Learn Complete Python In Simple Way

LANGUAGE FUNDAMENTALS STUDY MATERIAL

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.
- Python is recommended as first programming language for beginners.

Eg1: To print Helloworld

Java:

```
1)  public class HelloWorld
2)  {
3)    p s v main(String[] args)
4)    {
5)       SOP("Hello world");
6)    }
7)  }
```

C:

```
1)  #include<stdio.h>
2)  void main()
3)  {
4)     print("Hello world");
5)  }
```

Python:

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

Eg2: To print the sum of 2 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))
```

The name Python was selected from the TV Show
"The Complete Monty Python's Circus", which was broadcasted in BBC from 1969 to 1974.

Guido developed Python language by taking almost all programming features from different languages

1. Functional Programming Features from C
2. Object Oriented Programming Features from C++
3. Scripting Language Features from Perl and Shell Script
4. Modular Programming Features from Modula-3

Most of syntax in Python Derived from C and ABC languages.

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:
    - We can use already existing legacy non-Python code
    - 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. Etc.

Limitations of Python:

1) Performance wise not up to the mark because it is interpreted language.
2) Not using for mobile Applications.

Flavors of Python:

1) CPython:

It is the standard flavor of Python. It can be used to work with C lanugage Applications.

2) Jython OR JPython:

It is for Java Applications. It can run on JVM

3) IronPython:

It is for C#.Net platform

4) PyPy:

The main advantage of PyPy is performance will be improved because JIT compiler is available inside PVM.

5) RubyPython

For Ruby Platforms

6) AnacondaPython

It is specially designed for handling large volume of data processing.

Python Versions:

- **Python 1.0V introduced in Jan 1994**
- **Python 2.0V introduced in October 2000**
- **Python 3.0V introduced in December 2008**

Note: Python 3 won't provide backward compatibility to Python2 i.e there is no guarantee that Python2 programs will run in Python3.

Current versions

Python 3.6.1 **Python 2.7.13**

IDENTIFIERS

- A Name in Python Program is called Identifier.
- It can be Class Name OR Function Name OR Module Name OR Variable Name.

- a = 10

Rules to define Identifiers in Python:

1. The only allowed characters in Python are

   - alphabet symbols(either lower case or upper case)
   - digits(0 to 9)
   - underscore symbol(_)

   By mistake if we are using any other symbol like $ then we will get syntax error.

   - cash = 10 ✓
   - ca$h =20 ✗

2. Identifier should not starts with digit

   - 123total ✗
   - total123 ✓

3. Identifiers are case sensitive. Of course Python language is case sensitive language.

   - total=10
   - TOTAL=999
   - print(total) #10
   - print(TOTAL) #999

Identifier:

1) Alphabet Symbols (Either Upper case OR Lower case)

2) If Identifier is start with Underscore (_) then it indicates it is private.

3) Identifier should not start with Digits.

4) Identifiers are case sensitive.

5) We cannot use reserved words as identifiers
   Eg: def = 10 ✗

6) There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.

7) Dollor ($) Symbol is not allowed in Python.

Q) Which of the following are valid Python identifiers?

1) 123total ✗
2) total123 ✓
3) java2share ✓
4) ca$h ✗
5) _abc_abc_ ✓
6) def ✗
7) if ✗

Note:

1) If identifier starts with _ symbol then it indicates that it is private
2) If identifier starts with __(Two Under Score Symbols) indicating that strongly private identifier.
3) If the identifier starts and ends with two underscore symbols then the identifier is language defined special name, which is also known as magic methods.

4) Eg: __add__

RESERVED WORDS

In Python some words are reserved to represent some meaning or functionality.
Such types 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**

Eg: a= true ✗
a=True ✓

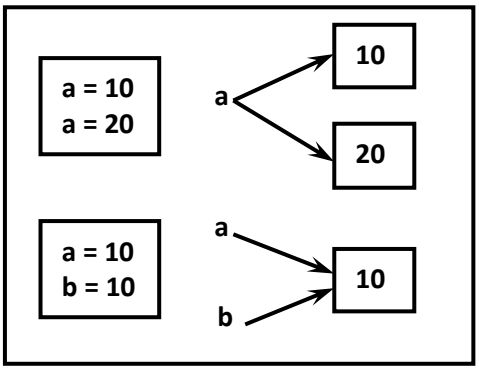
```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while',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

DATA TYPES

- Data Type represents 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



Note: Python contains several inbuilt functions

1) type()

to check the type of variable

2) id()

to get address of object

3) print()

to print the value

In Python everything is an Object.

1) int Data Type:

We can use int data type to represent whole numbers (integral values)

Eg: a = 10

type(a) #int

Note:

- In Python2 we have long data type to represent very large integral values.
- But in Python3 there is no long type explicitly and we can represent long values also by using int type only.

We can represent int values in the following ways

1) Decimal form
2) Binary form
3) Octal form
4) Hexa decimal form

I) Decimal Form (Base-10):

- It is the default number system in Python
- The allowed digits are: 0 to 9
- Eg: a =10

II) Binary Form (Base-2):

- The allowed digits are : 0 & 1
- Literal value should be prefixed with 0b or 0B

- Eg: a = 0B1111
  a = 0B123
  a = b111

III) Octal Form (Base-8):

- The allowed digits are : 0 to 7
- Literal value should be prefixed with 0o or 0O.

- Eg: a = 0o123
  a = 0o786

IV) Hexa Decimal Form (Base-16):

- The allowed digits are: 0 to 9, a-f (both lower and upper cases are allowed)
- Literal value should be prefixed with 0x or 0X

- Eg: a = 0XFACE
  a = 0XBeef
  a = 0XBeer

Note: Being a programmer we can specify literal values in decimal, binary, octal and hexa decimal forms. But PVM will always provide values only in decimal form.

- a=10
- b=0o10
- c=0X10
- d=0B10
- print(a)10
- print(b)8
- print(c)16
- print(d)2

Base Conversions

Python provide the following in-built functions for base conversions

1) bin():

We can use bin() to convert from any base to binary

```
1) >>> bin(15)
2) '0b1111'
3) >>> bin(0o11)
4) '0b1001'
5) >>> bin(0X10)
```

```
6) '0b10000'
```

2) oct():

We can use oct() to convert from any base to octal

```
1) >>> oct(10)
2) '0o12'
3) >>> oct(0B1111)
4) '0o17'
5) >>> oct(0X123)
6) '0o443'
```

3) hex():

We can use hex() to convert from any base to hexa decimal

```
1) >>> hex(100)
2) '0x64'
3) >>> hex(0B111111)
4) '0x3f'
5) >>> hex(0o12345)
6) '0x14e5'
```

2) Float Data Type:

- We can use float data type to represent floating point values (decimal values)
  Eg: f = 1.234
  type(f) float

- We can also represent floating point values by using exponential form (Scientific Notation)
  Eg: f = 1.2e3 → instead of 'e' we can use 'E'
  print(f) 1200.0

- The main advantage of exponential form is we can represent big values in less memory.

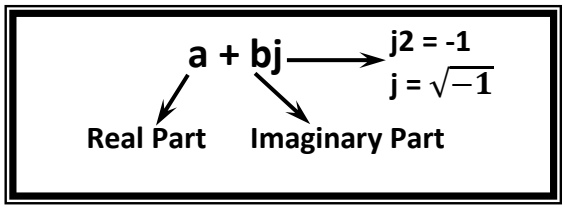
***Note:

We can represent int values in decimal, binary, octal and hexa decimal forms. But we can represent float values only by using decimal form.

```
1)  >>> f=0B11.01
2)   File "<stdin>", line 1
3)     f=0B11.01
4)           ^
5)  SyntaxError: invalid syntax
6)
7)  >>> f=0o123.456
8)   SyntaxError: invalid syntax
9)
10) >>> f=0X123.456
11)  SyntaxError: invalid syntax
```

3) 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.

```
1)  >>> a=10+1.5j
2)  >>> b=20+2.5j
3)  >>> c=a+b
4)  >>> print(c)
```

```
5) (30+4j)
6) >>> type(c)
7) <class 'complex'>
```

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.

4) bool Data Type:

- We can use this data type to represent boolean values.
- The only allowed values for this data type are:
- True and False

- Internally Python represents True as 1 and False as 0

```
b = True
type(b) →bool
```

Eg:

```
a = 10
b = 20
c = a<b
print(c) → True
```

True+True → 2
True-False → 1

5) str Data Type:

- str represents String data type.
- A String is a sequence of characters enclosed within single quotes or double quotes.

- s1='durga'
- s1="durga"
- By using single quotes or double quotes we cannot represent multi line string literals.

- s1="durga
  soft"

- For this requirement we should go for triple single quotes(''') or triple double quotes(""")

- s1='''durga
  soft'''

- s1="""durga
  soft"""

- We can also use triple quotes to use single quote or double quote in our String.
- ''' This is " character'''
  ' This i " Character '

- We can embed one string in another string
- '''This "Python class very helpful" for java students'''

Slicing of Strings:

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

-5	-4	-3	-2	-1
d	u	r	g	a
0	1	2	3	4

```
1) >>> s="durga"
2) >>> s[0]
3) 'd'
4) >>> s[1]
5) 'u'
6) >>> s[-1]
7) 'a'
8) >>> s[40]
```

IndexError: string index out of range

```
1)  >>> s[1:40]
2)  'urga'
3)  >>> s[1:]
4)  'urga'
5)  >>> s[:4]
6)  'durg'
7)  >>> s[:]
8)  'durga'
9)  >>>
10)
11) >>> s*3
12) 'durgadurgadurga'
13)
14) >>> len(s)
15) 5
```

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.

```
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 coersion.
- The following are various inbuilt functions for type casting.

1) int()
2) float()
3) complex()
4) bool()
5) str()

int():

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

```
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.

float():

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.

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)

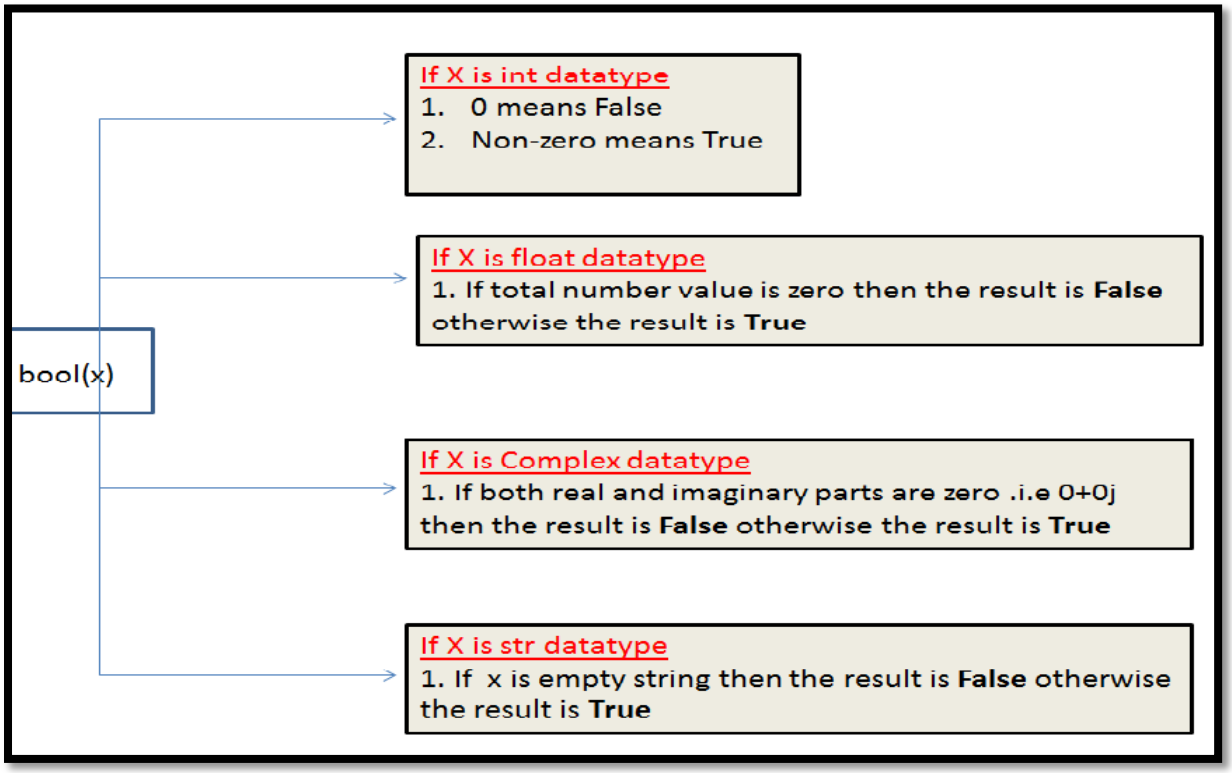
We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

Eg: complex(10, -2) → 10-2j
complex(True, False) → 1+0j

bool():

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

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



str():

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

```
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 won't 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.

```
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
```

6) bytes Data Type:

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

```
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
```

7) 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)
```

8) 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.**

Eg:

```
1) list=[10,10.5,'durga',True,10]
2)       print(list) # [10,10.5,'durga',True,10]
```

Eg:

```
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
14) 40
```

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

```
1) >>> list=[10,20,30]
2) >>> list.append("durga")
3) >>> list
4) [10, 20, 30, 'durga']
5) >>> list.remove(20)
6) >>> list
7) [10, 30, 'durga']
8) >>> list2=list*2
9) >>> list2
10) [10, 30, 'durga', 10, 30, 'durga']
```

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

9) 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

10) 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

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

Form-3: range(10, 20, 2)
2 means increment value

Eg:
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.

Eg:
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

We can create a list of values with range data type

Eg:

```
1) >>> l = list(range(10))
2) >>> l
3) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

11) 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.

1) Insertion order is not preserved
2) Duplicates are not allowed
3) Heterogeneous objects are allowed
4) Index concept is not applicable
5) It is mutable collection
6) Growable in nature

Eg:

```
1)  s={100,0,10,200,10,'durga'}
2)  s # {0, 100, 'durga', 200, 10}
3)  s[0] → TypeError: 'set' object does not support indexing
```

- set is growable in nature, based on our requirement we can increase or decrease the size.

```
1)  >>> s.add(60)
2)  >>> s
3)  {0, 100, 'durga', 200, 10, 60}
4)  >>> s.remove(100)
5)  >>> s
6)  {0, 'durga', 200, 10, 60}
```

12) 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'
```

13) 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:'durga',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.

Eg:

```
>>> d={101:'durga',102:'ravi',103:'shiva'}
>>> d[101]='sunny'
>>> d
{101: 'sunny', 102: 'ravi', 103: 'shiva'}

We can create empty dictionary as follows
d={ }
We can add key-value pairs as follows
d['a']='apple'
d['b']='banana'
print(d)
```

Note: dict is mutable and the order won't be preserved.

Note:

1) In general we can use bytes and bytearray data types to represent binary information like images, video files etc
2) 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.
3) In Python there is no char data type. Hence we can represent char values also by using str type.

Summary of Datatypes in Python 3

Datatype	Description	Is Immutable?	Example

Int	We can use to represent the whole/integral numbers	Immutable	>>> a=10
			>>> type(a)
			<class 'int'>

Float	We can use to represent the decimal/floating point numbers	Immutable	>>> b=10.5
			>>> type(b)
			<class 'float'>

Complex	We can use to represent the complex numbers	Immutable	>>> c=10+5j
			>>> type(c)
			<class 'complex'>
			>>> c.real
			10.0
			>>> c.imag
			5.0

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

Str	To represent sequence of Characters	Immutable	>>> s='durga'
			>>> type(s)
			<class 'str'>
			>>> s="durga"
			>>> s='''Durga Software Solutions... Ameerpet'''
			>>> 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)
			<class 'bytes'>

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

range	To represent a range of values	Immutable	>>> r=range(10)
			>>> r1=range(0,10)
			>>> r2=range(0,10,2)

list	To represent an ordered collection of objects	Mutable	>>> l=[10,11,12,13,14,15]
			>>> type(l)
			<class 'list'>

tuple	To represent an ordered collections of objects	Immutable	>>> t=(1,2,3,4,5)
			>>> type(t)
			<class 'tuple'>

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,'Durga',100,'Ramu'}
			>>> fs=frozenset(s)
			>>> type(fs)
			<class 'frozenset'>

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

14) 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
```

Escape Characters:

In String literals we can use esacpe characters to associate a special meaning.

```
>>> s="durga\nsoftware"
>>> print(s)
durga
software
>>> s="durga\tsoftware"
>>> print(s)
durga   software
>>> s="This is " symbol"
  File "<stdin>", line 1
    s="This is " symbol"
               ^
SyntaxError: invalid syntax
>>> s="This is \" symbol"
>>> print(s)
This is " symbol
```

The following are various important escape characters in Python

1) \n → New Line
2) \t → Horizontal Tab
3) \r → Carriage Return
4) \b → Back Space
5) \f → Form Feed
6) \v → Vertical Tab
7) \' → Single Quote
8) \" → Double Quote
9) \\ → 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.

Learn Complete Python In Simple Way

OPERATORS

STUDY MATERIAL

- Operator is a symbol that performs certain operations.
- Python provides the following set of operators

1) Arithmetic Operators
2) Relational Operators OR Comparison Operators
3) Logical operators
4) Bitwise oeprators
5) Assignment operators
6) Special operators

1) Arithmetic Operators:

1) + → Addition
2) – → Subtraction
3) * → Multiplication
4) / → Division Operator
5) % → Modulo Operator
6) // → Floor Division Operator
7) ** → Exponent Operator OR Power Operator

Eg: test.py

```
1) a=10
2) b=2
3) print('a+b=',a+b)
4) print('a-b=',a-b)
5) print('a*b=',a*b)
6) print('a/b=',a/b)
7) print('a//b=',a//b)
8) print('a%b=',a%b)
9) print('a**b=',a**b)
```

Output:

```
Python test.py OR py test.py
a+b = 12
a-b= 8
a*b= 20
```

a/b= 5.0
a//b= 5
a%b= 0
a**b= 100

Eg:

```
1)  a = 10.5
2)  b=2
3)
4)  a+b= 12.5
5)  a-b= 8.5
6)  a*b= 21.0
7)  a/b= 5.25
8)  a//b= 5.0
9)  a%b= 0.5
10) a**b= 110.25
```

Eg:

10/2 → 5.0
10//2 → 5
10.0/2 → 5.0
10.0//2 → 5.0

Note:

- / operator always performs floating point arithmetic. Hence it will always returns float value.

- But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If atleast one argument is float type then result is float type.

Note:

- We can use +,* operators for str type also.
- If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.

```
1)  >>> "durga"+10
2)  TypeError: must be str, not int
3)  >>> "durga"+"10"
4)  'durga10'
```

- If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

- 2*"durga"
  "durga"*2
  2.5*"durga" → TypeError: can't multiply sequence by non-int of type 'float'
  "durga"*"durga" → TypeError: can't multiply sequence by non-int of type 'str'

- + → String Concatenation Operator
- * → String Multiplication Operator

Note: For any number x,
x/0 and x%0 always raises "ZeroDivisionError"

10/0
10.0/0
.....

2) Relational Operators: >, >=, <, <=

```
1) a=10
2) b=20
3) print("a > b is ",a>b)
4) print("a >= b is ",a>=b)
5) print("a < b is ",a<b)
6) print("a <= b is ",a<=b)
7)
8) a > b is  False
9) a >= b is  False
10) a < b is  True
11)     a <= b is  True
```

We can apply relational operators for str types also.

Eg 2:

```
1) a="durga"
2) b="durga"
3) print("a > b is ",a>b)
4) print("a >= b is ",a>=b)
5) print("a < b is ",a<b)
6) print("a <= b is ",a<=b)
7)
```

```
8)  a > b is  False
9)  a >= b is  True
10) a < b is  False
11) a <= b is  True
```

Eg:

```
1)  print(True>True)   False
2)  print(True>=True)   True
3)  print(10 >True) True
4)  print(False > True) False
5)
6)  print(10>'durga')
7)     TypeError: '>' not supported between instances of 'int' and 'str'
```

Eg:

```
1)  a=10
2)  b=20
3)  if(a>b):
4)     print("a is greater than b")
5)  else:
6)     print("a is not greater than b")
```

Output: a is not greater than b

Note: Chaining of relational operators is possible. In the chaining, if all comparisons returns True then only result is True. If atleast one comparison returns False then the result is False

```
1)  10<20 → True
2)  10<20<30 → True
3)  10<20<30<40 → True
4)  10<20<30<40>50  → False
```

3) Equality Operators: ==, !=

We can apply these operators for any type even for incompatible types also.

```
1)  >>> 10==20
2)  False
3)  >>> 10!= 20
4)  True
5)  >>> 10==True
```

```
False
>>> False==False
True
>>> "durga"=="durga"
True
>>> 10=="durga"
    False
```

Note: Chaining concept is applicable for equality operators. If atleast one comparison returns False then the result is False. Otherwise the result is True.

```
>>> 10==20==30==40
False
>>> 10==10==10==10
True
```

4) Logical Operators: and, or, not

We can apply for all types.

❖ For boolean Types Behaviour:

and → If both arguments are True then only result is True
or → If atleast one arugemnt is True then result is True
not → Complement

True and False → False
True or False → True
not False → True

❖ For non-boolean Types Behaviour:

0 means False
non-zero means True
empty string is always treated as False

x and y:

If x is evaluates to false return x otherwise return y

Eg:

10 and 20
0 and 20

If first argument is zero then result is zero otherwise result is y

x or y:

If x evaluates to True then result is x otherwise result is y

10 or 20 → 10
0 or 20 → 20

not x:

If x is evalutates to False then result is True otherwise False

not 10 → False
not 0 → True

Eg:

```
1) "durga" and "durgasoft" ==>durgasoft
2) ""  and "durga" ==>""
3) "durga" and ""  ==>""
4) "" or "durga" ==>"durga"
5) "durga" or ""==>"durga"
6) not ""==>True
7) not "durga" ==>False
```

5) Bitwise Operators:

- **We can apply these operators bitwise.**
- **These operators are applicable only for int and boolean types.**
- **By mistake if we are trying to apply for any other type then we will get Error.**

- **&, |, ^, ~, <<, >>**

- **print(4&5) → Valid**
- **print(10.5 & 5.6)**
  → TypeError: unsupported operand type(s) for &: 'float' and 'float'

- **print(True & True) → Valid**

- **& → If both bits are 1 then only result is 1 otherwise result is 0**
- **| → If atleast one bit is 1 then result is 1 otherwise result is 0**
- **^ → If bits are different then only result is 1 otherwise result is 0**
- **~ → bitwise complement operator**
- **1 → 0 & 0 → 1**
- **<< → Bitwise Left Shift**

- >> → Bitwise Right Shift

- print(4&5) → 4
- print(4|5) → 5
- print(4^5) → 1

Operator	Description
&	If both bits are 1 then only result is 1 otherwise result is 0
|	If atleast one bit is 1 then result is 1 otherwise result is 0
^	If bits are different then only result is 1 otherwise result is 0
~	bitwise complement operator i.e 1 means 0 and 0 means 1
>>	Bitwise Left shift Operator
<<	Bitwise Right shift Operator

Bitwise Complement Operator (~):

We have to apply complement for total bits.

Eg: print(~5)→ -6

Note:

- **The most significant bit acts as sign bit. 0 value represents +ve number where as 1 represents -ve value.**
- **Positive numbers will be repesented directly in the memory where as -ve numbers will be represented indirectly in 2's complement form.**

6) Shift Operators:

<< Left Shift Operator

After shifting the empty cells we have to fill with zero

print(10<<2) → 40

~~0~~	~~0~~	0	0	1	0	1	0

0	0	1	0	1	0	0	0

>> Right Shift Operator

After shifting the empty cells we have to fill with sign bit.(0 for +ve and 1 for -ve)

print(10>>2) → 2

0	0	0	0	1	0	~~1~~	~~0~~

0	0	0	0	0	0	1	0

We can apply bitwise operators for boolean types also

- **print(True & False) → False**
- **print(True | False) → True**
- **print(True ^ False) → True**
- **print(~True) → -2**
- **print(True<<2) → 4**
- **print(True>>2) → 0**

7) Assignment Operators:

- **We can use assignment operator to assign value to the variable.**
  Eg: x = 10

- **We can combine asignment operator with some other operator to form compound assignment operator.**
  Eg: x += 10 → x = x+10

The following is the list of all possible compound assignment operators in Python.

- **+=**
- **-=**
- ***=**
- **/=**
- **%=**
- **//=**
- ****=**
- **&=**
- **|=**
- **^=**
- **>>=**
- **<<=**

Eg:

```
1) x=10
2) x+=20
3) print(x) → 30
```

Eg:

```
1) x=10
2) x&=5
3) print(x) → 0
```

8) Ternary Operator OR Conditional Operator

Syntax: x = firstValue if condition else secondValue

If condition is True then firstValue will be considered else secondValue will be considered.

Eg 1:

```
1) a,b=10,20
2) x=30 if a<b else 40
3) print(x)   #30
```

Eg 2: Read two numbers from the keyboard and print minimum value

```
1) a=int(input("Enter First Number:"))
2) b=int(input("Enter Second Number:"))
3) min=a if a<b else b
4) print("Minimum Value:",min)
```

Output:

```
Enter First Number:10
Enter Second Number:30
Minimum Value: 10
```

Note: Nesting of Ternary Operator is Possible.

Q) Program for Minimum of 3 Numbers

```
1) a=int(input("Enter First Number:"))
2) b=int(input("Enter Second Number:"))
3) c=int(input("Enter Third Number:"))
```

```
4) min=a if a<b and a<c else b if b<c else c
5) print("Minimum Value:",min)
```

Q) Program for Maximum of 3 Numbers

```
1) a=int(input("Enter First Number:"))
2) b=int(input("Enter Second Number:"))
3) c=int(input("Enter Third Number:"))
4) max=a if a>b and a>c else b if b>c else c
5) print("Maximum Value:",max)
```

Eg:

```
1) a=int(input("Enter First Number:"))
2) b=int(input("Enter Second Number:"))
3) print("Both numbers are equal" if a==b else "First Number is Less than Second
   Number" if a<b else "First Number Greater than Second Number")
```

Output:

```
D:\python_classes>py test.py
Enter First Number:10
Enter Second Number:10
Both numbers are equal

D:\python_classes>py test.py
Enter First Number:10
Enter Second Number:20
First Number is Less than Second Number

D:\python_classes>py test.py
Enter First Number:20
Enter Second Number:10
First Number Greater than Second Number
```

9) Special Operators:

Python defines the following 2 special operators

1) Identity Operators
2) Membership operators

1) Identity Operators

- We can use identity operators for address comparison.
- There are 2 identity operators are available
    1) is
    2) is not

- r1 is r2 returns True if both r1 and r2 are pointing to the same object.
- r1 is not r2 returns True if both r1 and r2 are not pointing to the same object.

Eg:

```
1) a=10
2) b=10
3) print(a is b)      True
4) x=True
5) y=True
6) print( x is y)     True
```

Eg:

```
1) a="durga"
2) b="durga"
3) print(id(a))
4) print(id(b))
5) print(a is b)
```

Eg:

```
1) list1=["one","two","three"]
2) list2=["one","two","three"]
3) print(id(list1))
4) print(id(list2))
5) print(list1 is list2)   False
6) print(list1 is not list2)   True
7) print(list1 == list2)    True
```

Note: We can use is operator for address comparison where as == operator for content comparison.

2) Membership Operators:

- We can use Membership operators to check whether the given object present in the given collection. (It may be String, List, Set, Tuple OR Dict)
- In → Returns True if the given object present in the specified Collection
- not in → Retruns True if the given object not present in the specified Collection

Eg:

```
1) x="hello learning Python is very easy!!!"
2) print('h' in x)    True
3) print('d' in x)    False
4) print('d' not in x)    True
5) print('Python' in x)    True
```

Eg:

```
1) list1=["sunny","bunny","chinny","pinny"]
2) print("sunny" in list1)  True
3) print("tunny" in list1)  False
4) print("tunny" not in list1)   True
```

Operator Precedence:

If multiple operators present then which operator will be evaluated first is decided by operator precedence.

Eg:
print(3+10*2) → 23
print((3+10)*2) → 26

The following list describes operator precedence in Python

1) () → Parenthesis
2) ** → Exponential Operator
3) ~, - → Bitwise Complement Operator, Unary Minus Operator
4) *, /, %, // → Multiplication, Division, Modulo, Floor Division
5) +, - → Addition, Subtraction
6) <<, >> → Left and Right Shift
7) & → Bitwise And
8) ^ → Bitwise X-OR
9) | → Bitwise OR
10) >, >=, <, <=, ==, != → Relational OR Comparison Operators
11) =, +=, -=, *=... → Assignment Operators

12) is , is not → Identity Operators
13) in , not in → Membership operators
14) not → Logical not
15) and → Logical and
16) or → Logical or

```
1) a=30
2) b=20
3) c=10
4) d=5
5) print((a+b)*c/d) → 100.0
6) print((a+b)*(c/d)) → 100.0
7) print(a+(b*c)/d) → 70.0
8)
9) 3/2*4+3+(10/5)**3-2
10) 3/2*4+3+2.0**3-2
11) 3/2*4+3+8.0-2
12) 1.5*4+3+8.0-2
13) 6.0+3+8.0-2
14) 15.0
```

Mathematical Functions (math Module)

- A Module is collection of functions, variables and classes etc.
- math is a module that contains several functions to perform mathematical operations.
- If we want to use any module in Python, first we have to import that module.
  `import math`
- Once we import a module then we can call any function of that module.

