.

Local variables of a method cannot be accessed from outside of method.

Example:

```
1)  class Test:
2)      def m1(self):
3)          a=1000
4)          print(a)
5)      def m2(self):
6)          b=2000
7)          print(b)
8)  t=Test()
9)  t.m1()
10) t.m2()
```

Output
1000
2000

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods:

Inside method implementation if we are using instance variables then such type of methods are called instance methods.
Inside instance method declaration,we have to pass self variable.

```
def m1(self):
```

By using self variable inside method we can able to access instance variables.

Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```
1)  class Student:
```

```
2)      def __init__(self,name,marks):
3)          self.name=name
4)          self.marks=marks
5)      def display(self):
6)          print('Hi',self.name)
7)          print('Your Marks are:',self.marks)
8)      def grade(self):
9)          if self.marks>=60:
10)             print('You got First Grade')
11)         elif self.marks>=50:
12)             print('Yout got Second Grade')
13)         elif self.marks>=35:
14)             print('You got Third Grade')
15)         else:
16)             print('You are Failed')
17) n=int(input('Enter number of students:'))
18) for i in range(n):
19)     name=input('Enter Name:')
20)     marks=int(input('Enter Marks:'))
21)     s= Student(name,marks)
22)     s.display()
23)     s.grade()
24)     print()
```

ouput:

```
D:\Python>py test.py
Enter number of students:2
Enter Name: Ranjan
Enter Marks:90
Hi Ranjan
Your Marks are: 90
You got First Grade

Enter Name:Anil
Enter Marks:12
Hi Anil
Your Marks are: 12
You are Failed
```

Setter and Getter Methods:

We can set and get the values of instance variables by using getter and setter methods.

Setter Method:

setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

syntax:

```
def  setVariable(self,variable):
     self.variable=variable
```

Example:

```
def  setName(self,name):
     self.name=name
```

Getter Method:

Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

```
def  getVariable(self):
    return self.variable
```

Example:

```
def  getName(self):
     return self.name
```

Demo Program:

```
1)  class Student:
2)      def setName(self,name):
3)          self.name=name
4)
5)      def getName(self):
6)          return self.name
7)
8)      def setMarks(self,marks):
9)          self.marks=marks
10)
11)     def getMarks(self):
12)         return self.marks
13)
14) n=int(input('Enter number of students:'))
15) for i in range(n):
16)     s=Student()
17)     name=input('Enter Name:')
18)     s.setName(name)
19)     marks=int(input('Enter Marks:'))
20)     s.setMarks(marks)
21)
22)     print('Hi',s.getName())
23)     print('Your Marks are:',s.getMarks())
24)     print()
```

output:

```
D:\Python>py test.py
Enter number of students:2
Enter Name:Kabir
Enter Marks:100
Hi Kabir
Your Marks are: 100

Enter Name:Bharat
Enter Marks:80
Hi Bharat
Your Marks are: 80
```

2. Class Methods:

Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

We can declare class method explicitly by using @classmethod decorator.
For class method we should provide cls variable at the time of declaration

We can call classmethod by using classname or object reference variable.

Demo Program:

```
1)  class Animal:
2)      legs=4
3)      @classmethod
4)      def walk(cls,name):
5)          print('{} walks with {} legs...'.format(name,cls.legs))
6)  Animal.walk('Dog')
7)  Animal.walk('Cat')
```

Output

```
D:\Python>py test.py
Dog walks with 4 legs...
Cat walks with 4 legs...
```

Program to track the number of objects created for a class:

```
1)  class Test:
2)      count=0
3)      def __init__(self):
4)          Test.count =Test.count+1
5)      @classmethod
6)      def noOfObjects(cls):
7)          print('The number of objects created for test class:',cls.count)
8)
9)  t1=Test()
10) t2=Test()
```

```
11) Test.noOfObjects()
12) t3=Test()
13) t4=Test()
14) t5=Test()
15) Test.noOfObjects()
```

3. Static Methods:

In general these methods are general utility methods.
Inside these methods we won't use any instance or class variables.
Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator
We can access static methods by using classname or object reference

```
1)  class SkillMine:
2)
3)      @staticmethod
4)      def add(x,y):
5)          print('The Sum:',x+y)
6)
7)      @staticmethod
8)      def product(x,y):
9)          print('The Product:',x*y)
10)
11)     @staticmethod
12)     def average(x,y):
13)         print('The average:',(x+y)/2)
14)
15) SkillMine.add(10,20)
16) SkillMine.product(10,20)
17) SkillMine.average(10,20)
```

Output
The Sum: 30
The Product: 200
The average: 15.0

Note: In general we can use only instance and static methods.Inside static method we can access class level variables by using class name.

class methods are most rarely used methods in python.

Inner classes:

Sometimes we can declare a class inside another class,such type of classes are called inner classes.

Without existing one type of object if there is no chance of existing another type of object,then we should go for inner classes.

Example: Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

```
class Car:
    .....
    class Engine:
    ......
```

Example: Without existing university object there is no chance of existing Department object

```
class University:
    .....
    class Department:
    ......
```

eg3:
Without existing Human there is no chance of existin Head. Hence Head should be part of Human.