```
1) import math
2) print(math.sqrt(16))
3) print(math.pi)
```

Output

```
4.0
3.141592653589793
```

- We can create alias name by using as keyword. `import math as m`
- Once we create alias name, by using that we can access functions and variables of that module.

```
1) import math as m
2) print(m.sqrt(16))
```

```
3) print(m.pi)
```

- We can import a particular member of a module explicitly as follows
  from math import sqrt
  from math import sqrt,pi

- If we import a member explicitly then it is not required to use module name while accessing.

```
1) from math import sqrt,pi
2) print(sqrt(16))
3) print(pi)
4) print NameError: name (math.pi) 'math' is not defined
```

Important Functions of math Module:

1) ceil(x)
2) floor(x)
3) pow(x,y)
4) factorial(x)
5) trunc(x)
6) gcd(x,y)
7) sin(x)
8) cos(x)
9) tan(x)
10) ....

Important Variables of math Module:

pi3.14
e → 2.71
inf → infinity
nan → not a number

Q) Write a Python Program to find Area of Circle pi*r**2

```
1) from math import pi
2) r = 16
3) print("Area of Circle is :",pi*r**2)
```

Output: Area of Circle is: 804.247719318987

Learn Complete Python In Simple Way

INPUT AND OUTPUT STATEMENTS STUDY MATERIAL

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
"durga"→ 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 Python 3.

```
1)  >>> type(input("Enter value:"))
2)  Enter value:10
3)  <class 'str'>
4)
5)  Enter value:10.5
6)  <class 'str'>
7)
8)  Enter value:True
9)  <class 'str'>
```

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

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

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

```
1) x=int(input("Enter First Number:"))
2) y=int(input("Enter Second Number:"))
3) 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

```
1) eno=int(input("Enter Employee No:"))
2) ename=input("Enter Employee Name:")
3) esal=float(input("Enter Employee Salary:"))
4) eaddr=input("Enter Employee Address:")
5) married=bool(input("Employee Married ?[True|False]:"))
6) print("Please Confirm Information")
7) print("Employee No :",eno)
8) print("Employee Name :",ename)
9) print("Employee Salary :",esal)
10) print("Employee Address :",eaddr)
11) print("Employee Married ? :",married)
```

```
D:\Python_classes>py test.py
Enter Employee No:100
Enter Employee Name:Sunny
Enter Employee Salary:1000
Enter Employee Address:Mumbai
Employee Married ?[True|False]:True
Please Confirm Information
```

Employee No : 100
Employee Name : Sunny
Employee Salary : 1000.0
Employee Address : Mumbai
Employee Married ? : True

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)
```

```
D:\Python_classes>py test.py
Enter 2 numbers :10 20
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

```
1) a,b,c= [float(x) for x in input("Enter 3 float numbers :").split(',')]
2) print("The Sum is :", a+b+c)
```

```
D:\Python_classes>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
Output: 11.5
eval() can evaluate the Input to list, tuple, set, etc based the provided Input.

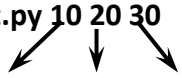
Eg: Write a Program to accept list from the keynboard on the display

```
1) l = eval(input("Enter List"))
2) print (type(l))
3) print(l)
```

COMMAND LINE ARGUMENTS

- argv is not Array it is a List. It is available sys Module.
- The Argument which are passing at the time of execution are called Command Line Arguments.

Eg: D:\Python_classes py test.py 10 20 30



Command Line Arguments

Within the Python Program this Command Line Arguments are available in argv. Which is present in SYS Module.

test.py	10	20	30

Note: argv[0] represents Name of Program. But not first Command Line Argument. argv[1] represent First Command Line Argument.

Program: To check type of argv from sys

```
import argv
print(type(argv))
```

D:\Python_classes\py test.py

Write a Program to display Command Line Arguments

```
1) from sys import argv
2) print("The Number of Command Line Arguments:", len(argv))
3) print("The List of Command Line Arguments:", argv)
4) print("Command Line Arguments one by one:")
5) for x in argv:
6)     print(x)
```

```
D:\Python_classes>py test.py 10 20 30
The Number of Command Line Arguments: 4
The List of Command Line Arguments: ['test.py', '10','20','30']
Command Line Arguments one by one:
test.py
10
20
30
```

```
1) from sys import argv
2) sum=0
3) args=argv[1:]
4) for x in args   :
5)    n=int(x)
6)    sum=sum+n
7) print("The Sum:",sum)
```

```
D:\Python_classes>py test.py 10 20 30 40
The Sum: 100
```

Note 1: Usually space is seperator between command line arguments. If our command line argument itself contains space then we should enclose within double quotes(but not single quotes)

```
1) from sys import argv
2) print(argv[1])
```

```
D:\Python_classes>py test.py Sunny Leone
Sunny

D:\Python_classes>py test.py 'Sunny Leone'
'Sunny

D:\Python_classes>py test.py "Sunny Leone"
Sunny Leone
```

Note 2: Within the Python program command line arguments are available in the String form. Based on our requirement, we can convert into corresponding type by using type casting methods.

```
1) from sys import argv
2) print(argv[1]+argv[2])
3) print(int(argv[1])+int(argv[2]))
```

```
D:\Python_classes>py test.py 10 20
1020
30
```

Note 3: If we are trying to access command line arguments with out of range index then we will get Error.

```
1) from sys import argv
2) print(argv[100])
```

```
D:\Python_classes>py test.py 10 20
IndexError: list index out of range
```

Note: In Python there is argparse module to parse command line arguments and display some help messages whenever end user enters wrong input.

input()
raw_input()

Command Line Arguments

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")
```

HelloWorld
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)
```

Output: The Values are : 10 20 30

By default output values are seperated by space.If we want we can specify seperator by using "sep" attribute

```
1) a,b,c=10,20,30
2) print(a,b,c,sep=',')
3) print(a,b,c,sep=':')
```

D:\Python_classes>py test.py
10,20,30
10:20:30

Form-4:print() with end attribute

```
1) print("Hello")
2) print("Durga")
3) print("Soft")
```

Output:
Hello
Durga
Soft

If we want output in the same line with space

```
1) print("Hello",end=' ')
2) print("Durga",end=' ')
3) print("Soft")
```

Output: Hello Durga Soft

Note: The default value for end attribute is \n, which is nothing but new line character.

Form-5: print(object) statement

We can pass any object (like list, tuple, set etc) as argument to the print() statement.

```
1) l=[10,20,30,40]
2) t=(10,20,30,40)
3) print(l)
4) print(t)
```

Form-6: print(String, variable list)

We can use print() statement with String and any number of arguments.

```
1) s  = "Durga"
2) a = 48
3) s1 ="Java"
4) s2 ="Python"
5) print("Hello",s,"Your Age is",a)
6) print("You are teaching",s1,"and",s2)
```

Output:

Hello Durga Your Age is 48
You are teaching java and Python

Form-7: print (formatted string)

1) %i → int
2) %d → int
3) %f → float
4) %s → String type

Syntax: print("formatted string" %(variable list))

Eg 1:

```
1) a=10
2) b=20
3) c=30
4) print("a value is %i" %a)
5) print("b  value is %d and c value is %d" %(b,c))
```

Output

```
a value is 10
b  value is 20 and c value is 30
```

Eg 2:

```
1) s="Durga"
2) list=[10,20,30,40]
3) print("Hello %s ...The List of Items are %s"  %(s,list))
```

Output: Hello Durga ...The List of Items are [10, 20, 30, 40]

Form-8: print() with replacement operator {}

Eg:

```
1) name = "Durga"
2) salary = 10000
3) gf = "Sunny"
4) print("Hello {0} your salary is {1} and Your Friend {2} is waiting".
   format(name,salary,gf))
5) print("Hello {x} your salary is {y} and Your Friend {z} is waiting".
   format(x=name,y=salary,z=gf))
```

Output

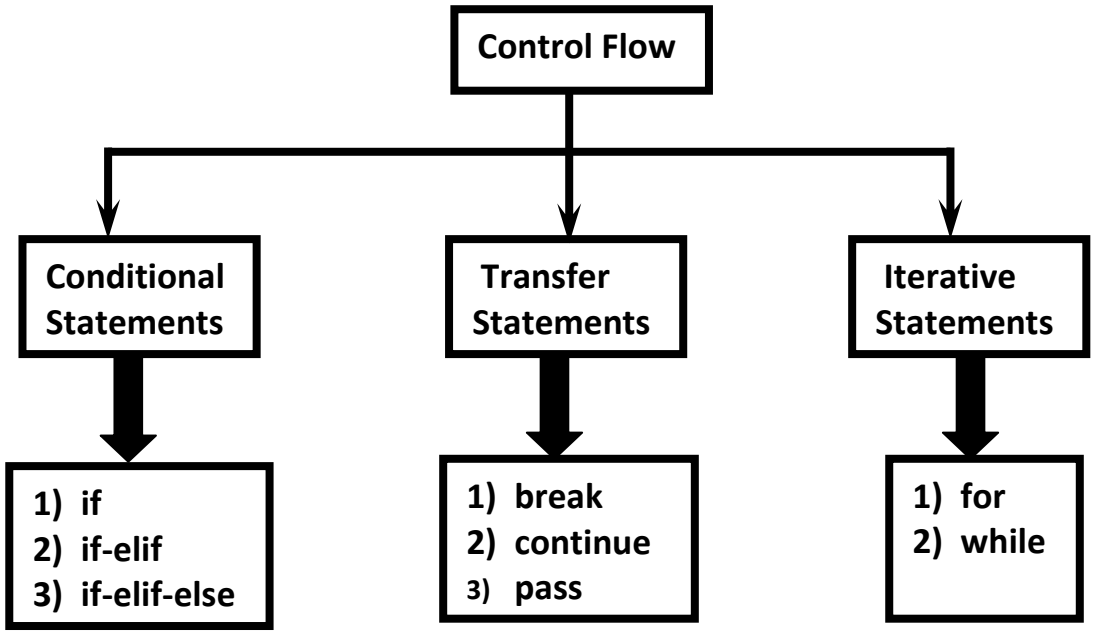
```
Hello Durga your salary is 10000 and Your Friend Sunny is waiting
Hello Durga your salary is 10000 and Your Friend Sunny is waiting
```

Learn Complete Python In Simple Way

FLOW CONTROL

STUDY MATERIAL

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=="durga" :
3)     print("Hello Durga Good Morning")
4) print("How are you!!!")
```

```
D:\Python_classes>py test.py
Enter Name:durga
Hello Durga Good Morning
How are you!!!
```

```
D:\Python_classes>py test.py
Enter Name: Ravi
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.

```
1) name=input("Enter Name:")
2) if name=="durga" :
3)     print("Hello Durga Good Morning")
4) else:
5)     print("Hello Guest Good Moring")
6) print("How are you!!!")
```

```
D:\Python_classes>py test.py
Enter Name:durga
Hello Durga Good Morning
How are you!!!

D:\Python_classes>py test.py
Enter Name:Ravi
Hello Guest Good Moring
How are you!!!
```

3) if-elif-else:

```
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.

```
1) brand=input("Enter Your Favourite Brand:")
2) if brand=="RC" :
3)     print("It is childrens brand")
4) elif brand=="KF":
5)     print("It is not that much kick")
6) elif brand=="FO":
7)     print("Buy one get Free One")
8) else :
9)     print("Other Brands are not recommended")
```

```
D:\Python_classes>py test.py
Enter Your Favourite Brand:RC
It is childrens brand

D:\Python_classes>py test.py
Enter Your Favourite Brand:KF
It is not that much kick

D:\Python_classes>py test.py
Enter Your Favourite Brand: KALYANI
Other Brands are not recommended
```

Note:

1) else part is always optional. Hence the following are various possible syntaxes.
    1) If
    2) if – else
    3) if-elif-else
    4) if-elif
2) There is no switch statement in Python

Q) Write a Program to find Biggest of given 2 Numbers from the Commad Prompt?

```
1) n1=int(input("Enter First Number:"))
2) n2=int(input("Enter Second Number:"))
3) if n1>n2:
4)     print("Biggest Number is:",n1)
5) else :
6)     print("Biggest Number is:",n2)
```

```
D:\Python_classes>py test.py
Enter First Number:10
Enter Second Number:20
Biggest Number is: 20
```

Q) Write a Program to find Biggest of given 3 Numbers from the Commad Prompt?

```
1) n1=int(input("Enter First Number:"))
2) n2=int(input("Enter Second Number:"))
3) n3=int(input("Enter Third Number:"))
4) if n1>n2 and n1>n3:
5)     print("Biggest Number is:",n1)
6) elif n2>n3:
7)     print("Biggest Number is:",n2)
8) else :
9)     print("Biggest Number is:",n3)
```

```
D:\Python_classes>py test.py
Enter First Number:10
Enter Second Number:20
Enter Third Number:30
Biggest Number is: 30

D:\Python_classes>py test.py
Enter First Number:10
Enter Second Number:30
Enter Third Number:20
Biggest Number is: 30
```

Q) Write a program to find smallest of given 2 numbers?
Q) Write a program to find smallest of given 3 numbers?
Q) Write a program to check whether the given number is even or odd?

Q) Write a Program to Check whether the given Number is in between 1 and 100?

```
1) n=int(input("Enter Number:"))
2) if n>=1 and n<=10 :
3)     print("The number",n,"is in between 1 to 10")
4) else:
5)     print("The number",n,"is not in between 1 to 10")
```

Q) Write a Program to take a Single Digit Number from the Key Board and Print is Value in English Word?

```
1)  0 → ZERO
2)  1 → ONE
3)
4)  n=int(input("Enter a digit from o to 9:"))
5)  if n==0 :
6)      print("ZERO")
7)  elif n==1:
8)      print("ONE")
9)  elif n==2:
10)     print("TWO")
11) elif n==3:
12)     print("THREE")
13) elif n==4:
14)     print("FOUR")
15) elif n==5:
16)     print("FIVE")
17) elif n==6:
18)     print("SIX")
19) elif n==7:
20)     print("SEVEN")
21) elif n==8:
22)     print("EIGHT")
23) elif n==9:
24)     print("NINE")
25) else:
26)     print("PLEASE ENTER A DIGIT FROM 0 TO 9")
```

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 the given string

```
1) s="Sunny Leone"
2) for x in s :
3)     print(x)
```

Output

```
S
u
n
n
y

L
e
o
n
e
```

Eg 2: 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
```

```
D:\Python_classes>py test.py
Enter some String: Sunny Leone
The character present at  0 index is : S
The character present at  1 index is : u
The character present at  2 index is : n
The character present at  3 index is : n
The character present at  4 index is : y
The character present at  5 index is :
The character present at  6 index is : L
The character present at  7 index is : e
The character present at  8 index is : o
The character present at  9 index is : n
The character present at  10 index is : e
```

Eg 3: To print Hello 10 times

```
1)  for x in range(10) :
2)      print("Hello")
```

Eg 4: To display numbers from 0 to 10

```
1)  for x in range(11) :
2)      print(x)
```

Eg 5: To display odd numbers from 0 to 20

```
1)  for x in range(21) :
2)      if (x%2!=0):
3)          print(x)
```

Eg 6: To display numbers from 10 to 1 in descending order

```
1)  for x in range(10,0,-1) :
2)          print(x)
```

Eg 7: To print sum of numbers presenst inside list

```
1)  list = eval(input("Enter List:"))
2)      sum=0;
3)      for x in list:
4)          sum=sum+x;
5)      print("The Sum=",sum)
```

```
D:\Python_classes>py test.py
Enter List:[10,20,30,40]
The Sum= 100

D:\Python_classes>py test.py
Enter List:[45,67]
The Sum= 112
```

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)
```

Eg: Write a program to prompt user to enter some name until entering Durga

```
1) name=""
2) while name!="durga":
3)    name=input("Enter Name:")
4) print("Thanks for confirmation")
```

Infinite Loops:

```
1) i=0;
2) while True :
3)    i=i+1;
4)    print("Hello",i)
```

Nested Loops:

Sometimes we can take a loop inside another loop,which are also known as nested loops.

```
1) for i in range(4):
2)    for j in range(4):
3)       print("i=",i,"   j=",j)
```

Output

```
D:\Python_classes>py test.py
i= 0   j= 0
i= 0   j= 1
i= 0   j= 2
```

```
i= 0   j= 3
i= 1   j= 0
i= 1   j= 1
i= 1   j= 2
i= 1   j= 3
i= 2   j= 0
i= 2   j= 1
i= 2   j= 2
i= 2   j= 3
i= 3   j= 0
i= 3   j= 1
i= 3   j= 2
i= 3   j= 3
```

Q) Write a Program to dispaly *'s in Right Angled Triangled Form

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
```

```
1) n = int(input("Enter number of rows:"))
2) for i in range(1,n+1):
3)     for j in range(1,i+1):
4)         print("*",end=" ")
5)     print()
```

Alternative Way

```
1) n = int(input("Enter number of rows:"))
2) for i in range(1,n+1):
3)     print("* " * i)
```

Q) <u>Write a Program to display *'s in Pyramid Style (Also known as Equivalent Triangle)</u>

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
* * * * * * *
```

```
1) n = int(input("Enter number of rows:"))
2) for i in range(1,n+1):
3)     print(" " * (n-i),end="")
4)     print("* "*i)
```

III. Transfer Statements

1) break:

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

```
1)  for i in range(10):
2)      if i==7:
3)          print("processing is enough..plz break")
4)          break
5)      print(i)
```

```
D:\Python_classes>py test.py
0
1
2
3
4
5
6
processing is enough..plz break
```

Eg:

```
1)  cart=[10,20,600,60,70]
2)  for item in cart:
3)      if item>500:
4)          print("To place this order insurence must be required")
5)          break
6)      print(item)
```

```
D:\Python_classes>py test.py
10
20
To place this order insurence must be required
```

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)
```

```
D:\Python_classes>py test.py
1
3
5
7
9
```

Eg 2:

```
1) cart=[10,20,500,700,50,60]
2) for item in cart:
3)     if item>=500:
4)         print("We cannot process this item :",item)
5)         continue
6)     print(item)
```

```
D:\Python_classes>py test.py
10
20
We cannot process this item : 500
We cannot process this item : 700
50
60
```

Eg 3:

```
1) numbers=[10,20,0,5,0,30]
2) for n in numbers:
3)     if n==0:
4)         print("Hey how we can divide with zero..just skipping")
5)         continue
6)     print("100/{} = {}".format(n,100/n))
```

Output

```
100/10 = 10.0
100/20 = 5.0
Hey how we can divide with zero..just skipping
100/5 = 20.0
Hey how we can divide with zero..just skipping
100/30 = 3.3333333333333335
```

Loops with else Block:

- Inside loop execution, if break statement not executed, then only else part will be executed.
- else means loop without break.

```
1) cart=[10,20,30,40,50]
2) for item in cart:
3)     if item>=500:
4)         print("We cannot process this order")
5)         break
6)     print(item)
7) else:
8)     print("Congrats ...all items processed successfully")
```

Output

```
10
20
30
40
50
Congrats ...all items processed successfully
```

Eg:

```
1) cart=[10,20,600,30,40,50]
2) for item in cart:
3)     if item>=500:
4)         print("We cannot process this order")
5)         break
6)     print(item)
7) else:
8)     print("Congrats ...all items processed successfully")
```

Output

```
D:\Python_classes>py test.py
10
20
We cannot process this order
```

Q) What is the difference between for loop and while loop in Python?

- We can use loops to repeat code execution
- Repeat code for every item in sequence → for loop
- Repeat code as long as condition is true → while loop

Q) How to exit from the loop? By using break statement

Q)How to skip some iterations inside loop? By using continue statement.

Q)When else part will be executed wrt loops? If loop executed without break

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:
```
if True:
    SyntaxError: unexpected EOF while parsing
    if True: pass → valid

def m1():
SyntaxError: unexpected EOF while parsing

 def m1(): pass
```

Use Case of pass:

Sometimes in the parent class we have to declare a function with empty body and child class responsible to provide proper implementation. Such type of empty body we can define by using pass keyword. (It is something like abstract method in Java)

Eg: def m1(): pass

```
1) for i in range(100):
2)     if i%9==0:
3)         print(i)
4)     else:pass
```

```
D:\Python_classes>py test.py
0
9
18
27
36
45
54
63
72
81
90
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.

```
1) x = 10
2) print(x)
3) del x
```

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

```
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.

```
1) s = "durga"
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 = "durga"
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 = "durga"
2) s = None
3) print(s) → None
```

Learn Complete Python In Simple Way

Topic: Python Pattern Programs

Pattern-1: To print given number of *s in a row

test.py

```
1) n=int(input('Enter n value:'))
2) for i in range(n):
3)     print('*',end=' ')
```

Output:

```
Enter n value:5
* * * * *
```

Pattern-2: To print square pattern with * symbols

test.py

```
1) n=int(input('Enter No Of Rows:'))
2) for i in range(n):
3)     print('* '*n)
```

Output:

```
Enter No Of Rows:5
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Pattern-3: To print square pattern with provided fixed digit in every row

test.py

```
1) n=int(input('Enter No Of Rows:'))
2) for i in range(n):
3)     print((str(i+1)+' ')*n)
```

Output:

```
Enter No Of Rows:5
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Pattern-4: To print square pattern with alphabet symbols

test.py

```
1) n=int(input('Enter No Of Rows:'))
2) for i in range(n):
3)     print((chr(65+i)+' ')*n)
```

Output:

```
Enter No Of Rows:5
A A A A A
B B B B B
C C C C C
D D D D D
E E E E E
```

Pattern-5: To print Right Angle Triangle pattern with * symbols

test.py

```
1) n=int(input('Enter No Of Rows:'))
2) for i in range(n):
3)     for j in range(i+1):
4)         print('*',end=' ')
5)     print()
```

Output:

```
Enter No Of Rows:5
*
* *
* * *
```

```
* * * *
```

Pattern-6: To print Inverted Right Angle Triangle pattern with * symbols

test.py

```
1)  n=int(input('Enter No Of Rows:'))
2)  for i in range(n):
3)      print('* '*(n-i))
```

Output:

```
Enter No Of Rows:5
* * * * *
* * * *
* * *
* *
*
```

Pattern-7: To print Pyramid pattern with * symbols

test.py

```
1)  n=int(input('Enter Number of rows:'))
2)  for i in range(n):# 0,1,2,3
3)      print((' '*(n-i-1))+ ('* ')*(i+1))
```

Output:

```
Enter number of rows:5
    *
   * *
  * * *
 * * * *
* * * * *
```

Pattern-8: To print Inverted Pyramid Pattern with * symbols

test.py

```
1) n=int(input('Enter Number of Rows:'))
2) for i in range(n): #0,1,2,3
3)     print(' '*i+'* '*(n-i))
```

Output:

```
Enter Number of Rows:5
* * * * *
 * * * *
  * * *
   * *
    *
```

Pattern-9: To print Diamond Pattern with * symbols

test.py

```
1) n=int(input('Enter n Value:'))
2) for i in range(n):#0,1,2,3
3)     print(' '*(n-i-1)+'* '*(i+1))
4) for i in range(n-1):#0,1,2
5)     print(' '*(i+1)+'* '*(n-i-1))
```

Output:

```
Enter n Value:5
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
```

Learn Complete Python In Simple Way

STRING DATA TYPE

STUDY MATERIAL

The most commonly used object in any project and in any programming language is String only. Hence we should aware complete information about String data type.

What is String?

Any sequence of characters within either single quotes or double quotes is considered as a String.

Syntax:

```
s = 'durga'
s = "durga"
```

Note: In most of other languges like C, C++, Java, a single character with in single quotes is treated as char data type value. But in Python we are not having char data type.Hence it is treated as String only.

Eg:

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

How to define multi-line String Literals?

We can define multi-line String literals by using triple single or double quotes.

Eg:

```
>>> s = '''durga
software
solutions'''
```

We can also use triple quotes to use single quotes or double quotes as symbol inside String literal.

1) s = 'This is ' single quote symbol' → Invalid
2) s = 'This is \' single quote symbol' → Valid
3) s = "This is ' single quote symbol" → Valid
4) s = 'This is " double quotes symbol' → Valid
5) s = 'The "Python Notes" by 'durga' is very helpful' → Invalid
6) s = "The "Python Notes" by 'durga' is very helpful" → Invalid
7) s = 'The \"Python Notes\" by \'durga\' is very helpful' → Valid
8) s = '''The "Python Notes" by 'durga' is very helpful''' → Valid

How to Access Characters of a String?

We can access characters of a string by using the following ways.

1) By using index
2) By using slice operator

1) Accessing Characters By using Index:

- Python supports both +ve and -ve Index.
- +ve Index means Left to Right (Forward Direction)
- -ve Index means Right to Left (Backward Direction)

Eg: s = 'durga'

```
1) >>> s='durga'
2) >>> s[0]
3) 'd'
4) >>> s[4]
5) 'a'
6) >>> s[-1]
7) 'a'
8) >>> s[10]
9) IndexError: string index out of range
```

Note: If we are trying to access characters of a string with out of range index then we will get error saying: IndexError

Q) Write a Program to Accept some String from the Keyboard and display its Characters by Index wise (both Positive and Negative Index)

test.py:

```
1) s=input("Enter Some String:")
2) i=0
3) for x in s:
4)     print("The character present at positive index {} and at nEgative index {}  is {}".fo
   rmat(i,i-len(s),x))
5)     i=i+1
```

Output:
```
D:\python_classes>py test.py
Enter Some String:durga
The character present at positive index 0 and at nEgative index -5  is d
The character present at positive index 1 and at nEgative index -4  is u
The character present at positive index 2 and at nEgative index -3  is r
The character present at positive index 3 and at nEgative index -2  is g
The character present at positive index 4 and at nEgative index -1  is a
```

2) Accessing Characters by using Slice Operator:

- **Syntax:** s[bEginindex:endindex:step]

- Begin Index: From where we have to consider slice (substring)
- End Index: We have to terminate the slice (substring) at endindex-1
- Step: Incremented Value.

Note:

- If we are not specifying bEgin index then it will consider from bEginning of the string.
- If we are not specifying end index then it will consider up to end of the string.
- The default value for step is 1.

```
1)  >>> s="Learning Python is very very easy!!!"
2)  >>> s[1:7:1]
3)  'earnin'
4)  >>> s[1:7]
5)  'earnin'
6)  >>> s[1:7:2]
7)  'eri'
8)  >>> s[:7]
9)  'Learnin'
10) >>> s[7:]
11) 'g Python is very very easy!!!'
12) >>> s[::]
13) 'Learning Python is very very easy!!!'
14) >>> s[:]
15) 'Learning Python is very very easy!!!'
16) >>> s[::-1]
17) '!!!ysae yrev yrev si nohtyP gninraeL'
```

Behaviour of Slice Operator:

1) s[bEgin:end:step]
2) Step value can be either +ve or –ve
3) If +ve then it should be forward direction(left to right) and we have to consider bEgin to end-1
4) If -ve then it should be backward direction (right to left) and we have to consider bEgin to end+1.

***Note:**

- In the backward direction if end value is -1 then result is always empty.
- In the forward direction if end value is 0 then result is always empty.

In Forward Direction:

default value for bEgin: 0
default value for end: length of string
default value for step: +1

In Backward Direction:

default value for bEgin: -1
default value for end: -(length of string+1)

Note: Either forward or backward direction, we can take both +ve and -ve values for bEgin and end index.

Slice Operator Case Study:

1) **S = 'abcdefghij'**
2) **s[1:6:2] → 'bdf'**
3) **s[::1] → 'abcdefghij'**
4) **s[::-1] → 'jihgfedcba'**
5) **s[3:7:-1] → ''**
6) **s[7:4:-1] → 'hgf'**
7) **s[0:10000:1] → 'abcdefghij'**
8) **s[-4:1:-1] → 'gfedc'**
9) **s[-4:1:-2] → 'gec'**
10) **s[5:0:1] → ''**
11) **s[9:0:0] → ValueError: slice step cannot be zero**
12) **s[0:-10:-1] → ''**
13) **s[0:-11:-1] → 'a'**
14) **s[0:0:1] → ''**
15) **s[0:-9:-2] → ''**
16) **s[-5:-9:-2] → 'fd'**
17) **s[10:-1:-1] → ''**
18) **s[10000:2:-1] → 'jihgfed'**

Note: Slice operator never raises IndexError

Mathematical Operators for String:

We can apply the following mathematical operators for Strings.

1) **+ operator for concatenation**
2) *** operator for repetition**

- **print("durga"+"soft") → durgasoft**
- **print("durga"*2) → durgadurga**

Note:

1) To use + operator for Strings, compulsory both arguments should be str type.
2) To use * operator for Strings, compulsory one argument should be str and other argument should be int.

len() in-built Function:

We can use len() function to find the number of characters present in the string.

Eg:

```
s = 'durga'
print(len(s)) → 5
```

Q) Write a Program to access each Character of String in Forward and Backward Direction by using while Loop?

```
1)  s = "Learning Python is very easy !!!"
2)  n = len(s)
3)  i = 0
4)  print("Forward direction")
5)  while i<n:
6)      print(s[i],end=' ')
7)      i +=1
8)  print("Backward direction")
9)  i = -1
10) while i >= -n:
11)     print(s[i],end=' ')
12)     i = i-1
```

Alternative ways:

```
1)  s = "Learning Python is very easy !!!"
2)  print("Forward direction")
3)  for i in s:
4)      print(i,end=' ')
5)  print("Forward direction")
6)  for i in s[::]:
7)      print(i,end=' ')
8)
9)  print("Backward direction")
10) for i in s[::-1]:
11)     print(i,end=' ')
```

Checking Membership:

We can check whether the character or string is the member of another string or not by using in and not in operators

```
s = 'durga'
print('d' in s) → True
print('z' in s) → False
```

```
1) s = input("Enter main string:")
2) subs = input("Enter sub string:")
3) if subs in s:
4)     print(subs,"is found in main string")
5) else:
6)     print(subs,"is not found in main string")
```

Output:

```
D:\python_classes>py test.py
Enter main string:durgasoftwaresolutions
Enter sub string:durga
durga is found in main string

D:\python_classes>py test.py
Enter main string:durgasoftwaresolutions
Enter sub string:python
python is not found in main string
```

Comparison of Strings:

- **We can use comparison operators (<, <=, >, >=) and equality operators (==, !=) for strings.**
- **Comparison will be performed based on alphabetical order.**

```
1) s1=input("Enter first string:")
2) s2=input("Enter Second string:")
3) if s1==s2:
4)     print("Both strings are equal")
5) elif s1<s2:
6)     print("First String is less than Second String")
7) else:
8)     print("First String is greater than Second String")
```

Output:

```
D:\python_classes>py test.py
Enter first string:durga
```

Enter Second string:durga
Both strings are equal

D:\python_classes>py test.py
Enter first string:durga
Enter Second string:ravi
First String is less than Second String

D:\python_classes>py test.py
Enter first string:durga
Enter Second string:anil
First String is greater than Second String

Removing Spaces from the String:

We can use the following 3 methods

1) rstrip() → To remove spaces at right hand side
2) lstrip() →To remove spaces at left hand side
3) strip() → To remove spaces both sides

```
1)  city=input("Enter your city Name:")
2)  scity=city.strip()
3)  if scity=='Hyderabad':
4)      print("Hello Hyderbadi..Adab")
5)  elif scity=='Chennai':
6)      print("Hello Madrasi...Vanakkam")
7)  elif scity=="Bangalore":
8)     print("Hello Kannadiga...Shubhodaya")
9)  else:
10)    print("your entered city is invalid")
```

Finding Substrings:

We can use the following 4 methods

For forward direction:

1) find()
2) index()

For backward direction:

1) rfind()
2) rindex()

find():

s.find(substring)

Returns index of first occurrence of the given substring. If it is not available then we will get -1.

```
1) s="Learning Python is very easy"
2) print(s.find("Python")) #9
3) print(s.find("Java"))   # -1
4) print(s.find("r"))#3
5) print(s.rfind("r"))#21
```

Note: **By default find() method can search total string. We can also specify the boundaries to search.**

s.find(substring,bEgin,end)

It will always search from bEgin index to end-1 index.

```
1) s="durgaravipavanshiva"
2) print(s.find('a'))#4
3) print(s.find('a',7,15))#10
4) print(s.find('z',7,15))#-1
```

index():

index() method is exactly same as find() method except that if the specified substring is not available then we will get ValueError.

```
1) s=input("Enter main string:")
2) subs=input("Enter sub string:")
3) try:
4)     n=s.index(subs)
5) except ValueError:
6)     print("substring not found")
7) else:
8)     print("substring found")
```

Output:

```
D:\python_classes>py test.py
Enter main string:learning python is very easy
Enter sub string:python
substring found
```

```
D:\python_classes>py test.py
Enter main string:learning python is very easy
Enter sub string:java
substring not found
```

Q) Program to display all Positions of Substring in a given Main String

```
1)  s=input("Enter main string:")
2)  subs=input("Enter sub string:")
3)  flag=False
4)  pos=-1
5)  n=len(s)
6)  while True:
7)      pos=s.find(subs,pos+1,n)
8)      if pos==-1:
9)          break
10)     print("Found at position",pos)
11)     flag=True
12) if flag==False:
13)     print("Not Found")
```

Output:

```
D:\python_classes>py test.py
Enter main string:abbababababacdefg
Enter sub string:a
Found at position 0
Found at position 3
Found at position 5
Found at position 7
Found at position 9
Found at position 11

D:\python_classes>py test.py
Enter main string:abbababababacdefg
Enter sub string:bb
Found at position 1
```

Counting substring in the given String:

We can find the number of occurrences of substring present in the given string by using count() method.

1) s.count(substring) → It will search through out the string.
2) s.count(substring, bEgin, end) → It will search from bEgin index to end-1 index.

```
1) s="abcabcabcabcadda"
2) print(s.count('a'))
3) print(s.count('ab'))
4) print(s.count('a',3,7))
```

Output:

6
4
2

Replacing a String with another String:

s.replace(oldstring, newstring)

inside s, every occurrence of old String will be replaced with new String.

Eg 1:

```
s = "Learning Python is very difficult"
s1 = s.replace("difficult","easy")
print(s1)
```

Output: Learning Python is very easy

Eg 2: All occurrences will be replaced

```
s = "abababababababab"
s1 = s.replace("a","b")
print(s1)
```

Output: bbbbbbbbbbbbbb

Q) String Objects are Immutable then how we can change the Content by using replace() Method

- Once we creates string object, we cannot change the content.This non changeable behaviour is nothing but immutability. If we are trying to change the content by using any method, then with those changes a new object will be created and changes won't be happend in existing object.
- Hence with replace() method also a new object got created but existing object won't be changed.

Eg:

```
s = "abab"
s1 = s.replace("a","b")
print(s,"is available at :",id(s))
print(s1,"is available at :",id(s1))
```

Output:

abab is available at : 4568672
bbbb is available at : 4568704

In the above example, original object is available and we can see new object which was created because of replace() method.

Splitting of Strings:

- We can split the given string according to specified seperator by using split() method.
- l = s.split(seperator)
- The default seperator is space. The return type of split() method is List.

```
1) s="durga software solutions"
2) l=s.split()
3) for x in l:
4)         print(x)
```

Output:

durga
software
solutions

```
1) s="22-02-2018"
2) l=s.split('-')
3) for x in l:
4)     print(x)
```

Output:

22
02
2018

Joining of Strings:

We can join a Group of Strings (List OR Tuple) wrt the given Seperator.
s = seperator.join(group of strings)

Eg 1:

```
t = ('sunny', 'bunny', 'chinny')
s = '-'.join(t)
print(s)
```

Output: sunny-bunny-chinny

Eg 2:

```
l = ['hyderabad', 'singapore', 'london', 'dubai']
s = ':'.join(l)
print(s)
```

Output: hyderabad:singapore:london:dubai

Changing Case of a String:

We can change case of a string by using the following 4 methods.

1) upper() → To convert all characters to upper case
2) lower() → To convert all characters to lower case
3) swapcase() → Converts all lower case characters to upper case and all upper case characters to lower case
4) title() → To convert all character to title case. i.e first character in every word should be upper case and all remaining characters should be in lower case.
5) capitalize() → Only first character will be converted to upper case and all remaining characters can be converted to lower case

```
1) s = 'learning Python is very Easy'
2) print(s.upper())
3) print(s.lower())
4) print(s.swapcase())
5) print(s.title())
6) print(s.capitalize())
```

Output:

```
LEARNING PYTHON IS VERY EASY
learning python is very easy
LEARNING pYTHON IS VERY eASY
Learning Python Is Very Easy
Learning python is very easy
```

Checking Starting and Ending Part of the String:

Python contains the following methods for this purpose

1) s.startswith(substring)
2) s.endswith(substring)

```
1) s = 'learning Python is very easy'
2) print(s.startswith('learning'))
3) print(s.endswith('learning'))
4) print(s.endswith('easy'))
```

Output:

```
True
False
True
```

To Check Type of Characters Present in a String:

Python contains the following methods for this purpose.

1) **isalnum():** Returns True if all characters are alphanumeric(a to z , A to Z ,0 to9)
2) **isalpha():** Returns True if all characters are only alphabet symbols(a to z,A to Z)
3) **isdigit():** Returns True if all characters are digits only(0 to 9)
4) **islower():** Returns True if all characters are lower case alphabet symbols
5) **isupper():** Returns True if all characters are upper case aplhabet symbols
6) **istitle():** Returns True if string is in title case
7) **isspace():** Returns True if string contains only spaces

Eg:

1) print('Durga786'.isalnum()) → True
2) print('durga786'.isalpha()) → False
3) print('durga'.isalpha()) → True
4) print('durga'.isdigit()) → False
5) print('786786'.isdigit()) → True
6) print('abc'.islower()) → True
7) print('Abc'.islower()) → False
8) print('abc123'.islower()) → True
9) print('ABC'.isupper()) → True
10) print('Learning python is Easy'.istitle()) → False
11) print('Learning Python Is Easy'.istitle()) → True
12) print(' '.isspace()) → True

Demo Program:

```
1)  s=input("Enter any character:")
2)  if s.isalnum():
3)      print("Alpha Numeric Character")
4)      if s.isalpha():
5)          print("Alphabet character")
6)          if s.islower():
7)              print("Lower case alphabet character")
8)          else:
9)              print("Upper case alphabet character")
10)     else:
11)         print("it is a digit")
12) elif s.isspace():
```

```
13)    print("It is space character")
14) else:
15)    print("Non Space Special Character")
```

```
D:\python_classes>py test.py
Enter any character:7
Alpha Numeric Character
it is a digit

D:\python_classes>py test.py
Enter any character:a
Alpha Numeric Character
Alphabet character
Lower case alphabet character

D:\python_classes>py test.py
Enter any character:$
Non Space Special Character

D:\python_classes>py test.py
Enter any character:A
Alpha Numeric Character
Alphabet character
Upper case alphabet character
```

Formatting the Strings:

We can format the strings with variable values by using replacement operator {} and format() method.

```
1)  name = 'durga'
2)  salary = 10000
3)  age = 48
4)  print("{} 's salary is {} and his age is {}".format(name,salary,age))
5)  print("{0} 's salary is {1} and his age is {2}".format(name,salary,age))
6)  print("{x} 's salary is {y} and his age is {z}".format(z=age,y=salary,x=name))
```

Output:

```
durga 's salary is 10000 and his age is 48
durga 's salary is 10000 and his age is 48
durga 's salary is 10000 and his age is 48
```

Important Programs regarding String Concept

Q1) Write a Program to Reverse the given String

Input: durga
Output: agrud

1st Way:

```
1) s = input("Enter Some String:")
2) print(s[::-1])
```

2nd Way:

```
1) s = input("Enter Some String:")
2) print(''.join(reversed(s)))
```

3rd Way:

```
1) s = input("Enter Some String:")
2) i=len(s)-1
3) target=''
4) while i>=0:
5)     target=target+s[i]
6)     i=i-1
7) print(target)
```

Q2) Program to Reverse Order of Words

Input: Learning Python is very Easy
Output: Easy Very is Python Learning

```
1) s=input("Enter Some String:")
2) l=s.split()
3) l1=[]
4) i=len(l)-1
5) while i>=0:
6)     l1.append(l[i])
7)     i=i-1
8) output=' '.join(l1)
9) print(output)
```

Output: Enter Some String:Learning Python is very easy!!
easy!!! very is Python Learning

Q3) Program to Reverse Internal Content of each Word

Input: Durga Software Solutions
Output: agruD erawtfoS snoituloS

```
1)  s=input("Enter Some String:")
2)  l=s.split()
3)  l1=[]
4)  i=0
5)  while i<len(l):
6)      l1.append(l[i][::-1])
7)      i=i+1
8)  output=' '.join(l1)
9)  print(output)
```

Q4) Write a Program to Print Characters at Odd Position and Even Position for the given String?

1st Way:

```
s = input("Enter Some String:")
print("Characters at Even Position:",s[0::2])
print("Characters at Odd Position:",s[1::2])
```

2nd Way:

```
1)  s=input("Enter Some String:")
2)  i=0
3)  print("Characters at Even Position:")
4)  while i< len(s):
5)      print(s[i],end=',')
6)      i=i+2
7)  print()
8)  print("Characters at Odd Position:")
9)  i=1
10) while i< len(s):
11)     print(s[i],end=',')
12)     i=i+2
```

Q5) Program to Merge Characters of 2 Strings into a Single String by taking Characters alternatively

Input: s1 = "ravi"
s2 = "reja"
Output: rtaevjia

```
1)  s1=input("Enter First String:")
2)  s2=input("Enter Second String:")
3)  output=''
4)  i,j=0,0
5)  while i<len(s1) or j<len(s2):
6)      if i<len(s1):
7)          output=output+s1[i]
8)          i+=1
9)      if j<len(s2):
10)         output=output+s2[j]
11)         j+=1
12) print(output)
```

Output:

```
Enter First String:durga
Enter Second String:ravisoft
druarvgiasoft
```

Q6) Write a Program to Sort the Characters of the String and First Alphabet Symbols followed by Numeric Values

Input: B4A1D3
Output: ABD134

```
1)  s=input("Enter Some String:")
2)  s1=s2=output=''
3)  for x in s:
4)      if x.isalpha():
5)          s1=s1+x
6)      else:
7)          s2=s2+x
8)  for x in sorted(s1):
9)      output=output+x
10) for x in sorted(s2):
11)     output=output+x
12) print(output)
```

Q7) Write a Program for the following Requirement

Input: a4b3c2
Output: aaaabbbcc

```
1) s=input("Enter Some String:")
2) output=''
3) for x in s:
4)     if x.isalpha():
5)         output=output+x
6)         previous=x
7)     else:
8)         output=output+previous*(int(x)-1)
9) print(output)
```

Note: chr(unicode) → The corresponding character
ord(character) → The corresponding unicode value

Q8) Write a Program to perform the following Activity

Input: a4k3b2
Outpt: aeknbd

```
1) s=input("Enter Some String:")
2) output=''
3) for x in s:
4)     if x.isalpha():
5)         output=output+x
6)         previous=x
7)     else:
8)         output=output+chr(ord(previous)+int(x))
9) print(output)
```

Q9) Write a Program to Remove Duplicate Characters from the given Input String?

Input: ABCDABBCDABBBCCCDDEEEF
Output: ABCDEF

```
1) s = input("Enter Some String:")
2) l=[]
3) for x in s:
4)     if x not in l:
5)         l.append(x)
6) output=''.join(l)
7) print(output)
```

Q10) Write a Program to find the Number of Occurrences of each Character present in the given String?

Input: ABCABCABBCDE
Output: A-3,B-4,C-3,D-1,E-1

```
1)  s=input("Enter the Some String:")
2)  d={}
3)  for x in s:
4)      if x in d.keys():
5)          d[x]=d[x]+1
6)      else:
7)          d[x]=1
8)  for k,v in d.items():
9)      print("{} = {} Times".format(k,v))
```

Q11) Write a Program to perform the following Task?

Input: 'one two three four five six seven'
Output: 'one owt three ruof five xis seven'

```
1)  s = input('Enter Some String:')
2)  l = s.split()
3)  l1 = []
4)  i = 0
5)  while i<len(l):
6)      if i%2==0:
7)          l1.append(l[i])
8)      else:
9)          l1.append(l[i][::-1])
10)     i=i+1
11) output=' '.join(l1)
12) print('Original String:',s)
13) print('output String:',output)
```

Output:

```
D:\durgaclasses>py test.py
Enter Some String:one two three four five six seven
Original String: one two three four five six seven
output String: one owt three ruof five xis seven
```

Formatting the Strings:

- We can format the strings with variable values by using replacement operator {} and format() method.
- The main objective of format() method to format string into meaningful output form.

Case- 1: Basic formatting for default, positional and keyword arguments

```
1) name = 'durga'
2) salary = 10000
3) age = 48
4) print("{} 's salary is {} and his age is {}".format(name,salary,age))
5) print("{0} 's salary is {1} and his age is {2}".format(name,salary,age))
6) print("{x} 's salary is {y} and his age is {z}".format(z=age,y=salary,x=name))
```

Output:

```
durga 's salary is 10000 and his age is 48
durga 's salary is 10000 and his age is 48
durga 's salary is 10000 and his age is 48
```

Case-2: Formatting Numbers

d → Decimal IntEger
f → Fixed point number(float).The default precision is 6
b → Binary format
o → Octal Format
x → Hexa Decimal Format (Lower case)
X → Hexa Decimal Format (Upper case)

Eg-1:

```
1) print("The intEger number is: {}".format(123))
2) print("The intEger number is: {:d}".format(123))
3) print("The intEger number is: {:5d}".format(123))
4) print("The intEger number is: {:05d}".format(123))
```

Output:

```
The intEger number is: 123
The intEger number is: 123
The intEger number is:   123
The intEger number is: 00123
```

Eg-2:

```
1) print("The float number is: {}".format(123.4567))
2) print("The float number is: {:f}".format(123.4567))
3) print("The float number is: {:8.3f}".format(123.4567))
4) print("The float number is: {:08.3f}".format(123.4567))
5) print("The float number is: {:08.3f}".format(123.45))
6) print("The float number is: {:08.3f}".format(786786123.45))
```

Output:

```
The float number is: 123.4567
The float number is: 123.456700
The float number is:  123.457
The float number is: 0123.457
The float number is: 0123.450
The float number is: 786786123.450
```

Note:

- {:08.3f}
- Total positions should be minimum 8.
- After decimal point exactly 3 digits are allowed.If it is less then 0s will be placed in the last positions
- If total number is < 8 positions then 0 will be placed in MSBs
- If total number is >8 positions then all intEgral digits will be considered.
- The extra digits we can take only 0

Note: For numbers default alignment is Right Alignment (>)

Eg-3: Print Decimal value in binary, octal and hexadecimal form

```
1) print("Binary Form:{0:b}".format(153))
2) print("Octal Form:{0:o}".format(153))
3) print("Hexa decimal Form:{0:x}".format(154))
4) print("Hexa decimal Form:{0:X}".format(154))
```

Output:

```
Binary Form:10011001
Octal Form:231
Hexa decimal Form:9a
Hexa decimal Form:9A
```

Note: We can represent only int values in binary, octal and hexadecimal and it is not possible for float values.

Note:

1) {:5d} It takes an intEger argument and assigns a minimum width of 5.
2) {:8.3f} It takes a float argument and assigns a minimum width of 8 including "." and after decimal point excatly 3 digits are allowed with round operation if required
3) {:05d} The blank places can be filled with 0. In this place only 0 allowed.

Case-3: Number formatting for signed numbers

- While displaying positive numbers,if we want to include + then we have to write {:+d} and {:+f}
- Using plus for -ve numbers there is no use and for -ve numbers - sign will come automatically.

```
1) print("int value with sign:{:+d}".format(123))
2) print("int value with sign:{:+d}".format(-123))
3) print("float value with sign:{:+f}".format(123.456))
4) print("float value with sign:{:+f}".format(-123.456))
```

Output:

```
int value with sign:+123
int value with sign:-123
float value with sign:+123.456000
float value with sign:-123.456000
```

Case-4: Number formatting with alignment

- <, >, ^ and = are used for alignment
- < → Left Alignment to the remaining space
- ^ → Center alignment to the remaining space
- > → Right alignment to the remaining space
- = → Forces the signed(+) (-) to the left most position

Note: Default Alignment for numbers is Right Alignment.

Ex:

```
1) print("{:5d}".format(12))
2) print("{:<5d}".format(12))
3) print("{:<05d}".format(12))
4) print("{:>5d}".format(12))
5) print("{:>05d}".format(12))
6) print("{:^5d}".format(12))
7) print("{:=5d}".format(-12))
8) print("{:^10.3f}".format(12.23456))
9) print("{:=8.3f}".format(-12.23456))
```

Output:

```
12
12
12000
12
00012
12
-12
12.235
- 12.235
```

Case-5: String formatting with format()

Similar to numbers, we can format String values also with format() method.

s.format(string)

```
1) print("{:5d}".format(12))
2) print("{:5}".format("rat"))
3) print("{:>5}".format("rat"))
4) print("{:<5}".format("rat"))
5) print("{:^5}".format("rat"))
6) print("{:*^5}".format("rat")) #Instead of * we can use any character(like +,$,a etc)
```

Output:

```
12
rat
rat
rat
rat
*rat*
```

Note: For numbers default alignment is right where as for strings default alignment is left

Case-6: Truncating Strings with format() method

```
1) print("{:.3}".format("durgasoftware"))
2) print("{:5.3}".format("durgasoftware"))
3) print("{:>5.3}".format("durgasoftware"))
4) print("{:^5.3}".format("durgasoftware"))
5) print("{:*^5.3}".format("durgasoftware"))
```

Output:

```
dur
dur
dur
```

dur
dur

Case-7: Formatting dictionary members using format()

```
1) person={'age':48,'name':'durga'}
2) print("{p[name]}'s age is: {p[age]}".format(p=person))
```

Output:
durga's age is: 48
Note: p is alias name of dictionary
person dictionary we are passing as keyword argument

More convinient way is to use **person

```
1) person={'age':48,'name':'durga'}
2) print("{name}'s age is: {age}".format(**person))
```

Output: durga's age is: 48

Case-8: Formatting class members using format()

```
1) class Person:
2)     age=48
3)     name="durga"
4) print("{p.name}'s age is :{p.age}".format(p=Person()))
```

Output: durga's age is :48

```
1) class Person:
2)     def __init__(self,name,age):
3)         self.name=name
4)         self.age=age
5) print("{p.name}'s age is :{p.age}".format(p=Person('durga',48)))
6) print("{p.name}'s age is :{p.age}".format(p=Person('Ravi',50)))
```

Note: Here Person object is passed as keyword argument. We can access by using its reference variable in the template string

Case-9: Dynamic Formatting using format()

```
1) string="{:{fill}{align}{width}}"
2) print(string.format('cat',fill='*',align='^',width=5))
3) print(string.format('cat',fill='*',align='^',width=6))
```

```
4) print(string.format('cat',fill='*',align='<',width=6))
5) print(string.format('cat',fill='*',align='>',width=6))
```

Output:

```
*cat*
*cat**
cat***
***cat
```

Case-10: Dynamic Float format template

```
1) num="{:{align}{width}.{precision}f}"
2) print(num.format(123.236,align='<',width=8,precision=2))
3) print(num.format(123.236,align='>',width=8,precision=2))
```

Output:

```
123.24
123.24
```

Case-11: Formatting Date values

```
1) import datetime
2) #datetime formatting
3) date=datetime.datetime.now()
4) print("It's now:{:%d/%m/%Y  %H:%M:%S}".format(date))
```

Output: It's now:09/03/2018 12:36:26

Case-12: Formatting complex numbers

```
1) complexNumber=1+2j
2) print("Real Part:{0.real} and Imaginary Part:{0.imag}".format(complexNumber))
```

Output: Real Part: 1.0 and Imaginary Part: 2.0

Learn Complete Python In Simple Way

Topic

String Coding Interview Questions

Q1) Write a Program To REVERSE content of the given String by using slice operator?

```
input: durga
output: agrud

s = input('Enter Some String to Reverse:')
output = s[::-1]
print(output)
```

Q2) Write a Program To REVERSE content of the given String by using reversed() function?

```
input: durga
output: agrud

s=input('Enter Some String to Reverse:')
r=reversed(s)
output=''.join(r)
print(output)
```

Q3) Write a Program To REVERSE content of the given String by using while loop?