```
class Human:
  class Head:
```

Note: Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

Demo Program-1:

```
1)  class Outer:
2)      def __init__(self):
3)          print("outer class object creation")
4)      class Inner:
5)          def __init__(self):
6)              print("inner class object creation")
7)          def m1(self):
8)              print("inner class method")
9)  o=Outer()
10) i=o.Inner()
11) i.m1()
```

Output

```
outer class object creation
inner class object creation
inner class method
```

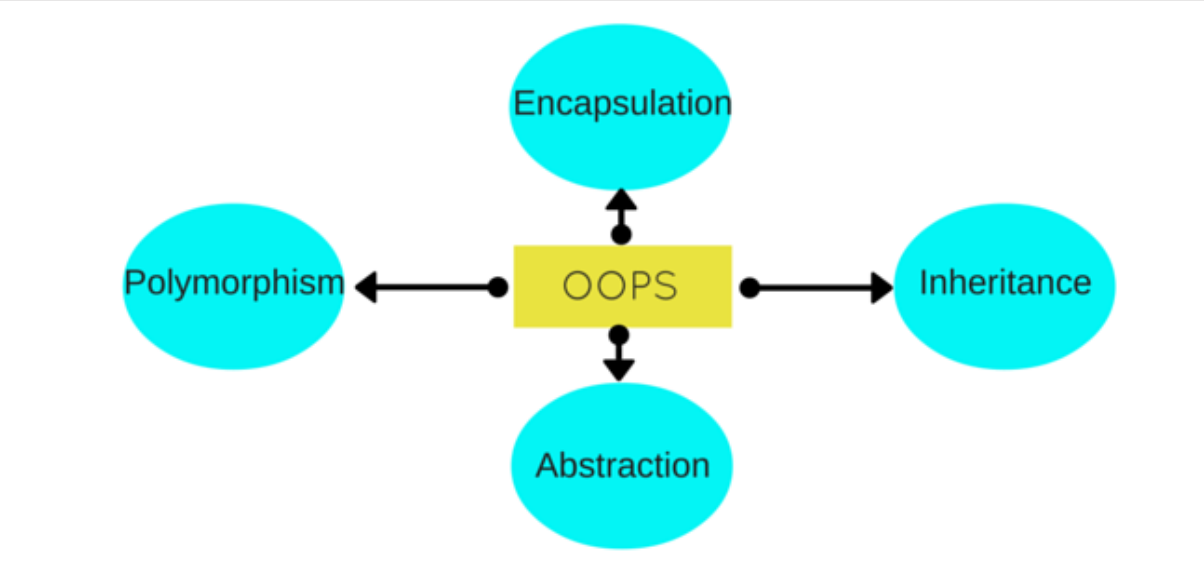
Note: The following are various possible syntaxes for calling inner class method

```
1.
o=Outer()
i=o.Inner()
i.m1()
```

2.
i=Outer().Inner()
i.m1()

3. Outer().Inner().m1()

Major Concepts of OOP:



Agenda:

- **Inheritance**
- **Has-A Relationship**
- **IS-A Relationship**
- **IS-A vs HAS-A Relationship**
- **Composition vs Aggregation**

- ## Types of Inheritance

  - **Single Inheritance.**
  - **Multi Level Inheritance.**
  - **Hierarchical Inheritance.**
  - **Multiple Inheritance.**
  - **Hybrid Inheritance.**
  - **Cyclic Inheritance.**

Using members of one class inside another class:

We can use members of one class inside another class by using the following ways

1. By Composition (Has-A Relationship)
2. By Inheritance (IS-A Relationship)

1. By Composition (Has-A Relationship):

By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).

The main advantage of Has-A Relationship is Code Reusability.

Demo Program-1:

```
1)  class Engine:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      def m1(self):
6)          print('Engine Specific Functionality')
7)  class Car:
8)      def __init__(self):
9)          self.engine=Engine()
10)     def m2(self):
11)         print('Car using Engine Class Functionality')
12)         print(self.engine.a)
13)         print(self.engine.b)
14)         self.engine.m1()
15) c=Car()
16) c.m2()
```

Output:
Car using Engine Class Functionality
10
20
Engine Specific Functionality

2. By Inheritance(IS-A Relationship):

What ever variables, methods and constructors available in the parent class by default available to the child classes and we are not required to rewrite. Hence the main advantage of inheritance is Code Reusability and we can extend existing functionality with some more extra functionality.

Syntax : class childclass(parentclass):

Demo Program for inheritance:

```
1)  class P:
2)      a=10
3)      def __init__(self):
4)          self.b=10
5)      def m1(self):
6)          print('Parent instance method')
7)      @classmethod
8)      def m2(cls):
9)          print('Parent class method')
10)     @staticmethod
11)     def m3():
12)         print('Parent static method')
13)
14) class C(P):
15)     pass
16)
17) c=C()
18) print(c.a)
19) print(c.b)
20) c.m1()
21) c.m2()
22) c.m3()
```

Output:

10
10
Parent instance method
Parent class method
Parent static method

Eg:

```
1)  class P:
2)      10 methods
3)  class C(P):
4)      5 methods
```

In the above example Parent class contains 10 methods and these methods automatically available to the child class and we are not required to rewrite those methods(Code Reusability) Hence child class contains 15 methods.

Note:

What ever members present in Parent class are by default available to the child class through inheritance. What ever methods present in Parent class are automatically available to the child class and hence on the child class reference we can call both parent class methods and child class methods.

Demo Program:

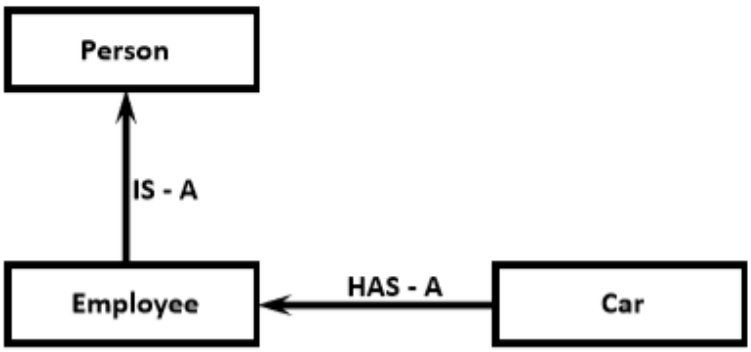
```
1)  class P:
2)      def m1(self):
3)          print("Parent class method")
4)  class C(P):
5)      def m2(self):
6)          print("Child class method")
7)
8)  c=C();
9)  c.m1()
10) c.m2()
```

IS-A vs HAS-A Relationship:

If we want to extend existing functionality with some more extra functionality then we should go for IS-A Relationship

If we dont want to extend and just we have to use existing functionality then we should go for HAS-A Relationship

Eg:
Employee class extends Person class Functionality
But Employee class just uses Car functionality but not extending