```
input: durga
output: agrud

s=input('Enter Some String to Reverse:')
output=''
i=len(s)-1
while i>=0:
    output=output+s[i]
    i=i-1
print(output)
```

Q4) Write a Program To REVERSE order of words present in the given string?

```
input: Learning Python Is Very Easy
output: Easy Very Is Python Learning

s=input('Enter Some String:')
l=s.split()
l1=l[::-1]
output=' '.join(l1)
print(output)
```

Q5) Write a Program To REVERSE internal content of each word?

```
input: 'Durga Software Solutions'
output: 'agruD erawtfoS snoituloS'

s=input('Enter Any String:')
l=s.split()
l1=[]
for word in l:
    l1.append(word[::-1])
output=' '.join(l1)
print(output)
```

Q6) Write a Program To REVERSE internal content of every second word present in the given string?

```
i/p: one two three four five six
o/p: one owt three ruof five xis

s='one two three four five six'
l=s.split()
l1=[]
i=0
while i<len(l):
    if i%2 == 0:
        l1.append(l[i])
    else:
```

```
12)       l1.append(l[i][::-1])
13)    i=i+1
14) output=' '.join(l1)
15) print(output)
```

Q7) Write a program to print the characters present at even index and odd index seperately for the given string?

1st Way:

```
1)  s=input('Enter Input String:')
2)  print('Characters present at Even Index:')
3)  i=0
4)  while i<len(s):
5)     print(s[i])
6)     i=i+2
7)  print('Characters present at Odd Index:')
8)  i=1
9)  while i<len(s):
10)    print(s[i])
11)    i=i+2
```

Output:

```
D:\durgaclasses>py test.py
Enter Input String:durgasoftware
Characters present at Even Index:
d
r
a
o
t
a
e
Characters present at Odd Index:
u
g
s
f
w
r
```

2nd Way:

```
1) s=input('Enter Input String:')
2) print('Characters present at Even Index:',s[0::2])
3) print('Characters present at Even Index:',s[::2])
4) print('Characters present at Odd Index:',s[1::2])
```

Q8) Write a program to merge characters of 2 strings into a single string by taking characters alternatively?

Input:

```
s1='RAVI'
s2='TEJA'
```

Output: RTAEVJIA

If strings are having same length:

```
1) s1='RAVI'
2) s2='TEJA'
3) output=''
4) i,j=0,0
5) while i<len(s1) or j<len(s2):
6)     output=output+s1[i]+s2[j]
7)     i=i+1
8)     j=j+1
9) print(output)
```

Output: RTAEVJIA

2nd way by using map():

```
1) s1='RAVI'
2) s2='TEJA'
3) l=list(map(lambda x,y:x+y,s1,s2))
4) print(''.join(l))
```

Note: The above program can work if the lengths of 2 strings are same.

If strings having different lengths:

```
1)  s1=input('Enter First String:')
2)  s2=input('Enter Second String:')
3)  output=''
4)  i,j=0,0
5)  while i<len(s1) or j<len(s2):
6)      if i<len(s1):
7)          output=output+s1[i]
8)          i=i+1
9)      if j<len(s2):
10)         output=output+s2[j]
11)         j=j+1
12) print(output)
```

Output:

```
D:\durgaclasses>py test.py
Enter First String:RAVIKIRAN
Enter Second String:TEJA
RTAEVJIAKIRAN

D:\durgaclasses>py test.py
Enter First String:RAVI
Enter Second String:TEJAKIRAN
RTAEVJIAKIRAN
```

Q9) Assume input string contains only alphabet symbols and digits. Write a program to sort characters of the string, first alphabet symbols followed by digits?

```
1)  input: B4A1D3
2)  output: ABD134
3)
4)  s='B4A1D3'
5)  alphabets=[]
6)  digits=[]
7)  for ch in s:
8)      if ch.isalpha():
9)          alphabets.append(ch)
10)     else:
```

```
        digits.append(ch)
output=''.join(sorted(alphabets)+sorted(digits))
print(output)
```

Alternative way:

```
s='B4A1D3'
alphabets=''
digits=''
for ch in s:
    if ch.isalpha():
        alphabets+=ch
    else:
        digits+=ch
output=''
for ch in sorted(alphabets):
    output=output+ch
for ch in sorted(digits):
    output=output+ch
print(output)
```

Q10) Write a program for the following requirement?

```
input: a4b3c2
output: aaaabbbcc

s=input('Enter Some String where alphabet symbol should be followed by digit:')
output=''
for ch in s:
    if ch.isalpha():
        x=ch
    else:
        d=int(ch)
        output=output+x*d
print(output)
```

Q11) Write a program for the following requirement?

```
1) input: a3z2b4
2) output: aaabbbbzz  (sorted String)
3)
4) s=input('Enter Some String where alphabet symbol should be followed by digit:')
5) target=''
6) for ch in s:
7)     if ch.isalpha():
8)         x=ch
9)     else:
10)        d=int(ch)
11)        target=target+x*d
12) output = ''.join(sorted(target))
13) print(output)
```

Q12) Write a program for the following requirement?

```
1) input: aaaabbbccz
2) output: 4a3b2c1z
3)
4) s='aaaabbbccz'
5) output=''
6) previous=s[0]
7) c=1
8) i=1
9) while i<len(s):
10)    if s[i]==previous:
11)        c=c+1
12)    else:
13)        output=output+str(c)+previous
14)        previous=s[i]
15)        c=1
16)    if i ==len(s)-1:
17)        output=output+str(c)+previous
18)    i=i+1
19) print(output)
```

Q13) Write a program for the following requirement?

Input: a4k3b2
Output: aeknbd

In this example the following two functions are required to use

1) **ord(): To find unicode value for the given character**
   Eg: print(ord('a')) #97
2) **chr(): To find corresponding character for the given unicode value**
   Eg: print(chr(97)) # a

```
1)  s='a4k3b2'
2)  output=''
3)  for ch in s:
4)      if ch.isalpha():
5)          x=ch
6)          output=output+ch
7)      else:
8)          d=int(ch)
9)          newc= chr(ord(x)+d)
10)         output=output+newc
11) print(output)
```

Q14) Write a program to remove duplicate characters from the given input String?

Input: AZZZBCDABBCDABBBBCCCCDDDDEEEEEF
Output: AZBCDEF

1st way:

```
1)  s='AZZZBCDABBCDABBBBCCCCDDDDEEEEEF'
2)  output=''
3)  for ch in s:
4)      if ch not in output:
5)          output=output+ch
6)  print(output) # AZBCDEF
```

2nd way:

```
1) s='AZZZBCDABBCDABBBBCCCCDDDDEEEEEF'
2) l=[]
3) for ch in s:
4)     if ch not in l:
5)         l.append(ch)
6) output=''.join(l)
7) print(output) # AZBCDEF
```

3rd way by using set (but no guarantee for the order)

```
1) s='ABCDABXXXBCDABBBBCCCZZZZCDDDDEEEEEF'
2) s1=set(s)
3) output=''.join(s1)
4) print(output) #CAEZBFD
```

Q15) Write a program to find the number of occurrences of each character present in the given string?

By using count() method and List:

```
1) s='ABCDABXXXBCDABBBBCCCZZZZCDDDDEEEEEF'
2) l=[]
3) for ch in s:
4)     if ch not in l:
5)         l.append(ch)
6)
7) for ch in sorted(l):
8)     print('{} occurrs {} times'.format(ch,s.count(ch)))
```

Without using count() method:

```
1) s='ABCDABXXXBCDABBBBCCCZZZZCDDDDEEEEEF'
2) d={}
3) for ch in s:
4)     d[ch]=d.get(ch,0)+1
5) for k,v in d.items():
6)     print('{} occurrs {} times'.format(k,v))
```

For sorting purpose:

```
1) for k,v in sorted(d.items()):
2)     print('{} occurrs {} times'.format(k,v))
```

Q16) Write the program for the following requirement:

Input: ABAABBCA
Output: 4A3B1C

```
1) s='ABAABBCA'
2) output=''
3) d={}
4) for ch in s:
5)     d[ch]=d.get(ch,0)+1
6) for k,v in sorted(d.items()):
7)     output=output+str(v)+k
8) print(output)
```

Q17) Write the program for the following requirement:

Input: ABAABBCA
Output: A4B3C1

```
1) s='ABAABBCA'
2) output=''
3) d={}
4) for ch in s:
5)     d[ch]=d.get(ch,0)+1
6) for k,v in sorted(d.items()):
7)     output=output+k+str(v)
8) print(output)
```

Q18) Write a program to find the number of occurrences of each vowel present in the given string?

```
1) s=input('Enter some string to search for vowels:')
2) v=['a','e','i','o','u','A','E','I','O','U']
3) d={}
4) for ch in s:
5)     if ch in v:
```

```
6)        d[ch]=d.get(ch,0)+1
7)  for k,v in sorted(d.items()):
8)     print('{} occurrs {} times'.format(k,v))
```

```
D:\durgaclasses>py test.py
Enter some string to search for vowels:DURGASOFTWARESOLUTIONS
A occurrs 2 times
E occurrs 1 times
I occurrs 1 times
O occurrs 3 times
U occurrs 2 times

D:\durgaclasses>py test.py
Enter some string to search for vowels:mississippi
i occurrs 4 times
```

Q19) Write a program to check whether the given two strings are anagrams or not?

Two strings are said to be anagrams iff both are having same content irrespective of characters position.

Eg: lazy and zaly

```
1)  s1=input("Enter first string:")
2)  s2=input("Enter second string:")
3)  if(sorted(s1)==sorted(s2)):
4)      print("The strings are anagrams.")
5)  else:
6)      print("The strings aren't anagrams.")
```

Output:

```
D:\durgaclasses>py test.py
Enter first string:lazy
Enter second string:zaly
The strings are anagrams.

D:\durgaclasses>py test.py
Enter first string:durga
Enter second string:urgadd
The strings aren't anagrams.
```

Q20) Write a program to check whether the given string is palindrome or not ?

A string is said to be palindrome iff original string and its reversed strings are equal.

```
1) s=input("Enter Some string:")
2) if s==s[::-1]:
3)     print('The given string is palindrome')
4) else:
5)     print('The given string is not palindrome')
```

```
D:\durgaclasses>py test.py
Enter Some string:level
The given string is palindrome

D:\durgaclasses>py test.py
Enter Some string:madam
The given string is palindrome

D:\durgaclasses>py test.py
Enter Some string:apple
The given string is not palindrome
```

Q21) Write the program for the following requirement:

```
1) inputs:
2)     s1='abcdefg'
3)     s2='xyz'
4)     s3='12345'
5) output: ax1, by2,cz3,d4,e5,f,g
6)
7) s1='abcdefg'
8) s2='xyz'
9) s3='12345'
10) i=j=k=0
11) while i<len(s1) or j<len(s2) or k<len(s3):
12)     output=''
13)     if i<len(s1):
14)         output=output+s1[i]
15)         i=i+1
```

```
16)  if j<len(s2):
17)      output=output+s2[j]
18)      j=j+1
19)  if k<len(s3):
20)      output=output+s3[k]
21)      k=k+1
22)  print(output)
```

Output:

```
ax1
by2
cz3
d4
e5
f
g
```

Learn Complete Python In Simple Way

LIST

DATA STRUCTURE STUDY MATERIAL

- 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]

-6	-5	-4	-3	-2	-1
10	A	B	20	30	10
0	1	2	3	4	5

- 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))
```

```
D:\Python_classes>py test.py
Enter List:[10,20,30,40]
[10, 20, 30, 40]
<class 'list'>
```

4) With list() Function:

```
1) l=list(range(0,10,2))
2) print(l)
3) print(type(l))
```

```
D:\Python_classes>py test.py
[0, 2, 4, 6, 8]
<class 'list'>
```

Eg:

```
1) s="durga"
2) l=list(s)
3) print(l)
```

```
D:\Python_classes>py test.py
['d', 'u', 'r', 'g', 'a']
```

5) With split() Function:

```
1) s="Learning Python is very very easy !!!"
2) l=s.split()
3) print(l)
4) print(type(l))
```

```
D:\Python_classes>py test.py
['Learning', 'Python', 'is', 'very', 'very', 'easy', '!!!']
<class 'list'>
```

Note: Sometimes we can take list inside another list, such type of lists are called nested lists.

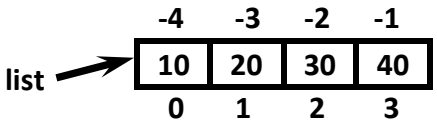
```
[10, 20, [30, 40]]
```

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

```
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])
```

Output

```
D:\Python_classes>py test.py
[3, 5, 7]
[5, 7, 9]
[4, 5, 6, 7]
[9, 7, 5]
[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.

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

```
D:\Python_classes>py test.py
[10, 20, 30, 40]
[10, 777, 30, 40]
```

Traversing the Elements of List:

The sequential access of each element in the list is called traversal.

1) By using while Loop:

```
1) n = [0,1,2,3,4,5,6,7,8,9,10]
2) i = 0
3) while I < len(n):
4)     print(n[i])
5)     i=i+1
```

```
D:\Python_classes>py test.py
0
1
2
3
4
5
6
7
8
9
10
```

2) By using for Loop:

```
1) n=[0,1,2,3,4,5,6,7,8,9,10]
2) for n1 in n:
3)     print(n1)
```

```
D:\Python_classes>py test.py
0
1
2
3
4
5
6
7
8
9
10
```

3) To display only Even Numbers:

```
1)  n=[0,1,2,3,4,5,6,7,8,9,10]
2)  for n1 in n:
3)      if n1%2==0:
4)          print(n1)
```

```
D:\Python_classes>py test.py
0
2
4
6
8
10
```

4) To display Elements by Index wise:

```
1)  l = ["A","B","C"]
2)  x = len(l)
3)  for i in range(x):
4)      print(l[i],"is available at positive index: ",i,"and at negative index: ",i-x)
```

```
D:\Python_classes>py test.py
A is available at positive index:  0 and at negative index:  -3
B is available at positive index:  1 and at negative index:  -2
C is available at positive index:  2 and at negative index:  -1
```

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))
```

```
D:\Python_classes>py test.py
1
4
2
0
```

3) index():

Returns the index of first occurrence of the specified item.

```
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.

```
1) list=[]
2) list.append("A")
3) list.append("B")
4) list.append("C")
5) print(list)
```

```
D:\Python_classes>py test.py
['A', 'B', 'C']
```

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)
```

```
D:\Python_classes>py test.py
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

2) insert() Function:

To insert item at specified index position

```
1) n=[1,2,3,4,5]
2) n.insert(1,888)
3) print(n)
```

```
D:\Python_classes>py test.py
[1, 888, 2, 3, 4, 5]
```

```
1) n=[1,2,3,4,5]
2) n.insert(10,777)
3) n.insert(-10,999)
4) print(n)
```

```
D:\Python_classes>py test.py
[999, 1, 2, 3, 4, 5, 777]
```

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

```
1) order1=["Chicken","Mutton","Fish"]
2) order2=["RC","KF","FO"]
3) order1.extend(order2)
4) print(order1)
```

```
D:\Python_classes>py test.py
['Chicken', 'Mutton', 'Fish', 'RC', 'KF', 'FO']
```

```
1) order = ["Chicken","Mutton","Fish"]
2) order.extend("Mushroom")
3) print(order)
```

```
D:\Python_classes>py test.py
['Chicken', 'Mutton', 'Fish', 'M', 'u', 's', 'h', 'r', 'o', 'o', 'm']
```

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.

```
1) n=[10,20,10,30]
2) n.remove(10)
3) print(n)
```

```
D:\Python_classes>py test.py
[20, 10, 30]
```

If the specified item not present in list then we will get ValueError

```
1) n=[10,20,10,30]
2) n.remove(40)
3) print(n)
```

ValueError: list.remove(x): x not in list

Note: Hence before using remove() method first we have to check specified element present in the list or not by using in operator.

5) pop() Function:

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

```
1) n=[10,20,30,40]
2) print(n.pop())
3) print(n.pop())
4) print(n)
```

```
D:\Python_classes>py test.py
40
30
[10, 20]
```

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

```
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

```
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.

```
1) n=[10,20,30,40]
2) n.reverse()
3) print(n)
```

```
D:\Python_classes>py test.py
[40, 30, 20, 10]
```

2) sort():

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

```
1) n=[20,10,"A","B"]
2) n.sort()
3) print(n)
```

TypeError: '<' not supported between instances of 'str' and 'int'

Note: In Python 2 if List contains both numbers and Strings then sort() function first sort numbers followed by strings

```
1) n=[20,"B",10,"A"]
2) n.sort()
3) print(n)# [10,20,'A','B']
```

But in Python 3 it is invalid.

To Sort in Reverse of Default Natural Sorting Order:

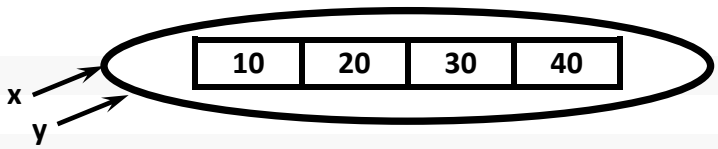
We can sort according to reverse of default natural sorting order by using reverse=True argument.

```
1) n = [40,10,30,20]
2) n.sort()
3) print(n) → [10,20,30,40]
4) n.sort(reverse = True)
5) print(n) → [40,30,20,10]
6) n.sort(reverse = False)
7) print(n) → [10,20,30,40]
```

Aliasing and Cloning of List Objects:

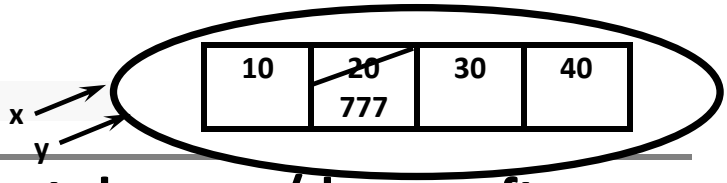
The process of giving another reference variable to the existing list is called aliasing.

```
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
```



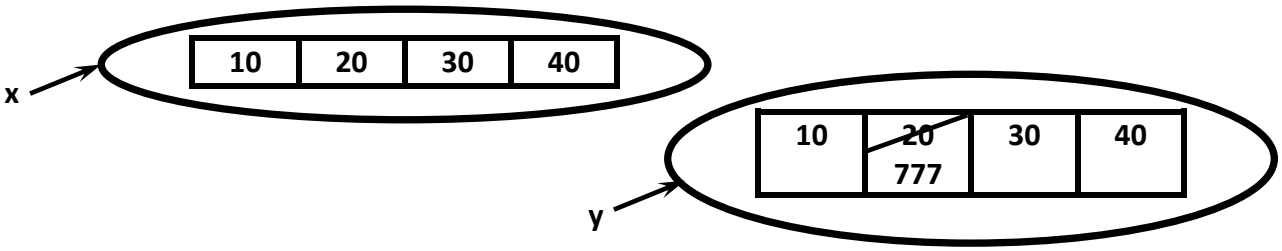
```
3) y[1] = 777
4) print(x) → [10,777,30,40]
```

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.

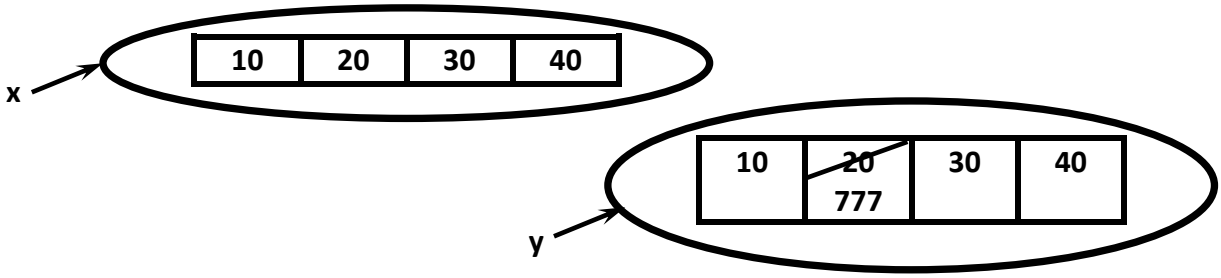
1) By using Slice Operator:

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



2) By using copy() Function:

```
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

Using Mathematical Operators for List Objects:

We can use + and * operators for List objects.

1) Concatenation Operator (+):

We can use + to concatenate 2 lists into a single list

```
1) a = [10, 20, 30]
2) b = [40, 50, 60]
3) c = a+b
4) print(c) → [10, 20, 30, 40, 50, 60]
```

Note: To use + operator compulsory both arguments should be list objects, otherwise we will get TypeError.

Eg:

c = a+40 → TypeError: can only concatenate list (not "int") to list.
c = a+[40] → Valid

2) Repetition Operator (*):

We can use repetition operator * to repeat elements of list specified number of times.

```
1) x = [10, 20, 30]
2) y = x*3
3) print(y) → [10, 20, 30, 10, 20, 30, 10, 20, 30]
```

Comparing List Objects

We can use comparison operators for List objects.

```
1) x = ["Dog", "Cat", "Rat"]
2) y = ["Dog", "Cat", "Rat"]
3) z = ["DOG", "CAT", "RAT"]
4) print(x == y) → True
5) print(x == z) → False
6) print(x !=  z) → True
```

Note: Whenever we are using comparison operators (==, !=) for List objects then the following should be considered

1) The Number of Elements
2) The Order of Elements
3) The Content of Elements (Case Sensitive)

Note: When ever we are using relatational Operators (<, <=, >, >=) between List Objects, only 1ST Element comparison will be performed.

```
1) x = [50, 20, 30]
2) y = [40, 50, 60, 100, 200]
3) print(x>y) → True
4) print(x>=y) → True
5) print(x<y) → False
6) print(x<=y) → False
```

Eg:

```
1) x = ["Dog", "Cat", "Rat"]
2) y=["Rat", "Cat", "Dog"]
3) print(x>y) → False
4) print(x>=y) → False
5) print(x<y) → True
6) print(x<=y) → True
```

Membership Operators:

We can check whether element is a member of the list or not by using memebership operators.

1) in Operator
2) not in Operator

```
1) n=[10,20,30,40]
2) print (10 in n)
3) print (10 not in n)
4) print (50 in n)
5) print (50 not in n)
```

Output

```
True
False
False
True
```

clear() Function:

We can use clear() function to remove all elements of List.

```
1) n=[10,20,30,40]
2) print(n)
3) n.clear()
4) print(n)
```

Output

```
D:\Python_classes>py test.py
[10, 20, 30, 40]
[]
```

Nested Lists:

Sometimes we can take one list inside another list. Such type of lists are called nested lists.

```
1)  n=[10,20,[30,40]]
2)  print(n)
3)  print(n[0])
4)  print(n[2])
5)  print(n[2][0])
6)  print(n[2][1])
```

Output

```
D:\Python_classes>py test.py
[10, 20, [30, 40]]
10
[30, 40]
30
40
```

Note: **We can access nested list elements by using index just like accessing multi dimensional array elements.**

Nested List as Matrix:

In Python we can represent matrix by using nested lists.

```
1)  n=[[10,20,30],[40,50,60],[70,80,90]]
2)  print(n)
3)  print("Elements by Row wise:")
4)  for r in n:
5)      print(r)
6)  print("Elements by Matrix style:")
7)  for i in range(len(n)):
8)      for j in range(len(n[i])):
9)          print(n[i][j],end=' ')
10)     print()
```

Output

```
D:\Python_classes>py test.py
[[10, 20, 30], [40, 50, 60], [70, 80, 90]]

Elements by Row wise:
[10, 20, 30]
[40, 50, 60]
[70, 80, 90]

Elements by Matrix style:
10 20 30
40 50 60
70 80 90
```

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]

```
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)
```

```
D:\Python_classes>py test.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[2, 4, 8, 16, 32]
[4, 16, 36, 64, 100]
```

```
1) words=["Balaiah","Nag","Venkatesh","Chiranjeevi"]
2) l=[w[0] for  w in words]
3) print(l)
```

Output: ['B', 'N', 'V', 'C']

```
1) num1=[10,20,30,40]
2) num2=[30,40,50,60]
3) num3=[ i for i in num1 if i not in num2]
4) print(num3)   [10,20]
5)
6) common elements present in num1 and num2
```

```
7)  num4=[i for i in num1 if i in num2]
8)  print(num4)   [30, 40]
```

Eg:

```
1)  words="the quick brown fox jumps over the lazy dog".split()
2)  print(words)
3)  l=[[w.upper(),len(w)]     for w in words]
4)  print(l)
```

Output

```
['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
[['THE', 3], ['QUICK', 5], ['BROWN', 5], ['FOX', 3], ['JUMPS', 5], ['OVER', 4],
['THE', 3], ['LAZY', 4], ['DOG', 3]]
```

Q) Write a Program to display Unique Vowels present in the given Word?

```
1)  vowels=['a','e','i','o','u']
2)  word=input("Enter the word to search for vowels: ")
3)  found=[]
4)  for letter in word:
5)      if letter in vowels:
6)          if letter not in found:
7)              found.append(letter)
8)  print(found)
9)  print("The number of different vowels present in",word,"is",len(found))
```

```
D:\Python_classes>py test.py
Enter the word to search for vowels: durgasoftwaresolutions
['u', 'a', 'o', 'e', 'i']
The number of different vowels present in durgasoftwaresolutions is 5
```

List out all Functions of List and write a Program to use these Functions

Learn Complete Python In Simple Way

TUPLE

DATA STRUCTURE STUDY MATERIAL

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.
2) Hence Tuple is Read only version of List.
3) If our data is fixed and never changes then we should go for Tuple.
4) Insertion Order is preserved
5) Duplicates are allowed
6) Heterogeneous objects are allowed.
7) We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.
8) 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)
9) We can represent Tuple elements within Parenthesis and with comma seperator.
10) Parenethesis are optional but recommended to use.

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

```
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:

```
1) list=[10,20,30]
2)       t=tuple(list)
3)       print(t)
4)
5)       t=tuple(range(10,20,2))
6)       print(t)
```

Accessing Elements of Tuple:

We can access either by index or by slice operator

1) By using Index:

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

2) By using Slice Operator:

```
1) t=(10,20,30,40,50,60)
2) print(t[2:5])
3) print(t[2:100])
4) 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 (+):

```
1) t1=(10,20,30)
2) t2=(40,50,60)
3) t3=t1+t2
4) print(t3) → (10,20,30,40,50,60)
```

2) Multiplication Operator OR Repetition Operator (*)

```
1) t1=(10,20,30)
2) t2=t1*3
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

```
1) t=(40,10,30,20)
2) t1=sorted(t)
3) print(t1)
4) print(t)
```

Output

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

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

```
1) t = (40,10,30,20)
2) print(min(t)) → 10
3) 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

```
1) t1=(10,20,30)
2) t2=(40,50,60)
3) t3=(10,20,30)
4) print(cmp(t1,t2)) → -1
5) print(cmp(t1,t3)) → 0
6) print(cmp(t2,t3)) → +1
```

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

Tuple Packing and Unpacking:

We can create a tuple by packing a group of variables.

Eg:

```
a = 10
b = 20
c = 30
d = 40
t = a, b, c, d
print(t) → (10, 20, 30, 40)
```

- Here a, b, c, d are packed into a Tuple t. This is nothing but Tuple packing.
- Tuple unpacking is the reverse process of Tuple packing.
- We can unpack a Tuple and assign its values to different variables.

```
1) t=(10,20,30,40)
2) a,b,c,d=t
3) print("a=",a,"b=",b,"c=",c,"d=",d)
```

Output: a= 10 b= 20 c= 30 d= 40

Note: At the time of tuple unpacking the number of variables and number of values should be same, otherwise we will get ValueError.

Eg:

t = (10,20,30,40)
a, b, c = t → ValueError: too many values to unpack (expected 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.

```
1)  t= ( x**2  for x in range(1,6))
2)  print(type(t))
3)  for x in t:
4)      print(x)
```

```
D:\Python_classes>py test.py
<class 'generator'>
1
4
9
16
25
```

Q) Write a Program to take a Tuple of Numbers from the Keyboard and Print its Sum and Average?

```
1)  t=eval(input("Enter Tuple of Numbers:"))
2)  l=len(t)
3)  sum=0
4)  for x in t:
5)      sum=sum+x
6)  print("The Sum=",sum)
7)  print("The Average=",sum/l)
```

```
D:\Python_classes>py test.py
Enter Tuple of Numbers:(10,20,30,40)
The Sum= 100
The Average= 25.0

D:\Python_classes>py test.py
Enter Tuple of Numbers: (100,200,300)
```

The Sum= 600
The Average= 200.0

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.

Learn Complete Python In Simple Way

SET

DATA STRUCTURE STUDY MATERIAL

- 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:

```
1) s={10,20,30,40}
2) print(s)
3) print(type(s))
```

Output

```
{40, 10, 20, 30}
<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.

```
1) s={}
2) print(s)
3) print(type(s))
```

Output

```
{}
<class 'dict'>
```

Eg:

```
1) s=set()
2) print(s)
3) print(type(s))
```

Output

```
set()
<class 'set'>
```

Important Functions of Set:

1) add(x):

Adds item x to the set.

```
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.

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

Output: {0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}

Q) What is the difference between add() and update() Functions in Set?

- We can use add() to add individual item to the Set,where as we can use update() function to add multiple items to Set.
- add() function can take only one argument where as update() function can take any number of arguments but all arguments should be iterable objects.

Q) Which of the following are valid for set s?

1) s.add(10)
2) s.add(10,20,30) → TypeError: add() takes exactly one argument (3 given)
3) s.update(10) → TypeError: 'int' object is not iterable
4) s.update(range(1,10,2),range(0,10,2))

3) copy():

- Returns copy of the set.
- It is cloned object.

```
1) s = {10,20,30}
2) s1 = s.copy()
3) print(s1)
```

4) pop():

It removes and returns some random element from the set.

```
1) s={40,10,30,20}
2) print(s)
3) print(s.pop())
4) print(s)
```

Output

```
{40, 10, 20, 30}
40
{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.

```
1) s = {40, 10, 30, 20}
2) s.remove(30)
3) print {◊(s) 40, 10, 20}
```

```
4) s.remove(50 KeyError: ◊) 50
```

6) discard(x):

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

```
1) s = {10, 20, 30}
2) s.discard(10)
3) print {◊(s)  20, 30}
4) s.discard(50)
5) print {◊(s) 20, 30}
```

Q) What is the difference between remove() and discard() functions in Set?
Q) Explain differences between pop(),remove() and discard() functionsin Set?

7) clear():

To remove all elements from the Set.

```
1) s={10,20,30}
2) print(s)
3) s.clear()
4) print(s)
```

Output

```
{10, 20, 30}
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.

```
1) x = {10, 20, 30, 40}
2) y = {30, 40, 50, 60}
3) print (x.union(y))  → {10, 20, 30, 40, 50, 60}
4) 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.

```
1) x = {10, 20, 30, 40}
2) y = {30, 40, 50, 60}
3) print (x.intersection(y)) → {40, 30}
4) print(x&y) → {40, 30}
```

3) difference():

- x.difference(y) OR x-y.
- Returns the elements present in x but not in y.

```
1) x = {10, 20, 30, 40}
2) y = {30, 40, 50, 60}
3) print (x.difference(y)) → 10, 20
4) print (x-y) → {10, 20}
5) 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.

```
1) x = {10, 20, 30, 40}
2) y = {30, 40, 50, 60}
3) print (x.symmetric_difference(y)) → {10, 50, 20, 60}
4) print(x^y) → {10, 50, 20, 60}
```

Membership Operators: (in, not in)

```
1) s=set("durga")
2) print(s)
3) print('d' in s)
4) print('z' in s)
```

Output

```
{'u', 'g', 'r', 'd', 'a'}
True
False
```

Set Comprehension:

Set comprehension is possible.

```
1) s = {x*x  for x in range(5)}
2) print (s)  → {0, 1, 4, 9, 16}
3)
```

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

Set Objects won't support indexing and slicing:

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

Q) Write a Program to eliminate Duplicates Present in the List?

Approach - 1

```
1) l=eval(input("Enter List of values: "))
2) s=set(l)
3) print(s)
```

```
D:\Python_classes>py test.py
Enter List of values: [10,20,30,10,20,40]
{40, 10, 20, 30}
```

Approach - 2

```
1) l=eval(input("Enter List of values: "))
2) l1=[]
3) for x in l:
4)     if x not in l1:
5)         l1.append(x)
6) print(l1)
```

```
D:\Python_classes>py test.py
Enter List of values: [10,20,30,10,20,40]
[10, 20, 30, 40]
```

Q) Write a Program to Print different Vowels Present in the given Word?

```
1) w=input("Enter word to search for vowels: ")
2) s=set(w)
3) v={'a','e','i','o','u'}
4) d=s.intersection(v)
5) print("The different vowel present in",w,"are",d)
```

```
D:\Python_classes>py test.py
Enter word to search for vowels: durga
The different vowel present in durga are {'u', 'a'}
```

Learn Complete Python In Simple Way

DICTIONARY

DATA STRUCTURE STUDY MATERIAL

- 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

```
1) d[100]="durga"
2) d[200]="ravi"
3) d[300]="shiva"
4) print(d) → {100: 'durga', 200: 'ravi', 300: 'shiva'}
```

- If we know data in advance then we can create dictionary as follows
- d = {100:'durga' ,200:'ravi', 300:'shiva'}
- d = {key:value, key:value}

How to Access Data from the Dictionary?

We can access data by using keys.

```
1) d = {100:'durga',200:'ravi', 300:'shiva'}
2) print(d[100])   #durga
3) 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

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

```
if 400 in d:
        print(d[400])
```

Q) Write a Program to Enter Name and Percentage Marks in a Dictionary and Display Information on the Screen

```
1)  rec={}
2)  n=int(input("Enter number of students: "))
3)  i=1
4)  while i <=n:
5)      name=input("Enter Student Name: ")
6)      marks=input("Enter % of Marks of Student: ")
7)      rec[name]=marks
8)      i=i+1
9)  print("Name of Student","\t","% of marks")
10) for x in rec:
11)     print("\t",x,"\t\t",rec[x])
```

```
D:\Python_classes>py test.py
Enter number of students: 3
Enter Student Name: durga
Enter % of Marks of Student: 60%
Enter Student Name: ravi
Enter % of Marks of Student: 70%
Enter Student Name: shiva
Enter % of Marks of Student: 80%

Name of Student          % of marks
---------------         ------------
        durga            60%
        ravi             70 %
        shiva            80%
```

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.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) d[400]="pavan"
4) print(d)
5) d[100]="sunny"
6) print(d)
```

Output

```
{100: 'durga', 200: 'ravi', 300: 'shiva'}
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
{100: 'sunny', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

How to Delete Elements from Dictionary?

1) del d[key]

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

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) del d[100]
4) print(d)
5) del d[400]
```

Output

```
{100: 'durga', 200: 'ravi', 300: 'shiva'}
{200: 'ravi', 300: 'shiva'}
KeyError: 400
```

2) d.clear()

To remove all entries from the dictionary.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) d.clear()
4) print(d)
```

Output

```
{100: 'durga', 200: 'ravi', 300: 'shiva'}
{}
```

3) del d

To delete total dictionary.Now we cannot access d.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) del d
4) print(d)
```

Output

```
{100: 'durga', 200: 'ravi', 300: 'shiva'}
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:"durga",200:"ravi"}) → It creates dictionary with specified elements
- d = dict([(100,"durga"),(200,"shiva"),(300,"ravi")])
  → 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.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d[100]) → durga
3) print(d[400]) → KeyError:400
4) print(d.get(100)) → durga
5) print(d.get(400)) → None
6) print(d.get(100,"Guest"))  → durga
7) print(d.get(400,"Guest"))  → Guest
```

5) 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.**

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.pop(100))
3) print(d)
4) print(d.pop(400))
```

Output

```
durga
{200: 'ravi', 300: 'shiva'}
KeyError: 400
```

6) popitem():

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

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) print(d.popitem())
4) print(d)
```

Output

```
{100: 'durga', 200: 'ravi', 300: 'shiva'}
(300, 'shiva')
{100: 'durga', 200: 'ravi'}
If the dictionary is empty then we will get KeyError
d={}
print(d.popitem()) ==>KeyError: 'popitem(): dictionary is empty'
```

7) keys():

It returns all keys associated eith dictionary.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.keys())
3) for k in d.keys():
4)     print(k)
```

Output

```
dict_keys([100, 200, 300])
100
200
300
```

8) values():

It returns all values associated with the dictionary.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.values())
3) for v in d.values():
4)     print(v)
```

Output

```
dict_values(['durga', 'ravi', 'shiva'])
durga
ravi
shiva
```

9) items():

It returns list of tuples representing key-value pairs.

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

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) for k,v in d.items():
3)     print(k,"--",v)
```

Output

```
100 -- durga
200 -- ravi
300 -- shiva
```

10) copy():

To create exactly duplicate dictionary (cloned copy)

```
d1 = d.copy();
```

11) 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.

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.setdefault(400,"pavan"))
3) print(d)
4) print(d.setdefault(100,"sachin"))
5) print(d)
```

Output

```
pavan
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
durga
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

12) update():

```
d.update(x)
```

All items present in the dictionary x will be added to dictionary d

Q) Write a Program to take Dictionary from the Keyboard and print the Sum of Values?

```
1) d=eval(input("Enter dictionary:"))
2) s=sum(d.values())
3) print("Sum= ",s)
```

Output

```
D:\Python_classes>py test.py
Enter dictionary:{'A':100,'B':200,'C':300}
Sum=  600
```

Q) Write a Program to find Number of Occurrences of each Letter present in the given String?

```
1) word=input("Enter any word: ")
2) d={}
3) for x in word:
4)     d[x]=d.get(x,0)+1
5) for k,v in d.items():
6)     print(k,"occurred ",v," times")
```

Output

```
D:\Python_classes>py test.py
Enter any word: mississippi
m occurred  1  times
i occurred  4  times
s occurred  4  times
p occurred  2  times
```

Q) Write a Program to find Number of Occurrences of each Vowel present in the given String?

```
1) word=input("Enter any word: ")
2) vowels={'a','e','i','o','u'}
3) d={}
4) for x in word:
5)     if x in vowels:
6)         d[x]=d.get(x,0)+1
7) for k,v in sorted(d.items()):
8)     print(k,"occurred ",v," times")
```

Output

```
D:\Python_classes>py test.py
Enter any word: doganimaldoganimal
a occurred  4  times
i occurred  2  times
o occurred  2  times
```

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?

```
1)  n=int(input("Enter the number of students: "))
2)  d={}
3)  for i in range(n):
4)      name=input("Enter Student Name: ")
5)      marks=input("Enter Student Marks: ")
6)      d[name]=marks
7)  while True:
8)      name=input("Enter Student Name to get Marks: ")
9)      marks=d.get(name,-1)
10)     if marks== -1:
11)         print("Student Not Found")
12)     else:
13)         print("The Marks of",name,"are",marks)
14)     option=input("Do you want to find another student marks[Yes|No]")
15)     if option=="No":
16)         break
17) print("Thanks for using our application")
```

Output

```
D:\Python_classes>py test.py
Enter the number of students: 5

Enter Student Name: sunny
Enter Student Marks: 90

Enter Student Name: banny
Enter Student Marks: 80

Enter Student Name: chinny
Enter Student Marks: 70

Enter Student Name: pinny
Enter Student Marks: 60

Enter Student Name: vinny
Enter Student Marks: 50

Enter Student Name to get Marks: sunny
The Marks of sunny are 90
```

```
Do you want to find another student marks[Yes|No]Yes

Enter Student Name to get Marks: durga
Student Not Found

Do you want to find another student marks[Yes|No]No
Thanks for using our application
```

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)
```

Output

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Learn Complete Python In Simple Way

FUNCTIONS

STUDY MATERIAL

- 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()
```

Parameters

Parameters are inputs to the function. If a function contains parameters, then at the time of calling,compulsory we should provide values otherwise,otherwise we will get error.

Eg: Write a function to take name of the student as input and print wish message by name.

```
1) def wish(name):
2)     print("Hello",name," Good Morning")
3) wish("Durga")
4) wish("Ravi")
```

```
D:\Python_classes>py test.py
Hello Durga  Good Morning
Hello Ravi  Good Morning
```

Eg: Write a function to take number as input and print its square value

```
1) def squareIt(number):
2)     print("The Square of",number,"is", number*number)
3) squareIt(4)
4) squareIt(5)
```

```
D:\Python_classes>py test.py
The Square of 4 is 16
The Square of 5 is 25
```

Return Statement:

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

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))
```

```
D:\Python_classes>py test.py
The sum is 30
The sum is 300
```

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

```
1) def  f1():
2)     print("Hello")
3) f1()
4) print(f1())
```

Output

```
Hello
Hello
None
```

Q) Write a Function to check whether the given Number is Even OR Odd?

```
1) def even_odd(num):
2)     if num%2==0:
3)         print(num,"is Even Number")
4)     else:
5)         print(num,"is Odd Number")
6) even_odd(10)
7) even_odd(15)
```

Output

```
D:\Python_classes>py test.py
10 is Even Number
15 is Odd Number
```

Q) Write a Function to find Factorial of given Number?

```
1) def fact(num):
2)     result=1
3)     while num>=1:
4)         result=result*num
5)         num=num-1
6)     return result
7) for i in range(1,5):
```

```
8)     print("The Factorial of",i,"is :",fact(i))
```

Output

```
D:\Python_classes>py test.py
The Factorial of 1 is : 1
The Factorial of 2 is : 2
The Factorial of 3 is : 6
The Factorial of 4 is : 24
```

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.

Eg 1:

```
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)
```

Output

```
The Sum is : 150
The Subtraction is : 50
```

Eg 2:

```
1)   def calc(a,b):
2)      sum=a+b
3)      sub=a-b
4)      mul=a*b
5)      div=a/b
6)      return sum,sub,mul,div
7)  t=calc(100,50)
8)  print("The Results are")
9)  for i in t:
10)     print(i)
```

Output

```
The Results are
150
```

50
5000
2.0

Types of Arguments

```
def  f1(a,b):
   ------
   ------
   ------
 f1(10,20)
```

a, b are formal arguments where as 10,20 are actual arguments.

There are 4 types are actual arguments are allowed in Python.

1) Positional Arguments
2) Keyword Arguments
3) Default Arguments
4) Variable Length Arguments

1) Positional Arguments:

- These are the arguments passed to function in correct positional order.

```
def sub(a, b):
    print(a-b)

sub(100, 200)
sub(200, 100)
```

- The number of arguments and position of arguments must be matched. If we change the order then result may be changed.
- If we change the number of arguments then we will get error.

2) Keyword Arguments:

We can pass argument values by keyword i.e by parameter name.

```
1) def wish(name,msg):
2)     print("Hello",name,msg)
3) wish(name="Durga",msg="Good Morning")
4) wish(msg="Good Morning",name="Durga")
```

Output

```
Hello Durga Good Morning
Hello Durga Good Morning
```

Here the order of arguments is not important but number of arguments must be matched.

Note: We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments,otherwise we will get syntaxerror.

```
1) def wish(name,msg):
2)     print("Hello",name,msg)
3) wish("Durga","GoodMorning") → Valid
4) wish("Durga",msg="GoodMorning") → Valid
5) wish(name="Durga","GoodMorning") → Invalid
6) SyntaxError: positional argument follows keyword argument
```

3) Default Arguments:

Sometimes we can provide default values for our positional arguments.

```
1) def wish(name="Guest"):
2)     print("Hello",name,"Good Morning")
3) wish("Durga")
4) wish()
```

Output

Hello Durga Good Morning
Hello Guest Good Morning
If we are not passing any name then only default value will be considered.

***Note:**

After default arguments we should not take non default arguments.

```
1) def wish(name="Guest",msg="Good Morning"): → Valid
2) def wish(name,msg="Good Morning"): → Valid
3) def wish(name="Guest",msg): → Invalid
```

SyntaxError: non-default argument follows default argument

4) Variable Length Arguments:

- Sometimes we can pass variable number of arguments to our function, such type of arguments are called variable length arguments.
- We can declare a variable length argument with * symbol as follows
- def f1(*n):
- We can call this function by passing any number of arguments including zero number.
- Internally all these values represented in the form of tuple.

```
1)  def sum(*n):
2)      total=0
3)      for n1 in n:
4)          total=total+n1
5)      print("The Sum=",total)
6)
7)  sum()
8)  sum(10)
9)  sum(10,20)
10) sum(10,20,30,40)
```

Output

```
The Sum= 0
The Sum= 10
The Sum= 30
The Sum= 100
```

Note: We can mix variable length arguments with positional arguments.

```
1)  def  f1(n1,*s):
2)      print(n1)
3)      for s1 in s:
4)          print(s1)
5)
6)  f1(10)
7)  f1(10,20,30,40)
8)  f1(10,"A",30,"B")
```

Output

```
10
10
20
30
40
10
A
30
B
```

Note: After variable length argument,if we are taking any other arguments then we should provide values as keyword arguments.

```
1)  def  f1(*s,n1):
2)      for s1 in s:
3)          print(s1)
4)      print(n1)
5)
6)  f1("A","B",n1=10)
```

Output

A
B
10
f1("A","B",10) → Invalid
TypeError: f1() missing 1 required keyword-only argument: 'n1'

Note: We can declare key word variable length arguments also.

- **For this we have to use **.**
- **def f1(**n):**
- **We can call this function by passing any number of keyword arguments. Internally these keyword arguments will be stored inside a dictionary.**

```
1)  def  display(**kwargs):
2)      for k,v in kwargs.items():
3)          print(k,"=",v)
4)  display(n1=10,n2=20,n3=30)
5)  display(rno=100,name="Durga",marks=70,subject="Java")
```

Output

n1 = 10
n2 = 20
n3 = 30
rno = 100
name = Durga
marks = 70
subject = Java

Case Study:

```
def f(arg1,arg2,arg3=4,arg4=8):
        print(arg1,arg2,arg3,arg4)
```

1) f(3,2) → 3 2 4 8

2) f(10,20,30,40) → 10 20 30 40

3) f(25,50,arg4=100) → 25 50 4 100

4) f(arg4=2,arg1=3,arg2=4) → 3 4 4 2

5) f() → Invalid
   TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'

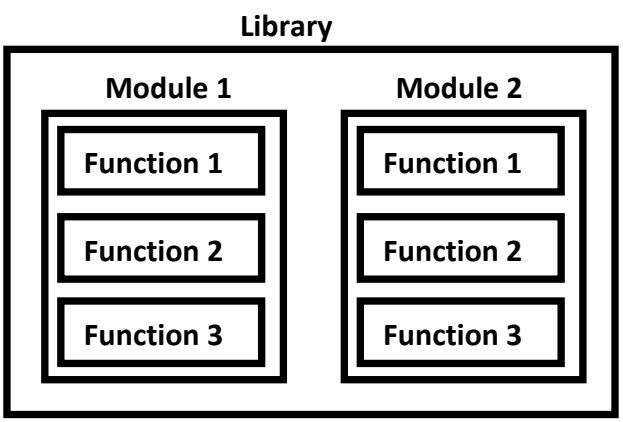
6) f(arg3=10, arg4=20, 30, 40) → Invalid
   SyntaxError: positional argument follows keyword argument
   [After keyword arguments we should not take positional arguments]

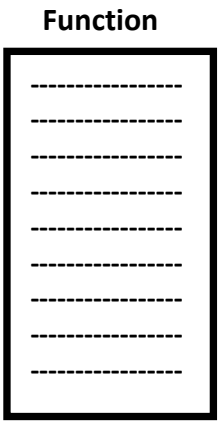
7) f(4, 5, arg2 = 6) → Invalid
   TypeError: f() got multiple values for argument 'arg2'

8) f(4, 5, arg3 = 5, arg5 = 6) → Invalid
   TypeError: f() got an unexpected keyword argument 'arg5'

Note: Function vs Module vs Library

1) A group of lines with some name is called a function
2) A group of functions saved to a file, is called Module
3) A group of Modules is nothing but Library





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.**

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

Output

```
10
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.**

```
1) def f1():
2)     a=10
3)     print(a)    # valid
4)
5) def f2():
6)     print(a)     #invalid
7)
8) f1()
9) f2()
```

```
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**

```
1)  a=10
2)  def f1():
3)      a=777
4)      print(a)
5)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
11)
```

Output

```
777
10
```

```
1)  a=10
2)  def f1():
3)      global a
4)      a=777
5)      print(a)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
```

Output

```
777
777
```

```
1)  def f1():
2)      a=10
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
```

```
8) f1()
9) f2()
```

Output: NameError: name 'a' is not defined

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

Output

10
10

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

```
1) a = 10 → Global Variable
2) def f1():
3)     a=777 → Local Variable
4)     print(a)
5)     print(globals()['a'])
6) f1()
```

Output

777
10

Recursive Functions

A function that calls itself is known as Recursive Function.

Eg:

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1 * factorial(0)
= 3 * 2 * 1 * 1
= 6

factorial(n) = n * factorial(n-1)

The main advantages of recursive functions are:
1) We can reduce length of the code and improves readability.
2) We can solve complex problems very easily.

Q) Write a Python Function to find Factorial of given Number with Recursion

```
1)  def factorial(n):
2)      if n==0:
3)          result=1
4)      else:
5)          result=n*factorial(n-1)
6)      return result
7)  print("Factorial of 4 is :",factorial(4))
8)  print("Factorial of 5 is :",factorial(5))
```

Output
Factorial of 4 is : 24
Factorial of 5 is : 120

Anonymous Functions:

- Sometimes we can declare a function without any name,such type of nameless functions are called anonymous functions or lambda functions.
- The main purpose of anonymous function is just for instant use(i.e for one time usage)

Normal Function:

We can define by using def keyword.

```
def squareIt(n):
    return n*n
```

Lambda Function:

We can define by using lambda keyword `lambda n:n*n`

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.

Q) Write a Program to create a Lambda Function to find Square of given Number?

```
1) s=lambda n:n*n
2) print("The Square of 4 is :",s(4))
3) print("The Square of 5 is :",s(5))
```

Output

```
The Square of 4 is : 16
The Square of 5 is : 25
```

Q) Lambda Function to find Sum of 2 given Numbers

```
1) s=lambda a,b:a+b
2) print("The Sum of 10,20 is:",s(10,20))
3) print("The Sum of 100,200 is:",s(100,200))
```

Output

```
The Sum of 10,20 is: 30
The Sum of 100,200 is: 300
```

Q) Lambda Function to find biggest of given Values

```
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))
```

Output

```
The Biggest of 10,20 is: 20
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, because 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.

Q) Program to filter only Even Numbers from the List by using filter() Function?

Without Lambda Function:

```
1) def isEven(x):
2)     if x%2==0:
3)         return True
4)     else:
5)         return False
6) l=[0,5,10,15,20,25,30]
7) l1=list(filter(isEven,l))
8) print(l1)  #[0,10,20,30]
```

With Lambda 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.**
- **Eg: For every element present in the list perform double and generate new list of doubles.**
- **Syntax: map(function, sequence)**
- **The function can be applied on each element of sequence and generates new sequence.**

Without Lambda

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

With Lambda

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

Eg 2: To find square of given numbers

```
1)  l=[1,2,3,4,5]
2)  l1=list(map(lambda x:x*x,l))
3)  print(l1)      #[1, 4, 9, 16, 25]
```

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

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

```
1)  from functools import *
2)  l=[10,20,30,40,50]
3)  result=reduce(lambda x,y:x+y,l)
4)  print(result) # 150
```

Eg:

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

Eg:

```
1) from functools import *
2) result=reduce(lambda x,y:x+y,range(1,101))
3) print(result)   #5050
```

Everything is an Object:

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

```
1) def f1():
2)     print("Hello")
3) print(f1)
4) print(id(f1))
```

Output:

```
<function f1 at 0x00419618>
4298264
```

Function Aliasing:

For the existing function we can give another name, which is nothing but function aliasing.

```
1) def wish(name):
2)     print("Good Morning:",name)
3)
4) greeting=wish
5) print(id(wish))
6) print(id(greeting))
7)
8) greeting('Durga')
9) wish('Durga')
```

Output:

```
4429336
4429336
Good Morning: Durga
Good Morning: Durga
```

Note:

- **In the above example only one function is available but we can call that function by using either wish name or greeting name.**
- **If we delete one name still we can access that function by using alias name.**

```
1)  def wish(name):
2)      print("Good Morning:",name)
3)
4)  greeting=wish
5)
6)  greeting('Durga')
7)  wish('Durga')
8)
9)  del wish
10) #wish('Durga') → NameError: name 'wish' is not defined
11) greeting('Pavan')
```

Output:

```
Good Morning: Durga
Good Morning: Durga
Good Morning: Pavan
```

Nested Functions:

We can declare a function inside another function,such type of functions are called Nested functions.

```
1)  def outer():
2)      print("outer function started")
3)      def inner():
4)          print("inner function execution")
5)      print("outer function calling inner function")
6)      inner()
7)  outer()
8)  #inner() → NameError: name 'inner' is not defined
```

Output:

```
outer function started
outer function calling inner function
inner function execution
```

In the above example inner() function is local to outer() function and hence it is not possible to call directly from outside of outer() function.

Note: A function can return another function.

```
1)  def outer():
2)      print("outer function started")
3)      def inner():
4)          print("inner function execution")
5)      print("outer function returning inner function")
6)      return inner
7)  f1=outer()
8)  f1()
9)  f1()
10) f1()
```

Output:

```
outer function started
outer function returning inner function
inner function execution
inner function execution
inner function execution
```

Q) What is the differenece between the following lines?

```
f1 = outer
f1 = outer()
```

- In the first case for the outer() function we are providing another name f1 (function aliasing).
- But in the second case we calling outer() function,which returns inner function.For that inner function() we are providing another name f1

Note: We can pass function as argument to another function

Eg: filter(function,sequence)
map(function,sequence)
reduce(function,sequence)

Learn Complete Python In Simple Way

MODULES

STUDY MATERIAL

- A group of functions, variables and classes saved to a file, which is nothing but module.
- Every Python file (.py) acts as a module.

durgamath.py

```
1)  x = 888
2)
3)  def add(a,b):
4)    print("The Sum:",a+b)
5)
6)  def product(a,b):
7)    print("The Product:",a*b)
```

- durgamath module contains one variable and 2 functions.
- If we want to use members of module in our program then we should import that module.
  import modulename

- We can access members by using module name.
  modulename.variable
  modulename.function()

test.py:

```
1)  import durgamath
2)  print(durgamath.x)
3)  durgamath.add(10,20)
4)  durgamath.product(10,20)
```

Output

888
The Sum: 30
The Product: 200

Note: Whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently.

Renaming a Module at the time of import (Module Aliasing):

- Eg: import durgamath as m
- Here durgamath is original module name and m is alias name.
- We can access members by using alias name m

test.py:

```
1) import durgamath as m
2) print(m.x)
3) m.add(10,20)
4) m.product(10,20)
```

from ... import:

We can import particular members of module by using from ... import .
The main advantage of this is we can access members directly without using module name.