```
1)  class Car:
2)      def __init__(self,name,model,color):
3)          self.name=name
4)          self.model=model
5)          self.color=color
6)      def getinfo(self):
7)          print("\tCar Name:{} \n\t Model:{} \n\t Color:{}".format(self.name,self.model,self.col
    or))
8)
9)  class Person:
10)     def __init__(self,name,age):
11)         self.name=name
12)         self.age=age
13)     def eatndrink(self):
14)         print('Eat Biryani and Drink Water')
15)
16) class Employee(Person):
17)     def __init__(self,name,age,eno,esal,car):
18)         super().__init__(name,age)
19)         self.eno=eno
20)         self.esal=esal
21)         self.car=car
22)     def work(self):
23)         print("Coding in Python is  easy just compare to other language")
24)     def empinfo(self):
```

```
25)         print("Employee Name:",self.name)
26)         print("Employee Age:",self.age)
27)         print("Employee Number:",self.eno)
28)         print("Employee Salary:",self.esal)
29)         print("Employee Car Info:")
30)         self.car.getinfo()
31)
32) c=Car("Innova","2.5V","Grey")
33) e=Employee('Ranjan',24,100,10000,c)
34) e.eatndrink()
35) e.work()
36) e.empinfo()
```

Output:

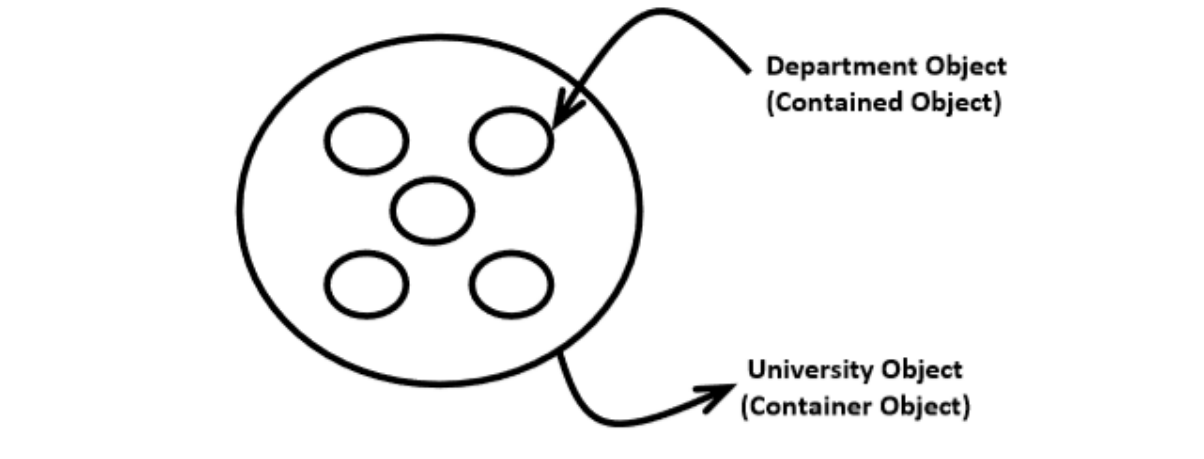
```
Eat Biryani and Drink Water
Coding Python is very easy compare to other language
Employee Name: Ranjan
Employee Age: 24
Employee Number: 100
Employee Salary: 10000
Employee Car Info:
    Car Name:Innova
    Model:2.5V
    Color:Grey
```

Composition vs Aggregation:

Composition:

Without existing container object if there is no chance of existing contained object then the container and contained objects are strongly associated and that strong association is nothing but Composition.

Eg:
University contains several Departments and without existing university object there is no chance of existing Department object. Hence University and Department objects are strongly associated and this strong association is nothing but Composition.

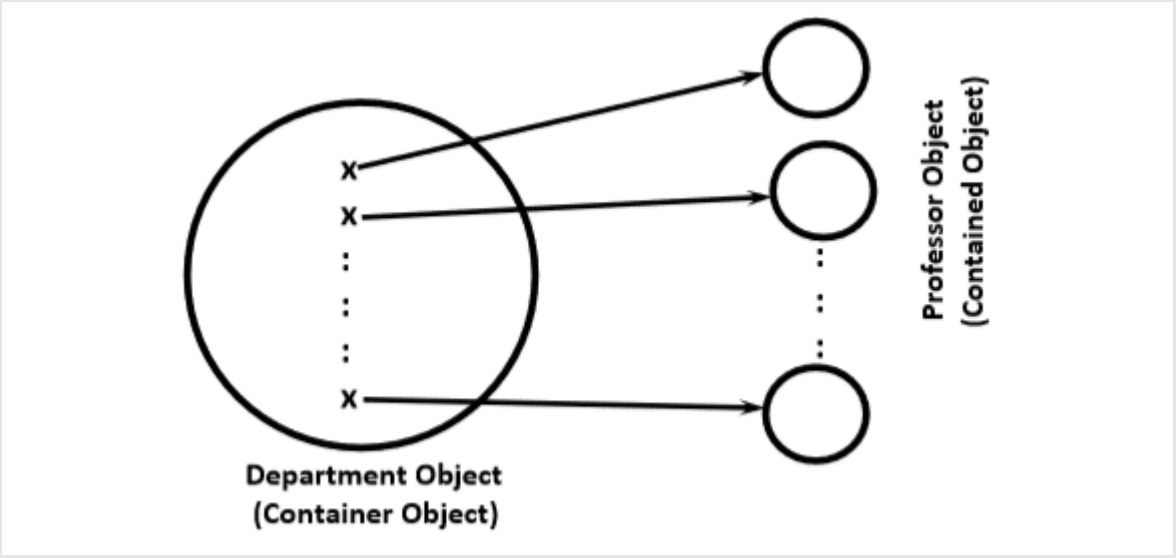


Aggregation:

Without existing container object if there is a chance of existing contained object then the container and contained objects are weakly associated and that weak association is nothing but Aggregation.

Eg:

Department contains several Professors. Without existing Department still there may be a chance of existing Professor. Hence Department and Professor objects are weakly associated, which is nothing but Aggregation.



Coding Example:

```
1)  class Student:
2)      collegeName='RCCIIT'
3)      def __init__(self,name):
4)          self.name=name
5)  print(Student.collegeName)
6)  s=Student('Ranjan')
7)  print(s.name)
```

Output:

RCCIIT

Ranjan

In the above example without existing Student object there is no chance of existing his name. Hence Student Object and his name are strongly associated which is nothing but Composition.

But without existing Student object there may be a chance of existing collegeName. Hence Student object and collegeName are weakly associated which is nothing but Aggregation.

Conclusion:

The relation between object and its instance variables is always Composition where as the relation between object and static variables is Aggregation.

Note: Whenever we are creating child class object then child class constructor will be executed. If the child class does not contain constructor then parent class constructor will be executed, but parent object won't be created.

Eg:

```
1)  class P:
2)      def __init__(self):
3)          print(id(self))
4)  class C(P):
5)      pass
6)  c=C()
7)  print(id(c))
```

Output:

6207088

6207088

Eg:

```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)  class Student(Person):
6)      def __init__(self,name,age,rollno,marks):
7)          super().__init__(name,age)
8)          self.rollno=rollno
9)          self.marks=marks
10)     def __str__(self):
11)         return 'Name={}\nAge={}\nRollno={}\nMarks={}'.format(self.name,self.age,self.rollno
12) s1=Student('Ranjan',24,101,90)
13) print(s1)
```

Output:

- Name=Ranjan
- Age=24
- Rollno=101
- Marks=90

Note:

In the above example when ever we are creating child class object both parent and child class constructors got executed to perform initialization of child object

Types of Inheritance:

1. Single Inheritance:

The concept of inheriting the properties from one class to another class is known as single inheritance.

Eg:

```
1)  class P:
2)      def m1(self):
3)          print("Parent Method")
4)  class C(P):
5)      def m2(self):
6)          print("Child Method")
7)  c=C()
8)  c.m1()
9)  c.m2()
```

Output:

Parent Method

Child Method

2. Multi Level Inheritance:

The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance

Eg:

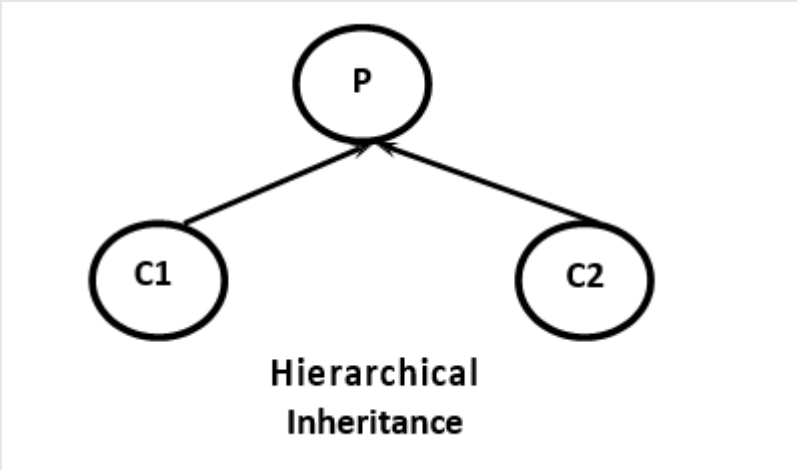
```
1)  class P:
2)      def m1(self):
3)          print("Parent Method")
4)  class C(P):
5)      def m2(self):
6)          print("Child Method")
7)  class CC(C):
8)      def m3(self):
9)          print("Sub Child Method")
10) c=CC()
11) c.m1()
12) c.m2()
13) c.m3()
```

Output:

- Parent Method
- Child Method
- Sub Child Method

3. Hierarchical Inheritance:

The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance.



Ex:

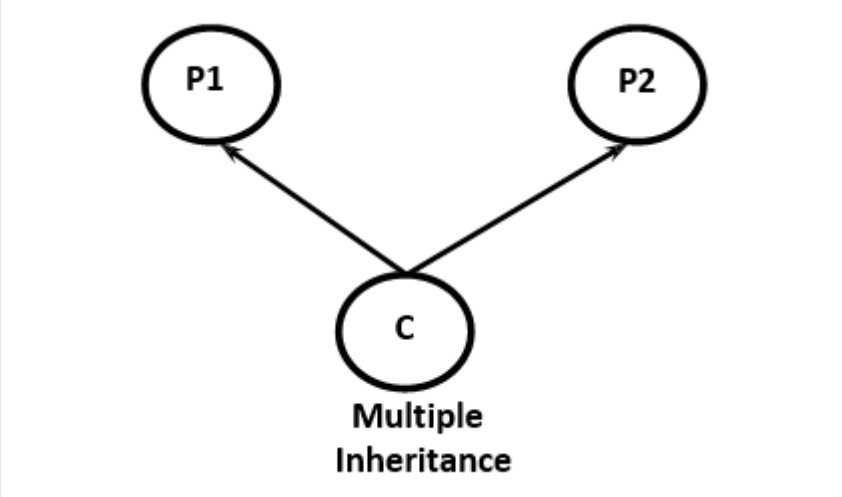
```
1)  class P:
2)      def m1(self):
3)          print("Parent Method")
4)  class C1(P):
5)      def m2(self):
6)          print("Child1 Method")
7)  class C2(P):
8)      def m3(self):
9)          print("Child2 Method")
10) c1=C1()
11) c1.m1()
12) c1.m2()
13) c2=C2()
14) c2.m1()
15) c2.m3()
```

Output:

- Parent Method
- Child1 Method
- Parent Method
- Child2 Method

4. Multiple Inheritance:

The concept of inheriting the properties from multiple classes into a single class at a time, is known as multiple inheritance.



Ex:

```
1)  class P1:
2)      def m1(self):
3)          print("Parent1 Method")
4)  class P2:
5)      def m2(self):
6)          print("Parent2 Method")
7)  class C(P1,P2):
8)      def m3(self):
9)          print("Child2 Method")
10) c=C()
11) c.m1()
12) c.m2()
13) c.m3()
```

Output:

- **Parent1 Method**
- **Parent2 Method**
- **Child2 Method**

If the same method is inherited from both parent classes,then Python will always consider the order of Parent classes in the declaration of the child class.

class C(P1,P2): ===>P1 method will be considered

class C(P2,P1): ===>P2 method will be considered

Ex:

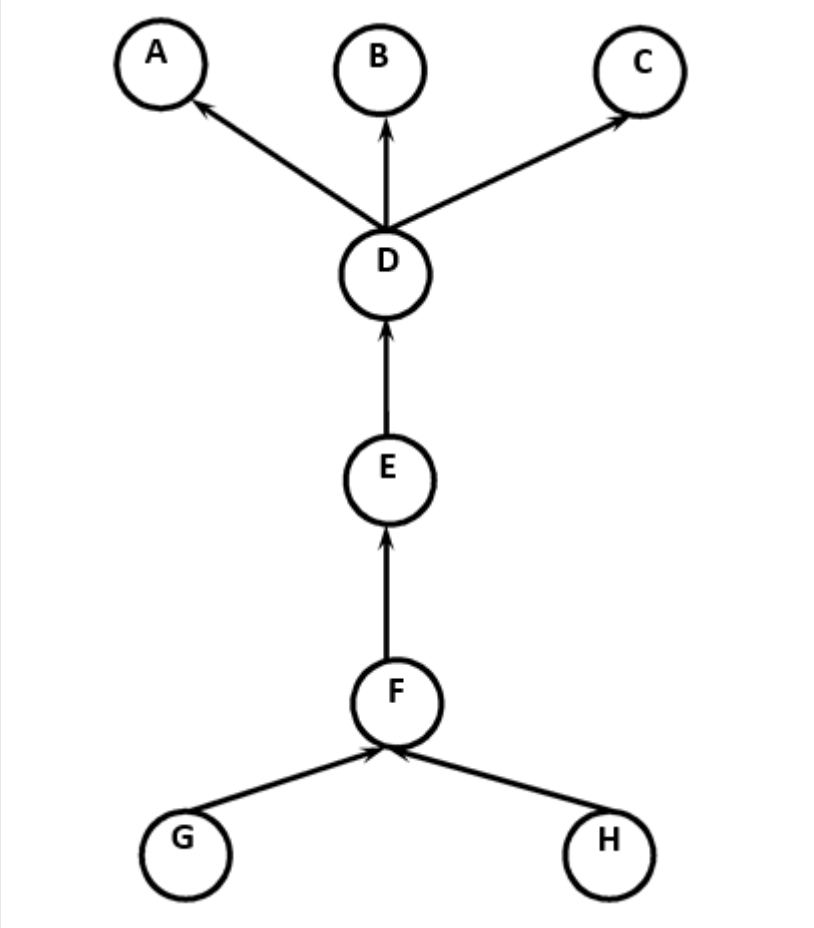
```
1)  class P1:
2)      def m1(self):
3)          print("Parent1 Method")
4)  class P2:
5)      def m1(self):
6)          print("Parent2 Method")
7)  class C(P1,P2):
8)      def m2(self):
9)          print("Child Method")
10) c=C()
11) c.m1()
12) c.m2()
```

Output:

- Parent1 Method
- Child Method

5. Hybrid Inheritance:

Combination of Single, Multi level, multiple and Hierarchical inheritance is known as Hybrid Inheritance.



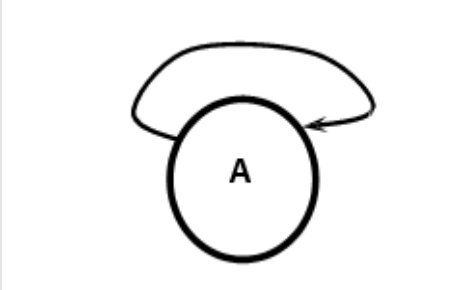
6. Cyclic Inheritance:

The concept of inheriting properties from one class to another class in cyclic way, is called Cyclic inheritance.Python won't support for Cyclic Inheritance of course it is really not required.

Eg – 1:

class A(A):pass

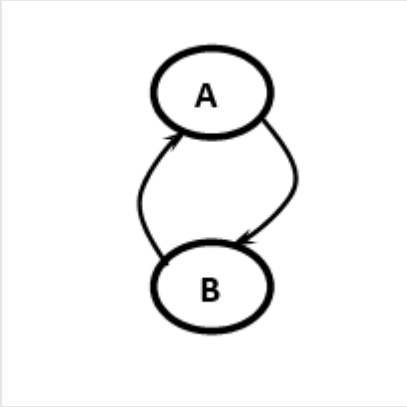
NameError: name 'A' is not defined



Ex-2:

```
1)  class A(B):
2)      pass
3)  class B(A):
4)      pass
```

NameError: name 'B' is not defined



super() Method:

super() is a built-in method which is useful to call the super class constructors, variables and methods from the child class.

Polymorphism

Poly means many. Morphs means forms.

Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes.(overriding)

Related to polymorphism the following topics are important

1. Overloading

- **a. Operator Overloading**
- **b. Method Overloading**
- **c. Constructor Overloading**

3. Overriding

1. **Method overriding**
2. **constructor overriding**

Overloading:

We can use same operator or methods for different purposes.

Eg1: + operator can be used for Arithmetic addition and String concatenation print(10+20)#30

```
print('Skill'+'Mine')#SkillMine
```

Eg2: * operator can be used for multiplication and string repetition purposes. print(10*20)#200

```
print('Ranjan'*2)#RanjanRanjan
```

Eg3: We can use deposit() method to deposit cash or cheque or dd deposit(cash)

```
deposit(cheque)

deposit(dd)
```

There are 3 types of overloading

1. Operator Overloading

2. Method Overloading

3. Constructor Overloading

1. Operator Overloading:

We can use the same operator for multiple purposes, which is nothing but operator

overloading. Python supports operator overloading.

Eg1: **+ operator can be used for Arithmetic addition and String concatenation print(10+20)#30**

```
print('Skill'+'Mine')#SkillMine
```

Eg2: *** operator can be used for multiplication and string repetition purposes. print(10*20)#200**

```
print('Ranjan'*3)#RanjanRanjanRanjan
```

Demo program to overload + operator for our Book class objects:

```
1)  class Book:
2)      def __init__(self,pages):
3)          self.pages=pages
4)
5)      def __add__(self,other):
6)          return self.pages+other.pages
7)
8)  b1=Book(100)
9)  b2=Book(200)
10) print('The Total Number of Pages:',b1+b2)
```

Output: The Total Number of Pages: 300

The following is the list of operators and corresponding magic methods.

- **+ ---> object. add (self,other)**
- **- ---> object.__sub (self,other)**
- *** ---> object. mul__(self,other)**
- **/ ---> object. div (self,other)**
- **// ---> object. floordiv (self,other)**
- **% ---> object. mod__(self,other)**
- **** ---> object. pow (self,other)**

- += ---> object. iadd (self,other)
- -= ---> object. isub (self,other)
- *= ---> object. imul (self,other)
- /= ---> object.__idiv (self,other)
- //= ---> object. ifloordiv (self,other)
- %= ---> object. imod (self,other)
- **= ---> object.__ipow__(self,other)
- < ---> object. lt__(self,other)
- <= ---> object. le__(self,other)
- > ---> object. gt__(self,other)
- >= ---> object. ge (self,other)
- == ---> object.__eq__(self,other)
- != ---> object.__ne__(self,other)

Overloading > and <= operators for Student class objects:

```
1)  class Student:
2)      def __init__(self,name,marks):
3)          self.name=name
4)          self.marks=marks
5)      def __gt__(self,other):
6)          return self.marks>other.marks
7)      def __le__(self,other):
8)          return self.marks<=other.marks
9)
10)
11) print("10>20 =",10>20)
12) s1=Student("Ranjan",100)
13) s2=Student("Ravi",200)
14) print("s1>s2=",s1>s2)
15) print("s1<s2=",s1<s2)
16) print("s1<=s2=",s1<=s2)
17) print("s1>=s2=",s1>=s2)
```

Output:

- 10>20 = False
- s1>s2= False
- s1<s2= True
- s1<=s2= True
- s1>=s2= False

2. Method Overloading:

If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.

Eg: m1(int a)

m1(double d)

But in Python Method overloading is not possible.

If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Demo Program:

```
1)  class Test:
2)      def m1(self):
3)          print('no-arg method')
4)      def m1(self,a):
5)          print('one-arg method')
6)      def m1(self,a,b):
7)          print('two-arg method')
8)
9)  t=Test()
10) #t.m1()
11) #t.m1(10)
12) t.m1(10,20)
```

Output: two-arg method

In the above program python will consider only last method.

How we can handle overloaded method requirements in Python:

Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

Demo Program with Default Arguments:

```
1)  class Test:
2)      def sum(self,a=None,b=None,c=None):
3)          if a!=None and b!= None and c!= None:
4)              print('The Sum of 3 Numbers:',a+b+c)
5)          elif a!=None and b!= None:
6)              print('The Sum of 2 Numbers:',a+b)
7)          else:
8)              print('Please provide 2 or 3 arguments')
9)
10) t=Test()
```

```
11) t.sum(10,20)
12) t.sum(10,20,30)
13) t.sum(10)
```

Output:

- The Sum of 2 Numbers: 30
- The Sum of 3 Numbers: 60
- Please provide 2 or 3 arguments

3. Constructor Overloading:

Constructor overloading is not possible in Python.

If we define multiple constructors then the last constructor will be considered.

Ex:

```
1)  class Test:
2)      def __init__(self):
3)          print('No-Arg Constructor')
4)
5)      def __init__(self,a):
6)          print('One-Arg constructor')
7)
8)      def __init__(self,a,b):
9)          print('Two-Arg constructor')
10) #t1=Test()
11) #t1=Test(10)
12) t1=Test(10,20)
```

Output: Two-Arg constructor

In the above program only Two-Arg Constructor is available.

But based on our requirement we can declare constructor with default arguments and variable number of arguments.

Method overriding:

What ever members available in the parent class are bydefault available to the child class through inheritance. If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.

Overriding concept applicable for both methods and constructors.

Demo Program for Method overriding:

```
1)  class P:
2)      def property(self):
3)          print('Gold+Land+Cash+Power')
4)      def marry(self):
5)          print('Father Choice')
6)  class C(P):
7)      def marry(self):
8)          print('Child Choice')
9)
10) c=C()
11) c.property()
12) c.marry()
```

Output:

Gold+Land+Cash+Power

Child Choice

From Overriding method of child class,we can call parent class method also by using super()

method.

Ex:

```
1)  class P:
2)      def property(self):
3)          print('Gold+Land+Cash+Power')
4)      def marry(self):
5)          print('Father Choice')
6)  class C(P):
7)      def marry(self):
8)          super().marry()
9)          print('Child Choice')
10)
11) c=C()
12) c.property()
13) c.marry()
```

Output:

Gold+Land+Cash+Power

Father Choice

Child Choice

Python Data Hiding:

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type.

By default all members of a class can be accessed outside of class.

You can prevent this by making class members private or protected.

In Python, we use double underscore (__) before the attributes name to make those attributes private.

We can use single underscore (_) before the attributes name to make those attributes protected.

Ex:

```
1. class MyClass:
2.     __hiddenVariable = 0     # Private member of MyClass
3.     _protectedVar = 0        # Protected member of MyClass
4.
5.     # A member method that changes __hiddenVariable
6.     def add(self, increment):
7.         self.__hiddenVariable += increment
8.         print (self.__hiddenVariable)
9.
10.
11. myObject = MyClass()
12. myObject.add(2)
13. myObject.add(5)
14.
15. # This will causes error
16. print (MyClass.__hiddenVariable)
17. print (MyClass._protectedVar)
```

In the above program, we tried to access hidden variable outside the class using object and it threw an exception.

We can access the value of hidden attribute by a tricky syntax as object._className__attrName:

Ex:

```
1. # A Python program to demonstrate that hidden members can be
   accessed outside a class
2. class MyClass:
3.     __hiddenVariable = 10     # Hidden member of MyClass
4.
5.
6. myObject = MyClass()
7. print(myObject._MyClass__hiddenVariable)
```

Private methods are accessible outside their class, just not easily accessible. Nothing in Python is truly private; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names.

Difference between public, private and protected.

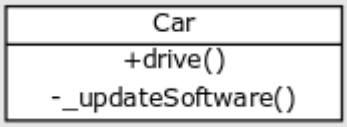
Mode	Description
public	A public member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function. By default all the members of a class would be public
private	A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class members can access private members. Practically, we make data private and related functions public so that they can be called from outside of the class
protected	A protected member is very similar to a private member but it provided one additional benefit that they can be accessed in sub classes which are called derived/child classes.

Encapsulation

In an object oriented python program, you can *restrict access* to methods and variables. T his can prevent the data from being modified by accident and is known as *encapsulation*.

Let's start with an example.

Private methods:



encapsulation. Restricted accesss to methods or variables

We create a class Car which has two methods: drive() and updateSoftware(). When a car object is created, it will call the private methods __updateSoftware().

This function cannot be called on the object directly, only from within the class.

Ex:

```
class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('driving. maxspeed ' + str(self.__maxspeed))

redcar = Car()
redcar.drive()
redcar.__maxspeed = 10  # will not change variable because its private
redcar.drive()
```

If you want to change the value of a private variable,a setter method is used.This is simply a method that sets the value of a private variable.

```
class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('driving. maxspeed ' + str(self.__maxspeed))

    def setMaxSpeed(self,speed):
        self.__maxspeed = speed

redcar = Car()
redcar.drive()
redcar.setMaxSpeed(320)
redcar.drive()
```

Type	Description
public methods	Accessible from anywhere
private methods	Accessible only in their own class. starts with two underscores
public variables	Accessible from anywhere
private variables	Accesible only in their own class or by a method if defined. starts with two underscores

Python Database Programming:

Sometimes as the part of Programming requirement we have to connect to the database and we have to perform several operations like creating tables, inserting data, updating data, deleting data, selecting data etc.

We can use SQL Language to talk to the database and we can use Python to send those SQL commands to the database.

Python provides inbuilt support for several databases like Oracle, MySql, SqlServer, GadFly, sqlite, etc.

Python has seperate module for each database.

Eg:

cx_Oralce module for communicating with Oracle database
pymssql module for communicating with Microsoft Sql Server

Standard Steps for Python database Programming:

1. Import database specific module

Eg: import cx_Oracle

2. Establish Connection between Python Program and database.

We can create this Connection object by using connect() function of the module.
con = cx_Oracle.connect(datbase information)

Eg: con=cx_Oracle.connect('username/password@hostname')

3. To execute our sql queries and to hold results some special object is required, which is nothing but Cursor object. We can create Cursor object by using cursor() method.

2. Establish Connection between Python Program and database. We can create this Conne ction object by using connect() function of the module.

con = cx_Oracle.connect(datbase information)

Eg: con=cx_Oracle.connect('username/password@hostname')

3. To execute our sql queries and to hold results some special object is required, which is n othing but Cursor object. We can create Cursor object by using cursor() method.