```
1) from durgamath import x,add
2) print(x)
3) add(10,20)
4) product(10,20) → NameError: name 'product' is not defined
```

We can import all members of a module as follows `from durgamath import *`

test.py:

```
1) from durgamath import *
2) print(x)
3) add(10,20)
4) product(10,20)
```

Various Possibilties of import:

1) import modulename
2) import module1,module2,module3
3) import module1 as m
4) import module1 as m1,module2 as m2,module3
5) from module import member
6) from module import member1,member2,memebr3
7) from module import memeber1 as x
8) from module import *

Member Aliasing:

```
1) from durgamath import x as y,add as sum
2) print(y)
3) sum(10,20)
```

Once we defined as alias name,we should use alias name only and we should not use original name

```
1) from durgamath import x as y
2) print(x) → NameError: name 'x' is not defined
```

Reloading a Module:

By default module will be loaded only once eventhough we are importing multiple multiple times.

module1.py:

```
print("This is  from module1")
```

test.py

```
1) import module1
2) import module1
3) import module1
4) import module1
5) print("This is test module")
```

Output

```
This is  from module1
This is test module
```

- **In the above program test module will be loaded only once eventhough we are importing multiple times.**

- **The problem in this approach is after loading a module if it is updated outside then updated version of module1 is not available to our program.**

- **We can solve this problem by reloading module explicitly based on our requirement.**
- **We can reload by using reload() function of imp module.**

```
1) import imp
2) imp.reload(module1)
```

test.py:

```
1) import module1
2) import module1
3) from imp import reload
4) reload(module1)
5) reload(module1)
6) reload(module1)
7) print("This is test module")
```

In the above program module1 will be loaded 4 times in that 1 time by default and 3 times explicitly. In this case output is

```
1) This is  from module1
2) This is  from module1
3) This is  from module1
4) This is  from module1
5) This is test module
```

The main advantage of explicit module reloading is we can ensure that updated version is always available to our program.

Finding Members of Module by using dir() Function:

Python provides inbuilt function dir() to list out all members of current module or a specified module.

dir() → To list out all members of current module
dir(moduleName) → To list out all members of specified module

Eg 1: test.py

```
1) x=10
2) y=20
3) def f1():
4)     print("Hello")
5) print(dir())    # To print all members of current module
```

Output

['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'f1', 'x', 'y']

Eg 2: To display members of particular module

durgamath.py:

```
1)  x=888
2)
3)  def add(a,b):
4)    print("The Sum:",a+b)
5)
6)  def product(a,b):
7)    print("The Product:",a*b)
```

test.py:

```
1)  import durgamath
2)  print(dir(durgamath))
```

Output

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'add', 'product', 'x']
```

Note: For every module at the time of execution Python interpreter will add some special properties automatically for internal use.

Eg: __builtins__,__cached__,'__doc__,__file__, __loader__, __name__,__package__, __spec__

Based on our requirement we can access these properties also in our program.

Eg: test.py

```
1)  print(__builtins__  )
2)  print(__cached__   )
3)  print(__doc__)
4)  print(__file__)
5)  print(__loader__)
6)  print(__name__)
7)  print(__package__)
8)  print(__spec__)
```

Output

```
<module 'builtins' (built-in)>
None
```

None

test.py

```
1) <_frozen_importlib_external.SourceFileLoader object at 0x00572170>
2) __main__
3) None
4) None
```

The Special Variable __name__:

- For every Python program, a special variable __name__ will be added internally.
- This variable stores information regarding whether the program is executed as an individual program or as a module.
- If the program executed as an individual program then the value of this variable is __main__
- If the program executed as a module from some other program then the value of this variable is the name of module where it is defined.
- Hence by using this __name__ variable we can identify whether the program executed directly or as a module.

Demo program:

module1.py:

```
1) def f1():
2)     if __name__=='__main__':
3)         print("The code executed as a program")
4)     else:
5)         print("The code executed as a module from some other program")
6) f1()
```

test.py:

```
1) import module1
2) module1.f1()
```

```
D:\Python_classes>py module1.py
The code executed as a program
```


```
D:\Python_classes>py test.py
The code executed as a module from some other program
The code executed as a module from some other program
```

Working with math Module:

- Python provides inbuilt module math.
- This module defines several functions which can be used for mathematical operations.
- The main important functions are
    1) sqrt(x)
    2) ceil(x)
    3) floor(x)
    4) fabs(x)
    5) log(x)
    6) sin(x)
    7) tan(x)
    8) ....

```
1) from math import *
2) print(sqrt(4))
3) print(ceil(10.1))
4) print(floor(10.1))
5) print(fabs(-10.6))
6) print(fabs(10.6))
```

Output

```
2.0
11
10
10.6
10.6
```

Note: We can find help for any module by using help() function

Eg:

```
import math
help(math)
```

Working with random Module:

- This module defines several functions to generate random numbers.
- We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication.

1) random() Function:

This function always generate some float value between 0 and 1 (not inclusive)

0<x<1

```
1) from random import *
2) for i in range(10):
3)     print(random())
```

Output

```
0.4572685609302056
0.6584325233197768
0.15444034016553587
0.18351427005232201
0.1330257265904884
0.9291139798071045
0.6586741197891783
0.8901649834019002
0.25540891083913053
0.7290504335962871
```

2) randint() Function:

To generate random integer beween two given numbers(inclusive)

```
1) from random import *
2) for i in range(10):
3)     print(randint(1,100)) # generate random int value between 1 and 100(inclusive)
```

Output

```
51
44
39
70
49
74
52
10
40
8
```

3) uniform() Function:

It returns random float values between 2 given numbers (not inclusive)

```
1) from random import *
2) for i in range(10):
3)     print(uniform(1,10))
```

Output

```
9.787695398230332
6.81102218793548
8.068672144377329
8.567976357239834
6.363511674803802
2.176137584071641
4.822867939432386
6.0801725149678445
7.508457735544763
1.9982221862917555
```

random() → in between 0 and 1 (not inclusive)
randint(x,y) → in between x and y (inclusive)
uniform(x,y) → in between x and y (not inclusive)

4) randrange ([start], stop, [step])

- Returns a random number from range
- start <= x < stop
- start argument is optional and default value is 0
- step argument is optional and default value is 1

- randrange(10) → generates a number from 0 to 9
- randrange(1,11) → generates a number from 1 to 10
- randrange(1,11,2) → generates a number from 1,3,5,7,9

```
1) from random import *
2) for i in range(10):
3)     print(randrange(10))
```

Output:

```
9
4
0
2
9
4
```

8
9
5
9

```
1)  from random import *
2)  for i in range(10):
3)      print(randrange(1,11))
```

Output: 2
2
8
10
3
5
9
1
6
3

```
1)  from random import *
2)  for i in range(10):
3)      print(randrange(1,11,2))
```

Output: 1
3
9
5
7
1
1
1
7
3

5) choice() Function:

- It won't return random number.
- It will return a random object from the given list or tuple.

```
1)  from random import *
2)  list=["Sunny","Bunny","Chinny","Vinny","pinny"]
3)  for i in range(10):
4)      print(choice(list))
```

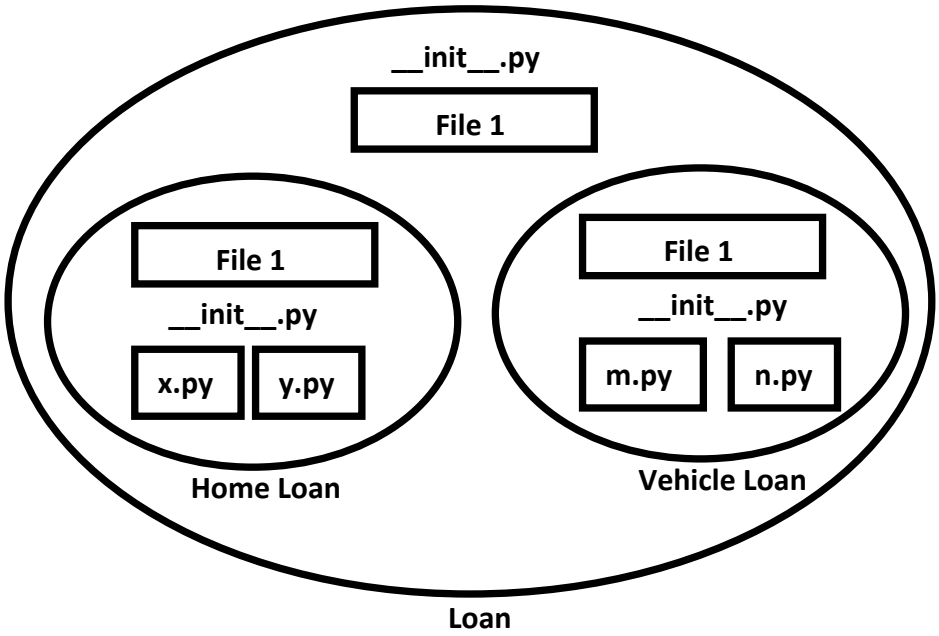
Output
Bunny
pinny
Bunny
Sunny
Bunny
pinny
pinny
Vinny
Bunny
Sunny

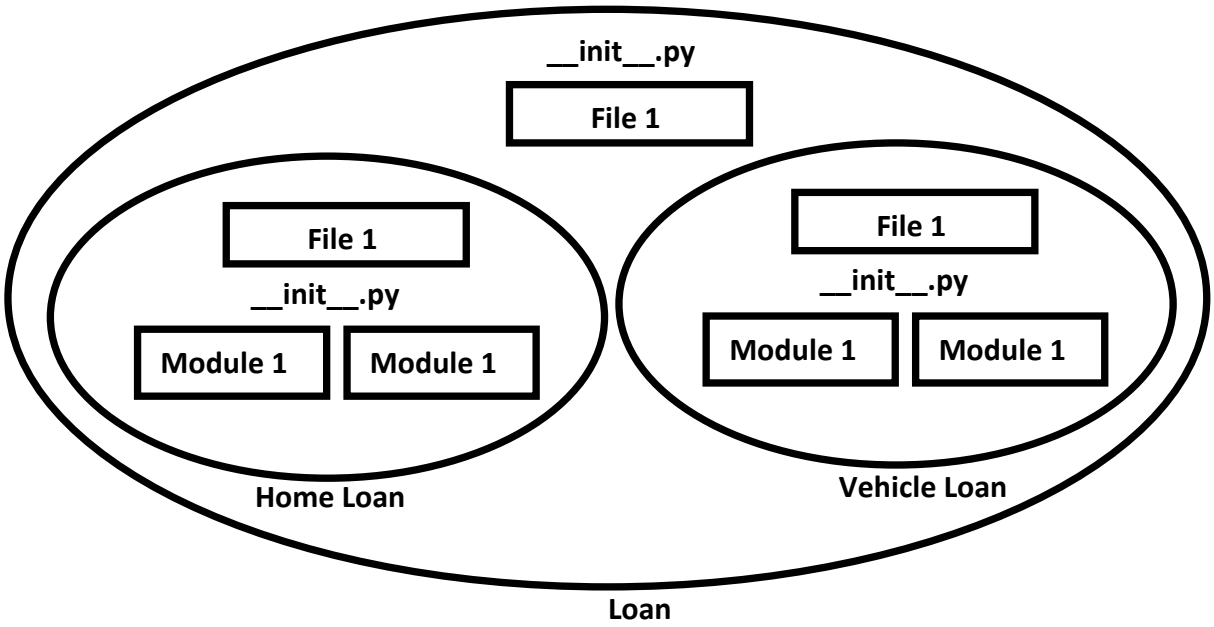
Learn Complete Python In Simple Way

PACKAGES

STUDY MATERIAL

- It is an encapsulation mechanism to group related modules into a single unit.
- package is nothing but folder or directory which represents collection of Python modules.
- Any folder or directory contains __init__.py file,is considered as a Python package.This file can be empty.
- A package can contains sub packages also.





The main advantages of package statement are

1) We can resolve naming conflicts
2) We can identify our components uniquely

3) It improves modularity of the application

Eg 1:

```
D:\Python_classes>
      |-test.py
      |-pack1
            |-module1.py
            |-__init__.py
```

__init__.py:

empty file

module1.py:

```
def f1():
   print("Hello this is from module1 present in pack1")
```

test.py (version-1):

```
import pack1.module1
pack1.module1.f1()
```

test.py (version-2):

```
from pack1.module1 import f1
f1()
```

Eg 2:

```
D:\Python_classes>
      |-test.py
      |-com
            |-module1.py
            |-__init__.py
               |-durgasoft
                  |-module2.py
                  |-__init__.py
```

__init__.py:

empty file

module1.py:

```
def f1():
        print("Hello this is from module1 present in com")
```

module2.py:

```
def f2():
        print("Hello this is from module2 present in com.durgasoft")
```

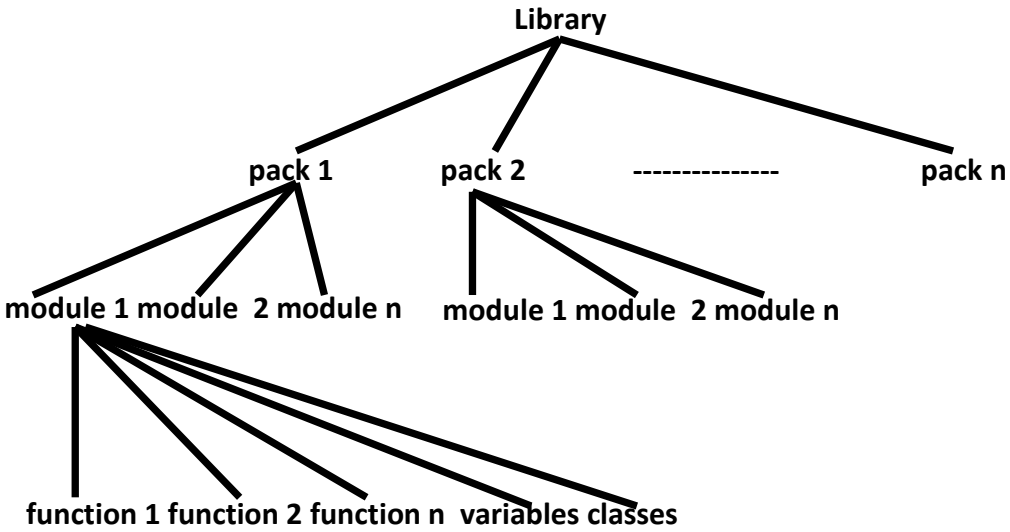
test.py

```
1) from com.module1 import f1
2) from com.durgasoft.module2 import f2
3) f1()
4) f2()
```

Output

```
D:\Python_classes>py test.py
Hello this is from module1 present in com
Hello this is from module2 present in com.durgasoft
```

Note: Summary diagram of library, packages, modules which contains functions, classes and variables.



Learn Complete Python In Simple Way

OOP's Part - 1

STUDY MATERIAL

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:
  ''' documenttation 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.

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

1) Instance Methods
2) Class Methods
3) Static Methods

Example for Class:

```
1)  class Student:
2)      '''Developed by durga for python demo'''
3)      def __init__(self):
4)          self.name='durga'
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)
11)         print("My Marks are:",self.marks)
```

What is Object:

Pysical 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("Durga",101,80)
14) s1.talk()
```

Output:

```
D:\durgaclasses>py test.py
Hello My Name is: Durga
My Rollno is: 101
My Marks are: 80
```

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 demonistrate 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...
```

Program:

```
1)  class Student:
2)
3)    ''' This is student class with required data'''
4)    def __init__(self,x,y,z):
5)       self.name=x
6)       self.rollno=y
7)       self.marks=z
8)
9)    def display(self):
10)      print("Student Name:{}\nRollno:{} \nMarks:{}".format(self.name,self.rollno,self
     .marks))
11)
12) s1=Student("Durga",101,80)
13) s1.display()
14) s2=Student("Sunny",102,100)
15) s2.display()
```

Output

Student Name:Durga
Rollno:101
Marks:80

Student Name:Sunny
Rollno:102
Marks:100

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.

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

Output: {'eno': 100, 'ename': 'Durga', '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.

```
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}

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)     def m1(self):
7)        self.c=30
8)
9)  t=Test()
10) t.m1()
11) t.d=40
12) 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.

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

```
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.

```
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)  t1=Test()
9)  t2=Test()
10) del t1.a
11) print(t1.__dict__)
12) 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.

```
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
```

2) 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 types 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.

Instance Variable vs Static Variable:

Note: In the case of instance variables for every object a seperate copy will be created,but in the case of static variables for total class only one copy will be created and shared by every object of that class.

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

```
12) print('t1:',t1.x,t1.y)
13) print('t2:',t2.x,t2.y)
```

Output

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

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__)
```

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 classname

```
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.

```
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.

```
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
```

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

Output

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

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

Output

```
888 999
888 20
```

```
1)   class Test:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      def m1(self):
6)          self.a=888
7)          self.b=999
8)
9)  t1=Test()
10) t2=Test()
11) t1.m1()
12) print(t1.a,t1.b)
13) print(t2.a,t2.b)
```

Output

```
888 999
10 20
```

```
1)   class Test:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      @classmethod
```

```
6)      def m1(cls):
7)          cls.a=888
8)          cls.b=999
9)
10) t1=Test()
11) t2=Test()
12) t1.m1()
13) print(t1.a,t1.b)
14) print(t2.a,t2.b)
15) print(Test.a,Test.b)
```

Output

888 20
888 20
888 999

How to Delete Static Variables of a Class:

1) We can delete static variables from anywhere by using the following syntax
   `del classname.variablename`

2) 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.

```
1)  import sys
2)  class Customer:
3)     ''' Customer class with bank operations.. '''
4)     bankname='DURGABANK'
```

```
5)     def __init__(self,name,balance=0.0):
6)         self.name=name
7)         self.balance=balance
8)     def deposit(self,amt):
9)         self.balance=self.balance+amt
10)        print('Balance after deposit:',self.balance)
11)    def withdraw(self,amt):
12)        if amt>self.balance:
13)            print('Insufficient Funds..cannot perform this operation')
14)            sys.exit()
15)        self.balance=self.balance-amt
16)        print('Balance after withdraw:',self.balance)
17)
18) print('Welcome to',Customer.bankname)
19) name=input('Enter Your Name:')
20) c=Customer(name)
21) while True:
22)    print('d-Deposit \nw-Withdraw \ne-exit')
23)    option=input('Choose your option:')
24)    if option=='d' or option=='D':
25)        amt=float(input('Enter amount:'))
26)        c.deposit(amt)
27)    elif option=='w' or option=='W':
28)        amt=float(input('Enter amount:'))
29)        c.withdraw(amt)
30)    elif option=='e' or option=='E':
31)        print('Thanks for Banking')
32)        sys.exit()
33)    else:
34)        print('Invalid option..Plz choose valid option')
```

Output:

```
D:\durga_classes>py test.py
Welcome to DURGABANK
Enter Your Name:Durga
d-Deposit
w-Withdraw
e-exit

Choose your option:d
Enter amount:10000
Balance after deposit: 10000.0
d-Deposit
w-Withdraw
```

```
e-exit

Choose your option:d
Enter amount:20000
Balance after deposit: 30000.0
d-Deposit
w-Withdraw
e-exit

Choose your option:w
Enter amount:2000
Balance after withdraw: 28000.0
d-Deposit
w-Withdraw
e-exit

Choose your option:r
Invalid option..Plz choose valid option
d-Deposit
w-Withdraw
e-exit

Choose your option:e
Thanks for Banking
```

3) 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.

```
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
```

```
1)  class Test:
2)      def m1(self):
3)          a=1000
4)          print(a)
5)      def m2(self):
6)          b=2000
7)          print(a)    #NameError: name 'a' is not defined
8)          print(b)
9)  t=Test()
10) t.m1()
11) t.m2()
```

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:\durga_classes>py test.py
Enter number of students:2
Enter Name:Durga
Enter Marks:90
Hi Durga
Your Marks are: 90
You got First Grade

Enter Name:Ravi
Enter Marks:12
Hi Ravi
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
```

```
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_classes>py test.py
Enter number of students:2

Enter Name:Durga
Enter Marks:100
```

```
Hi Durga
Your Marks are: 100

Enter Name:Ravi
Enter Marks:80
Hi Ravi
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.

```
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_classes>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 DurgaMath:
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) DurgaMath.add(10,20)
16) DurgaMath.product(10,20)
17) DurgaMath.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.

Passing Members of One Class to Another Class:

We can access members of one class inside another class.

```
1)  class Employee:
2)      def __init__(self,eno,ename,esal):
3)          self.eno=eno
4)          self.ename=ename
5)          self.esal=esal
6)      def display(self):
7)          print('Employee Number:',self.eno)
8)          print('Employee Name:',self.ename)
9)          print('Employee Salary:',self.esal)
10) class Test:
11)     def modify(emp):
12)         emp.esal=emp.esal+10000
13)         emp.display()
14) e=Employee(100,'Durga',10000)
15) Test.modify(e)
```

Output

```
D:\python_classes>py test.py
Employee Number: 100
Employee Name: Durga
Employee Salary: 20000
```

In the above application, Employee class members are available to Test class.

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:
    ......
```

Example: 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()
```

Demo Program-2:

```
1)  class Person:
2)      def __init__(self):
3)          self.name='durga'
4)          self.db=self.Dob()
5)      def display(self):
6)          print('Name:',self.name)
7)      class Dob:
8)          def __init__(self):
9)              self.dd=10
10)             self.mm=5
11)             self.yy=1947
12)         def display(self):
13)             print('Dob={}/{}/{}'.format(self.dd,self.mm,self.yy))
14) p=Person()
15) p.display()
16) x=p.db
17) x.display()
```

Output

Name: durga
Dob=10/5/1947

Demo Program-3:

Inside a class we can declare any number of inner classes.

```
1)  class Human:
2)
3)      def __init__(self):
4)          self.name = 'Sunny'
5)          self.head = self.Head()
6)          self.brain = self.Brain()
7)      def display(self):
8)          print("Hello..",self.name)
9)
10)     class Head:
11)         def talk(self):
12)             print('Talking...')
13)
14)     class Brain:
15)         def think(self):
16)             print('Thinking...')
17)
18) h=Human()
19) h.display()
20) h.head.talk()
21) h.brain.think()
```

Output

Hello.. Sunny
Talking...
Thinking...

Garbage Collection

- In old languages like C++, programmer is responsible for both creation and destruction of objects.Usually programmer taking very much care while creating object, but nEglecting destruction of useless objects. Because of his nEglectance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.

- But in Python, We have some assistant which is always running in the background to destroy useless objects.Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.
- Hence the main objective of Garbage Collector is to destroy useless objects.

- If an object does not have any reference variable then that object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our Program:

By default Gargbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.

1) gc.isenabled() ➔ Returns True if GC enabled

2) gc.disable()➔ To disable GC explicitly

3) gc.enable()➔ To enable GC explicitly

```
1) import gc
2) print(gc.isenabled())
3) gc.disable()
4) print(gc.isenabled())
5) gc.enable()
6) print(gc.isenabled())
```

Output

```
True
False
True
```

Destructors:

- Destructor is a special method and the name should be __del__
- Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).
- Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities.

```
1)  import time
2)  class Test:
3)      def __init__(self):
4)          print("Object Initialization...")
5)      def __del__(self):
6)          print("Fulfilling Last Wish and performing clean up activities...")
7)
8)  t1=Test()
9)  t1=None
10) time.sleep(5)
11) print("End of application")
```

Output

```
Object Initialization...
Fulfilling Last Wish and performing clean up activities...
End of application
```

Note: If the object does not contain any reference variable then only it is eligible fo GC. ie if the reference count is zero then only object eligible for GC.

```
1)  import time
2)  class Test:
3)      def __init__(self):
4)          print("Constructor Execution...")
5)      def __del__(self):
6)          print("Destructor Execution...")
7)  t1=Test()
8)  t2=t1
9)  t3=t2
10) del t1
11) time.sleep(5)
12) print("object not yet destroyed after deleting t1")
13) del t2
```

```
14) time.sleep(5)
15) print("object not yet destroyed even after deleting t2")
16) print("I am trying to delete last reference variable...")
17) del t3
```

Example:

```
1)  import time
2)  class Test:
3)      def __init__(self):
4)          print("Constructor Execution...")
5)      def __del__(self):
6)          print("Destructor Execution...")
7)  list=[Test(),Test(),Test()]
8)  del list
9)  time.sleep(5)
10) print("End of application")
```

Output

```
Constructor Execution...
Constructor Execution...
Constructor Execution...
Destructor Execution...
Destructor Execution...
Destructor Execution...
End of application
```

How to find the Number of References of an Object:

sys module contains getrefcount() function for this purpose.

sys.getrefcount (objectreference)

```
1)  import sys
2)  class Test:
3)      pass
4)  t1=Test()
5)  t2=t1
6)  t3=t1
7)  t4=t1
8)  print(sys.getrefcount(t1))
```

Output 5

Note: For every object, Python internally maintains one default reference variable self.

Learn Complete Python In Simple Way

EXCEPTION HANDLING

STUDY MATERIAL

In any programming language there are 2 types of errors are possible.

1) Syntax Errors
2) Runtime Errors

1) Syntax Errors:

The errors which occur because of invalid syntax are called syntax errors.

Eg 1:

```
x = 10
if x == 10
  print("Hello")
```

SyntaxError: invalid syntax

Eg 2:

```
print "Hello"
```

SyntaxError: Missing parentheses in call to 'print'

Note: Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2) Runtime Errors:

- Also known as exceptions.
- While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.

Eg:

1) print(10/0) → ZeroDivisionError: division by zero
2) print(10/"ten") → TypeError: unsupported operand type(s) for /: 'int' and 'str'

3)
```
x = int(input("Enter Number:"))
print(x)
```

```
D:\Python_classes>py test.py
Enter Number:ten
ValueError: invalid literal for int() with base 10: 'ten'
```

Note: Exception Handling concept applicable for Runtime Errors but not for syntax errors

What is Exception?

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

- **ZeroDivisionError**
- **TypeError**
- **ValueError**
- **FileNotFoundError**
- **EOFError**
- **SleepingError**
- **TyrePuncturedError**

It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program (i.e we should not block our resources and we should not miss anything)

Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Eg: For example our programming requirement is reading data from remote file locating at London. At runtime if London file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

try:

```
Read Data from Remote File locating at London.
except FileNotFoundError:
use local file and continue rest of the program  normally
```

Q. What is an Exception?
Q. What is the purpose of Exception Handling?
Q. What is the meaning of Exception Handling?

Default Exception Handing in Python:

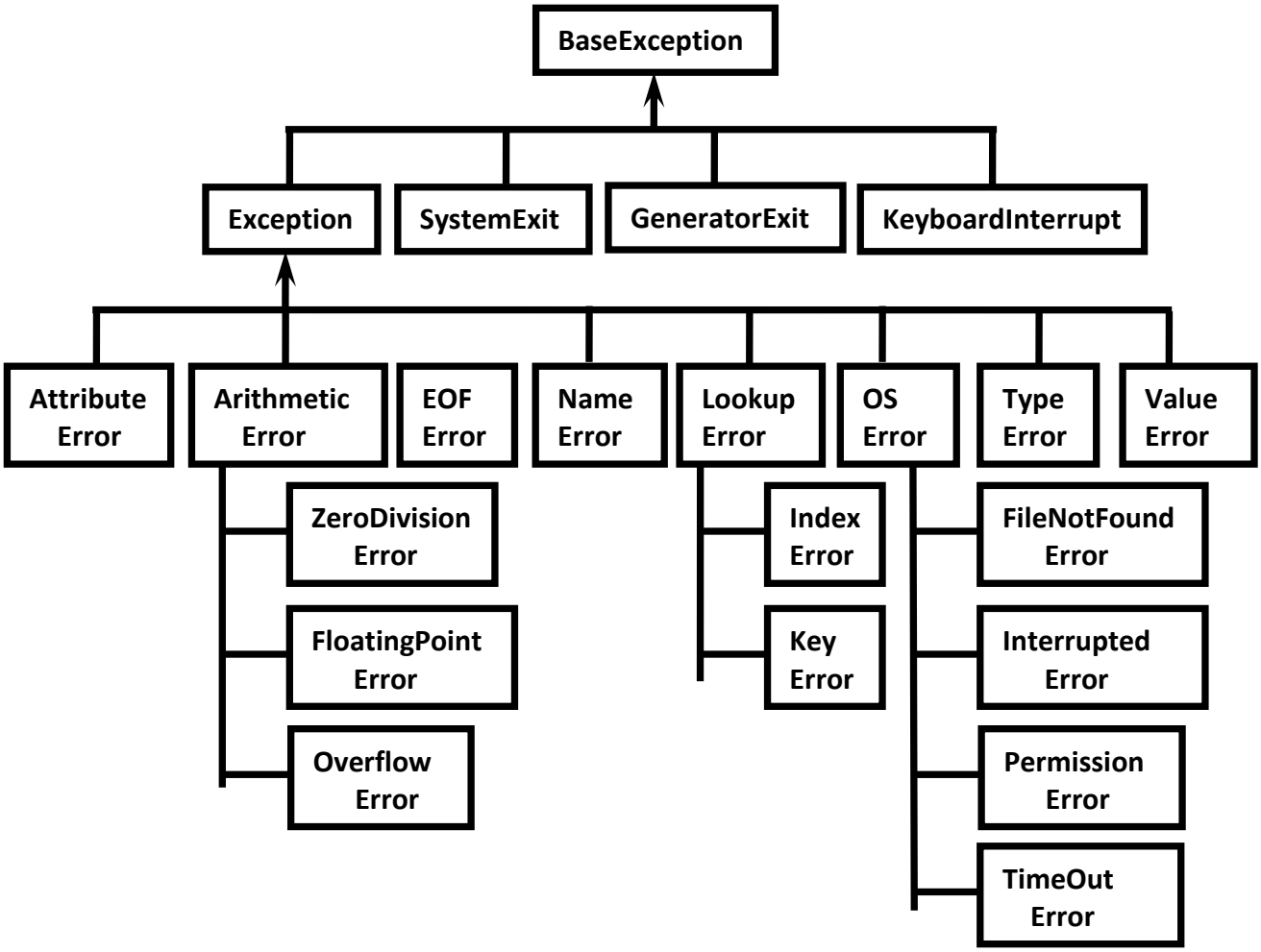
- **Every exception in Python is an object. For every exception type the corresponding classes are available.**
- **Whevever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.**

- The rest of the program won't be executed.

```
1) print("Hello")
2) print(10/0)
3) print("Hi")
```

```
D:\Python_classes>py test.py
Hello
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Python's Exception Hierarchy



- Every Exception in Python is a class.
- All exception classes are child classes of BaseException.i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy.
- Most of the times being a programmer we have to concentrate Exception and its child classes.

Customized Exception Handling by using try-except:

- It is highly recommended to handle exceptions.
- The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.

```
try:
    Risky Code
except XXX:
    Handling code/Alternative Code
```

Without try-except:

```
1) print("stmt-1")
2) print(10/0)
3) print("stmt-3")
```

Output

```
stmt-1
ZeroDivisionError: division by zero
Abnormal termination/Non-Graceful Termination
```

With try-except:

```
1) print("stmt-1")
2) try:
3)     print(10/0)
4) except ZeroDivisionError:
5)     print(10/2)
6) print("stmt-3")
```

Output

```
stmt-1
5.0
stmt-3
Normal termination/Graceful Termination
```

Control Flow in try-except:

```
try:
        stmt-1
        stmt-2
        stmt-3
except XXX:
      stmt-4
stmt-5
```

Case-1: If there is no exception
1,2,3,5 and Normal Termination

Case-2: If an exception raised at stmt-2 and corresponding except block matched
1,4,5 Normal Termination

Case-3: If an exception rose at stmt-2 and corresponding except block not matched
1, Abnormal Termination

Case-4: If an exception rose at stmt-4 or at stmt-5 then it is always abnormal termination.

Conclusions:

1) Within the try block if anywhere exception raised then rest of the try block won't be executed eventhough we handled that exception. Hence we have to take only risky code inside try block and length of the try block should be as less as possible.
2) In addition to try block, there may be a chance of raising exceptions inside except and finally blocks also.
3) If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to Print Exception Information:

try:

```
1) print(10/0)
2) except ZeroDivisionError as msg:
3)     print("exception raised and its description is:",msg)
```

Output exception raised and its description is: division by zero

try with Multiple except Blocks:

The way of handling exception is varied from exception to exception.Hence for every exception type a seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

Eg:

```
try:
    -------
    -------
    -------
except ZeroDivisionError:
        perform alternative arithmetic operations
except FileNotFoundError:
      use local file instead of remote file
```

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

```
1) try:
2)     x=int(input("Enter First Number: "))
3)     y=int(input("Enter Second Number: "))
4)     print(x/y)
5) except ZeroDivisionError :
6)     print("Can't Divide with Zero")
7) except ValueError:
8)     print("please provide int value only")
```

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 2
5.0

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
please provide int value only
```

If try with multiple except blocks available then the order of these except blocks is important .Python interpreter will always consider from top to bottom until matched except block identified.

```
1)  try:
2)      x=int(input("Enter First Number: "))
3)      y=int(input("Enter Second Number: "))
4)      print(x/y)
5)  except ArithmeticError :
6)      print("ArithmeticError")
7)  except ZeroDivisionError:
8)      print("ZeroDivisionError")
```

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
ArithmeticError
```

Single except Block that can handle Multiple Exceptions:

We can write a single except block that can handle multiple different types of exceptions.

```
except (Exception1,Exception2,exception3,..):    OR
except (Exception1,Exception2,exception3,..) as msg :
```

Parentheses are mandatory and this group of exceptions internally considered as tuple.

```
1)  try:
2)      x=int(input("Enter First Number: "))
3)      y=int(input("Enter Second Number: "))
4)      print(x/y)
5)  except (ZeroDivisionError,ValueError) as msg:
6)      print("Plz Provide valid numbers only and problem is: ",msg)
```

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Plz Provide valid numbers only and problem is:  division by zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Plz Provide valid numbers only and problem is:  invalid literal for int() with b are 10: 'ten'
```

Default except Block:

We can use default except block to handle any type of exceptions.
In default except block generally we can print normal error messages.

Syntax:

```
except:
    statements
```

```
1)  try:
2)      x=int(input("Enter First Number: "))
3)      y=int(input("Enter Second Number: "))
4)      print(x/y)
5)  except ZeroDivisionError:
6)      print("ZeroDivisionError:Can't divide with zero")
7)  except:
8)      print("Default Except:Plz provide valid input only")
```

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Default Except:Plz provide valid input only
```

***Note: If try with multiple except blocks available then default except block should be last, otherwise we will get SyntaxError.

```
1)  try:
2)      print(10/0)
3)  except:
4)      print("Default Except")
5)  except ZeroDivisionError:
6)      print("ZeroDivisionError")
```

```
SyntaxError: default 'except:' must be last
```

Note: The following are various possible combinations of except blocks

1) except ZeroDivisionError:
2) except ZeroDivisionError as msg:
3) except (ZeroDivisionError,ValueError) :

4) except (ZeroDivisionError,ValueError) as msg:
5) except :

finally Block:

- It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.
- It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.
- Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.
- Hence the main purpose of finally block is to maintain clean up code.

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup code
```

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

```
1) try:
2)     print("try")
3) except:
4)     print("except")
5) finally:
6)     print("finally")
```

Output

```
try
finally
```

Case-2: If there is an exception raised but handled

```
1) try:
2)     print("try")
3)     print(10/0)
```

```
except ZeroDivisionError:
    print("except")
finally:
    print("finally")
```

Output

```
try
except
finally
```

Case-3: If there is an exception raised but not handled

```
try:
    print("try")
    print(10/0)
except NameError:
    print("except")
finally:
    print("finally")
```

Output

```
try
finally
ZeroDivisionError: division by zero(Abnormal Termination)
```

*** **Note:** There is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.

Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown.In this particular case finally won't be executed.

```
imports
try:
    print("try")
    os._exit(0)
except NameError:
    print("except")
finally:
    print("finally")
```

Output: try

Note:

os._exit(0)

Where 0 represents status code and it indicates normal termination

There are multiple status codes are possible.

Control Flow in try-except-finally:

```
try:
        stmt-1
        stmt-2
        stmt-3
except:
        stmt-4
finally:
        stmt-5
        stmt-6
```

Case-1: **If there is no exception**

1,2,3,5,6 Normal Termination

Case-2: **If an exception raised at stmt2 and the corresponding except block matched**

1,4,5,6 Normal Termination

Case-3: **If an exception raised at stmt2 but the corresponding except block not matched**

1,5 Abnormal Termination

Case-4: **If an exception raised at stmt4 then it is always abnormal termination but before that finally block will be executed.**

Case-5: **If an exception raised at stmt-5 or at stmt-6 then it is always abnormal termination**

Nested try-except-finally Blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of try-except-finally is possible.

```
try:
    ----------
    ----------
    ----------
    try:
            ----------
            ----------
            ----------
```

except:

---

except:

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

```
1)  try:
2)      print("outer try block")
3)      try:
4)          print("Inner try block")
5)          print(10/0)
6)      except ZeroDivisionError:
7)          print("Inner except block")
8)      finally:
9)          print("Inner finally block")
10) except:
11)     print("outer except block")
12) finally:
13)     print("outer finally block")
```

Output

```
outer try block
Inner try block
Inner except block
Inner finally block
outer finally block
```

Control Flow in nested try-except-finally:

try:

stmt-1

stmt-2

stmt-3

try:

stmt-4

```
                    stmt-5
                    stmt-6
            except X:
                    stmt-7
            finally:
                    stmt-8
                    stmt-9
    except Y:
            stmt-10
    finally:
            stmt-11
            stmt-12
```

Case-1: If there is no exception
1,2,3,4,5,6,8,9,11,12 Normal Termination

Case-2: If an exception raised at stmt-2 and the corresponding except block matched
1,10,11,12 Normal Termination

Case-3: If an exceptiion raised at stmt-2 and the corresponding except block not matched
1,11,Abnormal Termination

Case-4: If an exception raised at stmt-5 and inner except block matched
1,2,3,4,7,8,9,11,12 Normal Termination

Case-5: If an exception raised at stmt-5 and inner except block not matched but outer except block matched 1,2,3,4,8,10,11,12,Normal Termination

Case-6: If an exception raised at stmt-5 and both inner and outer except blocks are not matched 1,2,3,4,8,11,Abnormal Termination

Case-7: If an exception raised at stmt-7 and corresponding except block matched
1,2,3,.,.,.,8,10,11,12,Normal Termination

Case-8: If an exception raised at stmt-7 and corresponding except block not matched
1,2,3,.,.,.,8,11,Abnormal Termination

Case-9: If an exception raised at stmt-8 and corresponding except block matched
1,2,3,.,.,.,.,10,11,12 Normal Termination

Case-10: If an exception raised at stmt-8 and corresponding except block not matched
1,2,3,.,.,.,.,11,Abnormal Termination

Case-11: If an exception raised at stmt-9 and corresponding except block matched
1,2,3,.,.,.,.,8,10,11,12,Normal Termination

Case-12: If an exception raised at stmt-9 and corresponding except block not matched
1,2,3,.,.,.,.,8,11,Abnormal Termination

Case-13: If an exception raised at stmt-10 then it is always abonormal termination but before abnormal termination finally block(stmt-11) will be executed.

Case-14: If an exception raised at stmt-11 or stmt-12 then it is always abnormal termination.

Note: If the control entered into try block then compulsary finally block will be executed. If the control not entered into try block then finally block won't be executed.

else Block with try-except-finally:

We can use else block with try-except-finally blocks.
else block will be executed if and only if there are no exceptions inside try block.

```
try:
        Risky Code
except:
        will be executed if exception inside try

else:
        will be executed if there is no exception inside try
finally:
        will be executed whether exception raised or not raised and handled or not
handled
```

Eg:

```
try:
        print("try")
        print(10/0) → 1
except:
        print("except")
else:
        print("else")
finally:
        print("finally")
```

If we comment line-1 then else block will be executed b'z there is no exception inside try.

In this case the output is:
try
else
finally

If we are not commenting line-1 then else block won't be executed b'z there is exception inside try block. In this case output is:
try
except
finally

Various possible Combinations of try-except-else-finally:

1) **Whenever we are writing try block, compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.**
2) **Wheneever we are writing except block, compulsory we should write try block. i.e except without try is always invalid.**
3) **Whenever we are writing finally block, compulsory we should write try block. i.e finally without try is always invalid.**
4) **We can write multiple except blocks for the same try,but we cannot write multiple finally blocks for the same try**
5) **Whenever we are writing else block compulsory except block should be there. i.e without except we cannot write else block.**
6) **In try-except-else-finally order is important.**
7) **We can define try-except-else-finally inside try, except, else and finally blocks. i.e nesting of try-except-else-finally is always possible.**

1	try: print("try")	✗
2	except: print("Hello")	✗
3	else: print("Hello")	✗
4	finally: print("Hello")	✗

5	try:
	  print("try")
	except:
	  print("except")
		✓

6	try:
	  print("try")
	finally:
	  print("finally")
		✓

7	try:
	print("try")
	except:
	print("except")
	else:
	print("else")
		✓

8	try:
	print("try")
	else:
	print("else")
		✗

9	try:
	print("try")
	else:
	print("else")
	finally:
	print("finally")
		✗

10	try: print("try") except XXX: print("except-1") except YYY: print("except-2")	✓

11	try:
	print("try")
	except :
	print("except-1")
	else:
	print("else")
	else:
	print("else")
		✗

12	try: print("try") except : print("except-1") finally: print("finally") finally: print("finally")	✗
13	try: print("try") print("Hello") except:	✗

	print("except")	
14	try: print("try") except: print("except") print("Hello") except: print("except")	✗
15	try: print("try") except: print("except") print("Hello") finally: print("finally")	✗

16	try:
	print("try")
	except:
	print("except")
	print("Hello")
	else:
	print("else")
		✗

17	try: print("try") except: print("except") try: print("try") except: print("except")	✓
18	try: print("try") except: print("except") try: print("try") finally: print("finally")	✓
19	try: print("try") except: print("except")	✓

	if 10>20:
	print("if")
	else:
	print("else")	

20	try: print("try") try: print("inner try") except: print("inner except block") finally: print("inner finally block") except: print("except")	✓
21	try: print("try") except: print("except") try: print("inner try") except: print("inner except block") finally: print("inner finally block")	✓
22	try: print("try") except: print("except") finally: try: print("inner try") except: print("inner except block") finally: print("inner finally block")	✓

23	try:
	print("try")
	except:
	print("except")
	try:
	print("try")
	else:
		✗

	print("else")	
24	try: print("try") try: print("inner try") except: print("except")	✗

25	try:
	print("try")
	else:
	print("else")
	except:
	print("except")
	finally:
	print("finally")
		✗

Types of Exceptions:

In Python there are 2 types of exceptions are possible.

1) Predefined Exceptions
2) User Definded Exceptions

1) Predefined Exceptions:

- Also known as inbuilt exceptions.
- The exceptions which are raised automatically by Python virtual machine whenver a particular event occurs are called pre defined exceptions.

Eg 1: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.

```
print(10/0)
```

Eg 2: Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

```
x=int("ten") → ValueError
```

2) User Defined Exceptions:

- Also known as Customized Exceptions or Programatic Exceptions
- Some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called User Defined Exceptions or Customized Exceptions

- **Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.**

  Eg:
  - **InSufficientFundsException**
  - **InvalidInputException**
  - **TooYoungException**
  - **TooOldException**

How to Define and Raise Customized Exceptions:

Every exception in Python is a class that extends Exception class either directly or indirectly.

Syntax:

```
class classname(predefined exception class name):
    def __init__(self,arg):
        self.msg=arg
```

```
1) class TooYoungException(Exception):
2)     def __init__(self,arg):
3)         self.msg=arg
```

TooYoungException is our class name which is the child class of Exception

We can raise exception by using raise keyword as follows
raise TooYoungException("message")

```
1) class TooYoungException(Exception):
2)     def __init__(self,arg):
3)         self.msg=arg
4)
5) class TooOldException(Exception):
6)     def __init__(self,arg):
7)         self.msg=arg
8)
9) age=int(input("Enter Age:"))
10) if age>60:
11)     raise TooYoungException("Plz wait some more time you will get best match soon!!!")
12) elif age<18:
13)     raise TooOldException("Your age already crossed marriage age...no chance of
    getting marriage")
14) else:
15)     print("You will get match details soon by email!!!")
```

D:\Python_classes>py test.py
Enter Age:90
__main__.TooYoungException: Plz wait some more time you will get best match soon!!!

D:\Python_classes>py test.py
Enter Age:12
__main__.TooOldException: Your age already crossed marriage age...no chance of g
etting marriage

D:\Python_classes>py test.py
Enter Age:27
You will get match details soon by email!!!

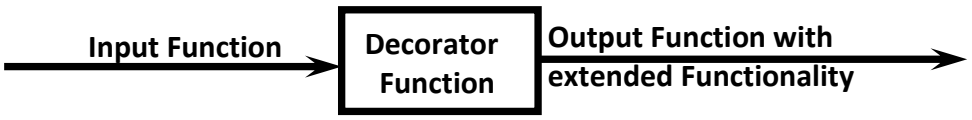
Note: raise keyword is best suitable for customized exceptions but not for pre defined exceptions

Learn Complete Python In Simple Way

DECORATOR FUNCTIONS

STUDY MATERIAL

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.

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

This function can always print same output for any name

Hello Durga Good Morning
Hello Ravi Good Morning
Hello Sunny 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.

```
1)  def decor(func):
2)      def inner(name):
3)          if name=="Sunny":
4)              print("Hello Sunny Bad Morning")
5)          else:
6)              func(name)
7)      return inner
8)
9)  @decor
10) def wish(name):
11)     print("Hello",name,"Good Morning")
12)
13) wish("Durga")
14) wish("Ravi")
15) wish("Sunny")
```

Output

Hello Durga Good Morning
Hello Ravi Good Morning
Hello Sunny Bad Morning
In the above program whenever we call wish() function automatically decor function will be executed.

How to call Same Function with Decorator and without Decorator:

We should not use @decor

```
1)  def decor(func):
2)      def inner(name):
3)          if name=="Sunny":
4)              print("Hello Sunny Bad Morning")
5)          else:
6)              func(name)
7)      return inner
8)
9)  def wish(name):
10)     print("Hello",name,"Good Morning")
11)
12) decorfunction=decor(wish)
13)
14) wish("Durga") #decorator wont be executed
15) wish("Sunny") #decorator wont be executed
16)
17) decorfunction("Durga")#decorator will be executed
18) decorfunction("Sunny")#decorator will be executed
```

Output

Hello Durga Good Morning
Hello Sunny Good Morning
Hello Durga Good Morning
Hello Sunny Bad Morning

```
1)  def  smart_division(func):
2)      def inner(a,b):
3)          print("We are dividing",a,"with",b)
4)          if b==0:
5)              print("OOPS...cannot divide")
6)              return
7)          else:
8)              return func(a,b)
```

```
9)      return inner
10)
11) @smart_division
12) def  division(a,b):
13)     return a/b
14) print(division(20,2))
15) print(division(20,0))
```

Without Decorator we will get Error. In this Case Output is:

```
10.0
Traceback (most recent call last):
File "test.py", line 16, in <module>
   print(division(20,0))
File "test.py", line 13, in division
   return a/b
ZeroDivisionError: division by zero
```

With Decorator we won't get any Error. In this Case Output is:

```
We are dividing 20 with 2
10.0
We are dividing 20 with 0
OOPS...cannot divide
None
```

```
1)  def marriagedecor(func):
2)      def inner():
3)          print('Hair decoration...')
4)          print('Face decoration with Platinum package')
5)          print('Fair and Lovely etc..')
6)          func()
7)      return inner
8)
9)  def getready():
10)     print('Ready for the marriage')
11)
12) decorated_getready=marriagedecor(getready)
13)
14) decorated_getready()
```

Decorator Chaining

We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:

```
@decor1
@decor
def num():
```

For num() function we are applying 2 decorator functions. First inner decorator will work and then outer decorator.

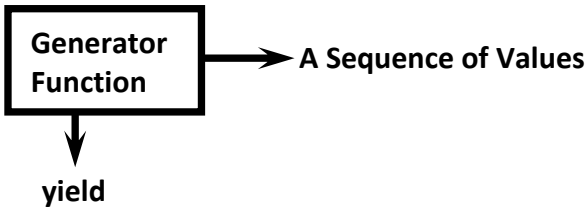
```
1)  def decor1(func):
2)      def inner():
3)          x=func()
4)          return x*x
5)      return inner
6)
7)  def decor(func):
8)      def inner():
9)          x=func()
10)         return 2*x
11)     return inner
12)
13) @decor1
14) @decor
15) def num():
16)     return 10
17)
18) print(num())
```

Learn Complete Python In Simple Way

GENERATOR FUNCTIONS

STUDY MATERIAL

Generator is a function which is responsible to generate a sequence of values.
We can write generator functions just like ordinary functions, but it uses yield keyword to return values.




```
1)  def mygen():
2)      yield 'A'
3)      yield 'B'
4)      yield 'C'
5)
6)  g=mygen()
7)  print(type(g))
8)
9)  print(next(g))
10) print(next(g))
11) print(next(g))
12) print(next(g))
```

Output

```
<class 'generator'>
A
B
C
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print(next(g))
StopIteration
```

```
1)  def countdown(num):
2)      print("Start Countdown")
3)      while(num>0):
4)          yield num
5)          num=num-1
6)  values=countdown(5)
7)  for x in values:
8)      print(x)
```

Output

```
Start Countdown
5
4
3
2
1
```

Eg 3: To generate first n numbers

```
1) def firstn(num):
2)     n=1
3)     while n<=num:
4)         yield n
5)         n=n+1
6)
7) values=firstn(5)
8) for x in values:
9)     print(x)
```

Output

```
1
2
3
4
5
```

We can convert generator into list as follows:

```
values = firstn(10)
l1 = list(values)
print(l1)    #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eg 4: To generate Fibonacci Numbers...
The next is the sum of previous 2 numbers

Eg: 0,1,1,2,3,5,8,13,21,...

```
1) def fib():
2)     a,b=0,1
3)     while True:
4)         yield a
5)         a,b=b,a+b
6) for f in fib():
```

```
7)    if f>100:
8)        break
9)    print(f)
```

Output

```
0
1
1
2
3
5
8
13
21
34
55
89
```

Advantages of Generator Functions:

1) When compared with Class Level Iterators, Generators are very easy to use.
2) Improves Memory Utilization and Performance.
3) Generators are best suitable for reading Data from Large Number of Large Files.
4) Generators work great for web scraping and crawling.

Generators vs Normal Collections wrt Performance:

```
1)  import random
2)  import time
3)
4)  names = ['Sunny','Bunny','Chinny','Vinny']
5)  subjects = ['Python','Java','Blockchain']
6)
7)  def people_list(num_people):
8)      results = []
9)      for i in range(num_people):
10)         person = {
11)                 'id':i,
12)                 'name': random.choice(names),
13)                 'subject':random.choice(subjects)
14)                }
15)         results.append(person)
16)     return results
```

```
17)
18) def people_generator(num_people):
19)     for i in range(num_people):
20)         person = {
21)                   'id':i,
22)                   'name': random.choice(names),
23)                   'major':random.choice(subjects)
24)                  }
25)         yield person
26)
27) """t1 = time.clock()
28) people = people_list(10000000)
29) t2 = time.clock()"""
30)
31) t1 = time.clock()
32) people = people_generator(10000000)
33) t2 = time.clock()
34)
35) print('Took {}'.format(t2-t1))
```

Note: In the above program observe the differnce wrt execution time by using list and generators

Generators vs Normal Collections wrt Memory Utilization:

Normal Collection:

```
l=[x*x for x in range(10000000000000000)]
print(l[0])
```

We will get MemoryError in this case because all these values are required to store in the memory.

Generators:

```
g=(x*x for x in range(10000000000000000))
print(next(g))
```

Output: 0

We won't get any MemoryError because the values won't be stored at the beginning

Learn Complete Python In Simple Way

ASSERTIONS

STUDY MATERIAL

Debugging Python Program by using assert Keyword:

- The process of identifying and fixing the bug is called debugging.
- Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug,compulsory we have to delete the extra added print() statments,otherwise these will be executed at runtime which creates performance problems and disturbs console output.
- To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.
- Hence the main purpose of assertions is to perform debugging. Usully we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of assert Statements:

There are 2 types of assert statements

1) Simple Version
2) Augmented Version

1) Simple Version:

assert conditional_expression

2) Augmented Version:

- assert conditional_expression, message
- conditional_expression will be evaluated and if it is true then the program will be continued.
- If it is false then the program will be terminated by raising AssertionError.
- By seeing AssertionError, programmer can analyze the code and can fix the problem.

```
1) def squareIt(x):
2)     return x**x
3) assert squareIt(2)==4,"The square of 2 should be 4"
4) assert squareIt(3)==9,"The square of 3 should be 9"
5) assert squareIt(4)==16,"The square of 4 should be 16"
6) print(squareIt(2))
7) print(squareIt(3))
8) print(squareIt(4))
9)
10) D:\Python_classes>py test.py
11) Traceback (most recent call last):
```

```
12)  File "test.py", line 4, in <module>
13)    assert squareIt(3)==9,"The square of 3 should be 9"
14) AssertionError: The square of 3 should be 9
15)
16) def squareIt(x):
17)    return x*x
18) assert squareIt(2)==4,"The square of 2 should be 4"
19) assert squareIt(3)==9,"The square of 3 should be 9"
20) assert squareIt(4)==16,"The square of 4 should be 16"
21) print(squareIt(2))
22) print(squareIt(3))
23) print(squareIt(4))
```

Output

```
4
9
16
```

Exception Handling vs Assertions:

Assertions concept can be used to alert programmer to resolve development time errors.
Exception Handling can be used to handle runtime errors.

Learn Complete Python In Simple Way

FILE HANDLING

STUDY MATERIAL

- As the part of programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files.
- Files are very common permanent storage areas to store our data.

Types of Files:

There are 2 types of files

1) **Text Files:**
   Usually we can use text files to store character data
   Eg: abc.txt

2) **Binary Files:**
   Usually we can use binary files to store binary data like images,video files, audio files etc...