```
cursor=con.cursor()
```

4. Execute SQL Queries By using Cursor object. For this we can use the following methods

i) execute(sqlquery) ->To execute a single sql query

ii) executescript(sqlqueries) -> To execute a string of sql queries seperated by semi-colon ';
iii) executemany() -> To execute a Parameterized query

Eg: cursor.execute("select * from employees")

5. commit or rollback changes based on our requirement in the case of DML Queries(insert |update|delete)

- **commit() -> Saves the changes to the database**
- **rollback() -> rolls all temporary changes back**

6. Fetch the result from the Cursor object in the case of select queries

- **fetchone() -> To fetch only one row**
- **fetchall() -> To fetch all rows and it returns a list of rows**
- **fecthmany(n) -> To fetch first n rows**

Eg 1:

```
data =cursor.fetchone()

    print(data)
```

Eg 2:

```
data=cursor.fetchall()

for row in data:

print(row)
```

7. Close the resources

After completing our operations it is highly recommended to close the resources in the rev erse order of their opening by using close() methods.

```
cursor.close()
```

con.close()

Note: The following is the list of all important methods which can be used for python data base programming.

- **connect()**
- **cursor()**
- **execute()**
- **executescript()**
- **executemany()**
- **commit()**
- **rollback()**
- **fetchone()**
- **fetchall()**
- **fetchmany(n)**
- **fetch()**
- **close()**

Installing cx_Oracle:

From Normal Command Prompt (But not from Python console)execute the following command

Command: pip install cx_Oracle

How to Test Installation:

From python console execute the following command:

>>> help("modules")

In the output we can see cx_Oracle

....

App1: Program to connect with Oracle database and print its version.

```
1)  import cx_Oracle
2)  con= cx_Oracle.connect('scott/tiger@localhost')
3)  print(con.version)
4)  con.close()
```

Output:

D:\python>py db1.py

11.2.0.2.0

App2: Write a Program to create employees table in the oracle database :

employees(eno,ename,esal,eaddr)

```
2)   try:
3)       con=cx_Oracle.connect('username/password@hostname')
4)       cursor=con.cursor()
5)       cursor.execute("create table employees(eno number,ename varchar2(10),esal
number(10,2),eaddr varchar2(10))")
6)       print("Table created successfully")
7)   except cx_Oracle.DatabaseError as e:
8)       if con:
9)           con.rollback()
10)          print("There is a problem with sql",e)
11)  finally:
12)      if cursor:
13)          cursor.close()
14)      if con:
15)          con.close()
```

Note:

While performing DML Operations (insert|update|delte), compulsory we have to use commit() method,then only the results will be reflected in the database.

Assignment:

Q. Write a program to drop employees table from oracle database?

Q. Write a program to update employees table from oracle database?

Q. Write a program to insert employees table from oracle database?

Working with Mysql database:

Current version: 8.x

Vendor: SUN Micro Systems/Oracle Corporation

Open Source and Freeware

Default Port: 3306

Default user: root

Note: In MySQL, everything we have to work with our own databases, which are also known as Logical Databases.

The following are 4 default databases available in mysql.

1. **information_schema**
2. **mysql**
3. **performance_schema**
4. **test**

Commonly used commands in MySql:

1. To know available databases:

```
mysql> show databases;
```

2. To create our own logical database

```
mysql> create database ranjandb;
```

3. To drop our own database:

```
mysql> drop database ranjandb;
```

4. To use a particular logical database

```
mysql> use ranjandb;    OR    mysql> connect ranjandb;
```

5. To create a table:

```
create table employees(eno int(5) primary key,ename varchar(10),esal double(10,2),eaddr
varchar(10));
```

6. To insert data:

insert into employees values(100,'Ranjan',1000,'Mumbai');

insert into employees values(200,'Fahad',2000,'Bangalore');

In MySQL instead of single quotes we can use double quotes also.

Driver/Connector Information:

From Python program if we want to communicates with MySql database,compulsory some translator is required to convert python specific calls into mysql database specific calls and mysql database specific calls into python specific calls. This translator is nothing but Driver or Connector.

We have to download connector seperately from mysql database.

https://dev.mysql.com/downloads/connector/python/2.1.html

How to check installation:

From python console we have to use help("modules")

In the list of modules,compulsory mysql should be there.

Q. Write a Program to create table,insert data and display data by using mysql database

Program:

```
1)   import mysql.connector
2)   try:
3)       con=mysql.connector.connect(host='hostaname',database='ranjandb',user='SkillMine',password='12345')
4)       cursor=con.cursor()
5)       cursor.execute("create table employees(eno int(5) primary key,ename varchar(10),esal double(10,2),eaddr varchar(
10))")
6)       print("Table Created...")
7)
8)       sql = "insert into employees(eno, ename, esal, eaddr) VALUES(%s, %s, %s, %s)"
9)       records=[(100,'Ranjan',1000,'Mumbai'),
10)                (200,'Fahad',2000,'Bangalore'),
11)                (300,'JP',3000,'Bangalore')]
12)      cursor.executemany(sql,records)
13)      con.commit()
14)      print("Records Inserted Successfully...")
15)
16)      cursor.execute("select * from employees")
17)      data=cursor.fetchall()
18)      for row in data:
19)          print("Employee Number:",row[0])
20)          print("Employee Name:",row[1])
21)          print("Employee Salary:",row[2])
22)          print("Employee Address:",row[3])
23)          print()
24)          print()
25)  except mysql.connector.DatabaseError as e:
26)      if con:
27)          con.rollback()
28)          print("There is a problem with sql :",e)
29)  finally:
30)      if cursor:
31)          cursor.close()
32)      if con:
33)          con.close()
```

Assignment:

Q. Write a Program to copy data present in employees table of mysql database into Oracle database.

Thanks For Attending Python Training Session