Opening a File:

- Before performing any operation (like read or write) on the file,first we have to open that file.For this we should use Python's inbuilt function open()
- But at the time of open, we have to specify mode,which represents the purpose of opening file.

```
f = open(filename, mode)
```

The allowed modes in Python are

1) r → open an existing file for read operation. The file pointer is positioned at the beginning of the file.If the specified file does not exist then we will get FileNotFoundError.This is default mode.

2) w → open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already avaialble then this mode will create that file.

3) a → open an existing file for append operation. It won't override existing data.If the specified file is not already avaialble then this mode will create a new file.

4) r+ → To read and write data into the file. The previous data in the file will not be deleted.The file pointer is placed at the beginning of the file.
5) w+ → To write and read data. It will override existing data.

6) a+ → To append and read data from the file.It wont override existing data.

7) x → To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

Eg: rb,wb,ab,r+b,w+b,a+b,xb

```
f = open("abc.txt","w")
```

We are opening abc.txt file for writing data.

Closing a File:

After completing our operations on the file, it is highly recommended to close the file. For this we have to use close() function.

```
f.close()
```

Various Properties of File Object:

Once we opend a file and we got file object, we can get various details related to that file by using its properties.

- **name → Name of opened file**
- **mode → Mode in which the file is opened**
- **closed → Returns boolean value indicates that file is closed or not**
- **readable()→ Retruns boolean value indicates that whether file is readable or not**
- **writable()→ Returns boolean value indicates that whether file is writable or not.**

```
1) f=open("abc.txt",'w')
2) print("File Name: ",f.name)
3) print("File Mode: ",f.mode)
4) print("Is File Readable:  ",f.readable())
5) print("Is File Writable:  ",f.writable())
6) print("Is File Closed : ",f.closed)
7) f.close()
8) print("Is File Closed : ",f.closed)
```

Output

```
D:\Python_classes>py test.py
File Name:  abc.txt
File Mode:  w
Is File Readable: False
Is File Writable: True
Is File Closed: False
Is File Closed: True
```

Writing Data to Text Files:

We can write character data to the text files by using the following 2 methods.

1) write(str)
2) writelines(list of lines)

```
1) f=open("abcd.txt",'w')
2) f.write("Durga\n")
3) f.write("Software\n")
4) f.write("Solutions\n")
5) print("Data written to the file successfully")
6) f.close()
```

abcd.txt:

```
Durga
Software
Solutions
```

Note: In the above program, data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

```
f = open("abcd.txt","a")
```

Eg 2:

```
1) f=open("abcd.txt",'w')
2) list=["sunny\n","bunny\n","vinny\n","chinny"]
3) f.writelines(list)
4) print("List of lines written to the file successfully")
5) f.close()
```

abcd.txt:

```
sunny
bunny
vinny
chinny
```

Note: While writing data by using write() methods, compulsory we have to provide line seperator(\n), otherwise total data should be written to a single line.

Reading Character Data from Text Files:

We can read character data from text file by using the following read methods.

- **read()→ To read total data from the file**
- **read(n) → To read 'n' characters from the file**
- **readline()→ To read only one line**
- **readlines()→ To read all lines into a list**

Eg 1: To read total data from the file

```
1) f=open("abc.txt",'r')
2) data=f.read()
3) print(data)
4) f.close()
```

Output

```
sunny
bunny
chinny
vinny
```

Eg 2: To read only first 10 characters:

```
1) f=open("abc.txt",'r')
2) data=f.read(10)
3) print(data)
4) f.close()
```

Output

```
sunny
bunn
```

Eg 3: To read data line by line:

```
1) f=open("abc.txt",'r')
2) line1=f.readline()
3) print(line1,end='')
4) line2=f.readline()
5) print(line2,end='')
6) line3=f.readline()
7) print(line3,end='')
8) f.close()
```

Output

```
sunny
bunny
chinny
```

Eg 4: To read all lines into list:

```
1) f=open("abc.txt",'r')
2) lines=f.readlines()
3) for line in lines:
4)     print(line,end='')
5) f.close()
```

Output

```
sunny
bunny
chinny
vinny
```

Eg 5:

```
1) f=open("abc.txt","r")
2) print(f.read(3))
3) print(f.readline())
4) print(f.read(4))
5) print("Remaining data")
6) print(f.read())
```

Output

```
sun
ny

bunn
Remaining data
y
chinny
vinny
```

The with Statement:

- The with statement can be used while opening a file.We can use this to group file operation statements within a block.
- The advantage of with statement is it will take care closing of file,after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

```
1) with open("abc.txt","w") as f:
2)     f.write("Durga\n")
3)     f.write("Software\n")
4)     f.write("Solutions\n")
5)     print("Is File Closed: ",f.closed)
6) print("Is File Closed: ",f.closed)
```

Output

```
Is File Closed:  False
Is File Closed:  True
```

The seek() and tell() Methods:

tell():

- We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [can you plese telll current cursor position]
- The position(index) of first character in files is zero just like string index.

```
1) f=open("abc.txt","r")
2) print(f.tell())
3) print(f.read(2))
4) print(f.tell())
5) print(f.read(3))
6) print(f.tell())
```

abc.txt:

```
sunny
bunny
chinny
vinny
```

Output:

```
0
su
2
nny
5
```

seek():

We can use seek() method to move cursor (file pointer) to specified location.
[Can you please seek the cursor to a particular location]
f.seek(offset, fromwhere) → offset represents the number of positions
The allowed Values for 2nd Attribute (from where) are
0 → From beginning of File (Default Value)
1 → From Current Position
2 → From end of the File

Note: Python 2 supports all 3 values but Python 3 supports only zero.

```
1)  data="All Students are STUPIDS"
2)  f=open("abc.txt","w")
3)  f.write(data)
4)  with open("abc.txt","r+") as f:
5)      text=f.read()
6)      print(text)
7)      print("The Current Cursor Position: ",f.tell())
8)      f.seek(17)
9)      print("The Current Cursor Position: ",f.tell())
10)     f.write("GEMS!!!")
11)     f.seek(0)
12)     text=f.read()
13)     print("Data After Modification:")
14)     print(text)
```

Output

```
All Students are STUPIDS
The Current Cursor Position:  24
The Current Cursor Position:  17
Data After Modification:
All Students are GEMS!!!
```

How to check a particular File exists OR not?

We can use os library to get information about files in our computer.
os module has path sub module,which contains isFile() function to check whether a particular file exists or not?

```
os.path.isfile(fname)
```

Q) Write a Program to check whether the given File exists OR not. If it is available then print its content?

```
1)  import os,sys
2)  fname=input("Enter File Name: ")
3)  if os.path.isfile(fname):
4)      print("File exists:",fname)
5)      f=open(fname,"r")
6)  else:
7)      print("File does not exist:",fname)
8)      sys.exit(0)
9)  print("The content of file is:")
10) data=f.read()
11) print(data)
```

Output

```
D:\Python_classes>py test.py
Enter File Name: durga.txt
File does not exist: durga.txt
D:\Python_classes>py test.py
Enter File Name: abc.txt
File exists: abc.txt
The content of file is:
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
```

Note:

sys.exit(0) → To exit system without executing rest of the program.
argument represents status code. 0 means normal termination and it is the default value.

Q) Program to print the Number of Lines, Words and Characters present in the given File?

```
1)  import os,sys
2)  fname=input("Enter File Name: ")
3)  if os.path.isfile(fname):
4)      print("File exists:",fname)
5)      f=open(fname,"r")
6)  else:
7)      print("File does not exist:",fname)
8)      sys.exit(0)
9)  lcount=wcount=ccount=0
10) for line in f:
11)     lcount=lcount+1
12)     ccount=ccount+len(line)
13)     words=line.split()
14)     wcount=wcount+len(words)
15) print("The number of Lines:",lcount)
16) print("The number of Words:",wcount)
17) print("The number of Characters:",ccount)
```

Output

```
D:\Python_classes>py test.py
Enter File Name: durga.txt
File does not exist: durga.txt

D:\Python_classes>py test.py
Enter File Name: abc.txt
File exists: abc.txt
The number of Lines: 6
The number of Words: 24
The number of Characters: 149
```

abc.txt

```
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
```

Handling Binary Data:

It is very common requirement to read or write binary data like images,video files,audio files etc.

Q) Program to read Image File and write to a New Image File?

```
1) f1=open("rossum.jpg","rb")
2) f2=open("newpic.jpg","wb")
3) bytes=f1.read()
4) f2.write(bytes)
5) print("New Image is available with the name: newpic.jpg")
```

Handling CSV Files:

- **CSV → Comma seperated values**
- **As the part of programming, it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.**

Writing Data to CSV File:

```
1) import csv
2) with open("emp.csv","w",newline='') as f:
3)     w=csv.writer(f)  # returns csv writer object
4)     w.writerow(["ENO","ENAME","ESAL","EADDR"])
5)     n=int(input("Enter Number of Employees:"))
6)     for i in range(n):
7)         eno=input("Enter Employee No:")
8)         ename=input("Enter Employee Name:")
9)         esal=input("Enter Employee Salary:")
10)        eaddr=input("Enter Employee Address:")
11)        w.writerow([eno,ename,esal,eaddr])
12) print("Total Employees data written to csv file successfully")
```

Note: Observe the difference with newline attribute and without

```
with open("emp.csv","w",newline='') as f:
with open("emp.csv","w") as f:
```

Note: If we are not using newline attribute then in the csv file blank lines will be included between data. To prevent these blank lines, newline attribute is required in Python-3,but in Python-2 just we can specify mode as 'wb' and we are not required to use newline attribute.

Reading Data from CSV File:

```
1) import csv
2) f=open("emp.csv",'r')
3) r=csv.reader(f) #returns csv reader object
4) data=list(r)
5) #print(data)
6) for line in data:
7)     for word in line:
8)         print(word,"\t",end='')
9)     print()
```

Output

```
D:\Python_classes>py test.py

ENO   ENAME   ESAL   EADDR
100    Durga   1000   Hyd
200    Sachin  2000   Mumbai
300    Dhoni   3000   Ranchi
```

Zipping and Unzipping Files:

It is very common requirement to zip and unzip files.

The main advantages are:

1) To improve memory utilization
2) We can reduce transport time
3) We can improve performance.

To perform zip and unzip operations, Python contains one in-bulit module zip file. This module contains a class: ZipFile

To Create Zip File:

We have to create ZipFile class object with name of the zip file, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

```
f = ZipFile("files.zip","w","ZIP_DEFLATED")
```

Once we create ZipFile object,we can add files by using write() method.

```
f.write(filename)
```

```
1) from zipfile import *
2) f=ZipFile("files.zip",'w',ZIP_DEFLATED)
3) f.write("file1.txt")
4) f.write("file2.txt")
5) f.write("file3.txt")
6) f.close()
7) print("files.zip file created successfully")
```

To perform unzip Operation:

We have to create ZipFile object as follows

```
f = ZipFile("files.zip","r",ZIP_STORED)
```

ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify.

Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist() method.

```
names = f.namelist()
```

```
1) from zipfile import *
2) f=ZipFile("files.zip",'r',ZIP_STORED)
3) names=f.namelist()
4) for name in names:
5)     print( "File Name: ",name)
6)     print("The Content of this file is:")
7)     f1=open(name,'r')
8)     print(f1.read())
9)     print()
```

Working with Directories:

It is very common requirement to perform operations for directories like

1) To know current working directory
2) To create a new directory
3) To remove an existing directory
4) To rename a directory
5) To list contents of the directory

   etc...

To perform these operations, Python provides inbuilt module os,which contains several functions to perform directory related operations.

Q1) To Know Current Working Directory

```
import os
cwd = os.getcwd()
print("Current Working Directory:",cwd)
```

Q2) To Create a Sub Directory in the Current Working Directory

```
import os
os.mkdir("mysub")
print("mysub directory created in cwd")
```

Q3) To Create a Sub Directory in mysub Directory

```
cwd
    |-mysub
        |-mysub2
```

```
import os
os.mkdir("mysub/mysub2")
print("mysub2  created inside mysub")
```

Note: Assume mysub already present in cwd.

Q4) To Create Multiple Directories like sub1 in that sub2 in that sub3

```
import os
os.makedirs("sub1/sub2/sub3")
print("sub1 and in that sub2 and in that sub3 directories created")
```

Q5) To Remove a Directory

```
import os
os.rmdir("mysub/mysub2")
print("mysub2 directory deleted")
```

Q6) To Remove Multiple Directories in the Path

```
import os
os.removedirs("sub1/sub2/sub3")
print("All 3 directories sub1,sub2 and sub3 removed")
```

Q7) To Rename a Directory

```
import os
os.rename("mysub","newdir")
print("mysub directory renamed to newdir")
```

Q8) To know Contents of Directory

OS Module provides listdir() to list out the contents of the specified directory. It won't display the contents of sub directory.

```
1) import os
2) print(os.listdir("."))
```

Output

```
D:\Python_classes>py test.py
['abc.py', 'abc.txt', 'abcd.txt', 'com', 'demo.py', 'durgamath.py', 'emp.csv', '
file1.txt', 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'myl
og.txt', 'newdir', 'newpic.jpg', 'pack1', 'rossum.jpg', 'test.py', '__pycache__']
```

- The above program display contents of current working directory but not contents of sub directories.
- If we want the contents of a directory including sub directories then we should go for walk() function.

Q9) To Know Contents of Directory including Sub Directories

- We have to use walk() function
- [Can you please walk in the directory so that we can aware all contents of that directory]
- os.walk(path, topdown = True, onerror = None, followlinks = False)
- It returns an Iterator object whose contents can be displayed by using for loop

- path → Directory Path. cwd means .
- topdown = True → Travel from top to bottom
- onerror = None → On error detected which function has to execute.
- followlinks = True → To visit directories pointed by symbolic links

Eg: To display all contents of Current working directory including sub directories:

```
1) import os
2) for dirpath,dirnames,filenames  in os.walk('.'):
3)     print("Current Directory Path:",dirpath)
4)     print("Directories:",dirnames)
5)     print("Files:",filenames)
6)     print()
```

Output

```
Current Directory Path: .
Directories: ['com', 'newdir', 'pack1', '__pycache__']
```

Files: ['abc.txt', 'abcd.txt', 'demo.py', 'durgamath.py', 'emp.csv', 'file1.txt', 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'mylog.txt', 'newpic.jpg', 'rossum.jpg', 'test.py']

Current Directory Path: .\com
Directories: ['durgasoft', '__pycache__']
Files: ['module1.py', '__init__.py']
...

Note: To display contents of particular directory, we have to provide that directory name as argument to walk() function.

```
os.walk("directoryname")
```

Q) What is the difference between listdir() and walk() Functions?

In the case of listdir(), we will get contents of specified directory but not sub directory contents. But in the case of walk() function we will get contents of specified directory and its sub directories also.

Running Other Programs from Python Program:

OS Module contains system() function to run programs and commands.
It is exactly same as system() function in C language.

```
os.system("commad string")
```

The argument is any command which is executing from DOS.

Eg:

```
import os
os.system("dir *.py")
os.system("py abc.py")
```

How to get Information about a File:

We can get statistics of a file like size, last accessed time,last modified time etc by using stat() function of os module.

```
stats = os.stat("abc.txt")
```

The statistics of a file includes the following parameters:

1) st_mode → Protection Bits
2) st_ino → Inode number
3) st_dev → Device
4) st_nlink → Number of Hard Links
5) st_uid → User id of Owner
6) st_gid → Group id of Owner

7) st_size → Size of File in Bytes
8) st_atime → Time of Most Recent Access
9) st_mtime → Time of Most Recent Modification
10) st_ctime → Time of Most Recent Meta Data Change

Note: st_atime, st_mtime and st_ctime returns the time as number of milli seconds since Jan 1st 1970, 12:00 AM. By using datetime module from timestamp() function, we can get exact date and time.

Q) To Print all Statistics of File abc.txt

```
1) import os
2) stats=os.stat("abc.txt")
3) print(stats)
```

Output

```
os.stat_result(st_mode=33206, st_ino=844424930132788, st_dev=2657980798, st_nlin
k=1, st_uid=0, st_gid=0, st_size=22410, st_atime=1505451446, st_mtime=1505538999
, st_ctime=1505451446)
```

Q) To Print specified Properties

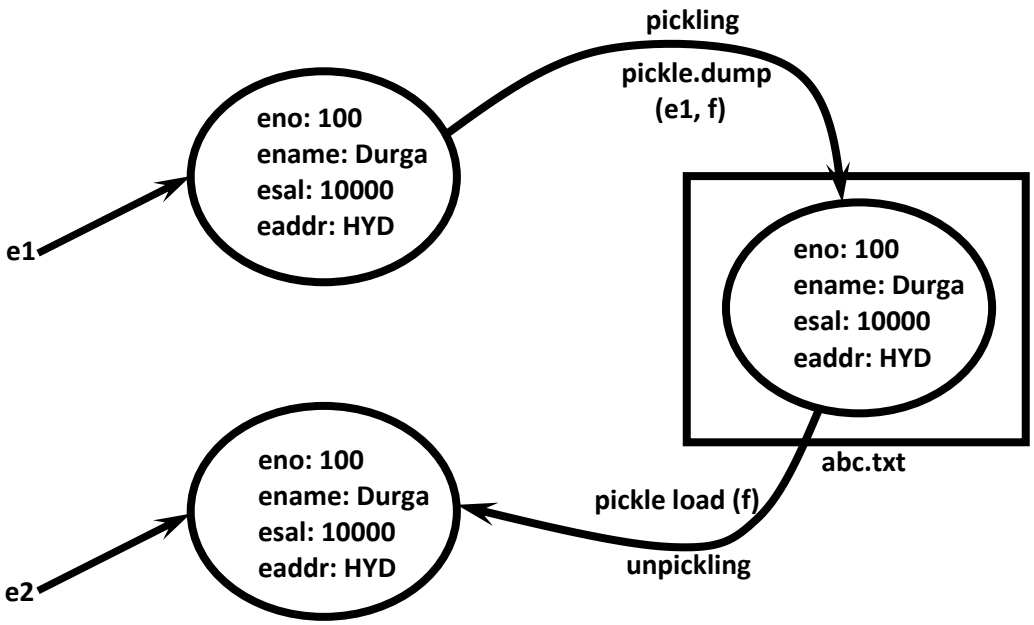
```
1) import os
2) from  datetime  import *
3) stats=os.stat("abc.txt")
4) print("File Size in Bytes:",stats.st_size)
5) print("File Last Accessed Time:",datetime.fromtimestamp(stats.st_atime))
6) print("File Last Modified Time:",datetime.fromtimestamp(stats.st_mtime))
```

Output

```
File Size in Bytes: 22410
File Last Accessed Time: 2017-09-15 10:27:26.599490
File Last Modified Time: 2017-09-16 10:46:39.245394
```

Pickling and Unpickling of Objects:

- Sometimes we have to write total state of object to the file and we have to read total object from the file.
- The process of writing state of object to the file is called pickling and the process of reading state of an object from the file is called unpickling.
- We can implement pickling and unpickling by using pickle module of Python.
- pickle module contains dump() function to perform pickling. `pickle.dump(object,file)`
- pickle module contains load() function to perform unpickling `obj=pickle.load(file)`



Writing and Reading State of Object by using pickle Module:

```
1)  import pickle
2)  class Employee:
3)      def __init__(self,eno,ename,esal,eaddr):
4)          self.eno=eno;
5)          self.ename=ename;
6)          self.esal=esal;
7)          self.eaddr=eaddr;
8)      def display(self):
9)          print(self.eno,"\t",self.ename,"\t",self.esal,"\t",self.eaddr)
10) with open("emp.dat","wb") as f:
11)     e=Employee(100,"Durga",1000,"Hyd")
12)     pickle.dump(e,f)
13)     print("Pickling of Employee Object completed...")
14)
15) with open("emp.dat","rb") as f:
16)     obj=pickle.load(f)
17)     print("Printing Employee Information after unpickling")
18)     obj.display()
```

Writing Multiple Employee Objects to the File:

emp.py:

```
1)  class Employee:
2)      def __init__(self,eno,ename,esal,eaddr):
3)          self.eno=eno;
4)          self.ename=ename;
5)          self.esal=esal;
6)          self.eaddr=eaddr;
7)      def display(self):
8)
9)      print(self.eno,"\t",self.ename,"\t",self.esal,"\t",self.eaddr)
```

pick.py:

```
1)  import emp,pickle
2)  f=open("emp.dat","wb")
3)  n=int(input("Enter The number of Employees:"))
4)  for i in range(n):
5)      eno=int(input("Enter Employee Number:"))
6)      ename=input("Enter Employee Name:")
7)      esal=float(input("Enter Employee Salary:"))
8)      eaddr=input("Enter Employee Address:")
9)      e=emp.Employee(eno,ename,esal,eaddr)
10)     pickle.dump(e,f)
11) print("Employee Objects pickled successfully")
```

unpick.py:

```
1)  import emp,pickle
2)  f=open("emp.dat","rb")
3)  print("Employee Details:")
4)  while True:
5)      try:
6)          obj=pickle.load(f)
7)          obj.display()
8)      except EOFError:
9)          print("All employees Completed")
10)         break
11) f.close()
```

Object Serialization

The process of converting an object from python supported form to either file supported form or network supported form, is called serialization (Marshalling or pickling)

The process of converting an object from either file supported form or network supported form to python supported form is called deserialization (Unmarshalling or unpickling).

Object Serialization by using Pickle
Object Serialization by using JSON
Object Serialization by using YAML

Object Serialization by using Pickle:

We can perform serialization and deserialization of an object wrt file by using pickle module. It is Python's inbuilt module.

pickle module contains dump() function to perform Serialization(pickling).

```
pickle.dump(object,file)
```

pickle module contains load() function to perform Deserialization (unpickling).

```
object = pickle.load(file)
```

Program to perform pickling and unpickling of Employee Object:

```
1)  import pickle
2)  class Employee:
3)      def __init__(self,eno,ename,esal,eaddr):
4)          self.eno=eno
5)          self.ename=ename
6)          self.esal=esal
7)          self.eaddr=eaddr
8)      def display(self):
9)          print('ENO:{}, ENAME:{}, ESAL:{}, EADDR:{}'.format(self.eno,self.ename,self.es
    al,self.eaddr))
10)
11) e=Employee(100,'Durga',1000,'Hyderabad')
12) with open('emp.dat','wb') as f:
13)     pickle.dump(e,f)
14)     print('Pickling of Employee object completed')
15)
```

```
16) with open('emp.dat','rb') as f:
17)     obj = pickle.load(f)
18)     print('Unpickling of Employee object complected')
19)     print('Printing Employee Information:')
20)     obj.display()
```

Program for Serializing Multiple Employee objects to the file:

Based on our requirement, we can serialize any number of objects to the file.

emp.py

```
1)  class Employee:
2)      def __init__(self,eno,ename,esal,eaddr):
3)          self.eno=eno
4)          self.ename=ename
5)          self.esal=esal
6)          self.eaddr=eaddr
7)      def display(self):
8)          print('ENO:{}, ENAME:{}, ESAL:{}, EADDR:{}'.format(self.eno,self.ename,self.es
    al,self.eaddr))
```

sender.py

```
1)  #Sender is responsible to save Employee objects to the file
2)  from emp import *
3)  import pickle
4)  f=open('emp.dat','wb')
5)  while True:
6)      eno=int(input('Enter Employee Number:'))
7)      ename=input('Enter Employee Name:')
8)      esal=float(input('Enter Employee Salary:'))
9)      eaddr=input('Enter Employee Address:')
10)     e=Employee(eno,ename,esal,eaddr)
11)     pickle.dump(e,f)
12)     option=input('Do you want to serialize one more Employee object[Yes|No]:')
13)     if option.lower()=='no':
14)         break
15) print('All Employee objects serialized')
```

receiver.py

```
1)  #Receiver is responsible to deserialize Employee objects
2)  import pickle
3)  f=open('emp.dat','rb')
4)  print('Deserializing Employee objects and printing data...')
5)  while True:
6)      try:
7)          obj=pickle.load(f)
8)          obj.display()
9)      except EOFError:
10)         print('All Employees Completed')
11)         break
```

Object Serialization by using JSON

Importance of JSON:

JSON ➔ JavaScript Object Notation

Any programming language can understand json. Hence JSON is the most commonly used message format for applications irrespective of programming language and platform. It is very helpful for interoperability between applications.
It is human readable format.
It is light weight and required less memory to store data.

Eg:
Java Application sends request to Python application
Python application provide required response in json form.
Java application can understand json form and can be used based on its requirement.

What is JSON?

Python Data Types vs JavaScript Data Types

int → number
float → number
list → array
dict → object(JSON)
str → string
True → true
False → false

None → null
etc

In javascript if we want to represent a group of key value pairs, then we should go for object data type, which is nothing but json.
JSON is very similar to Python's dict object.

Why preference for JSON over XML:

1) Light weight
2) Human readable

Python's json module:

As the part of programming, it is very common requirement to convert python object into json form and from json form to python object. For these conversions (Serialization and Deserialization) Python provides inbuilt module json.

json module defines the following important functions:

For Serialization Purpose (From Python to JSON Form):

1) dumps() → It serializes python dict object to json string.
2) dump() → Converting python dict object to json and write that json to provided json file. It serializes to a file.

For Deserialization Purpose (From JSON form to Python form):

1) loads() → Converting JSON string to python dict. It deserializes to a string.
2) load() → Reading json from a file and converting to python dict object. Deserializes from a json file.

Demo program for Serialization

```
1)  import json
2)  employee={'name':'durga',
3)            'age':35,
4)           'salary':1000.0,
5)            'ismarried':True,
6)            'ishavinggirlfriend':None
7)           }
8)  json_string = json.dumps(employee,indent=4,sort_keys=True)
9)  print(json_string)
10)
11) with open('emp.json','w') as f:
```

```
12)    json.dump(employee,f,indent=4)
```

Demo Program for Deserialization from json String

```
1)  import json
2)  json_string='''{
3)     "name": "durga",
4)     "age": 35,
5)     "salary": 1000.0,
6)     "ismarried": true,
7)     "ishavinggirlfriend": null
8)  }'''
9)  emp_dict=json.loads(json_string)
10) print(type(emp_dict))
11) print('Employee Name:',emp_dict['name'])
12) print('Employee Age:',emp_dict['age'])
13) print('Employee Salary:',emp_dict['salary'])
14) print('Is Employee Married:',emp_dict['ismarried'])
15) print('Is Employee Has GF:',emp_dict['ishavinggirlfriend'])
16)
17) for k,v in emp_dict.items():
18)     print('{} : {}'.format(k,v))
```

Demo Program for Deserialization from json file

```
1)  import json
2)  with open('emp.json','r') as f:
3)     emp_dict=json.load(f)
4)
5)  print(type(emp_dict))
6)  print('Employee Name:',emp_dict['name'])
7)  print('Employee Age:',emp_dict['age'])
8)  print('Employee Salary:',emp_dict['salary'])
9)  print('Is Employee Married:',emp_dict['ismarried'])
10) print('Is Employee Has GF:',emp_dict['ishavinggirlfriend'])
11)
12) for k,v in emp_dict.items():
13)     print('{} : {}'.format(k,v))
```

FAQs from json module:

Q1. What is the difference between dump() and load() functions of json module?
Q2. What is the difference between dump() and dumps() functions of json module?
Q3. What is the difference between load() and loads() functions of json module?

Communicate with coindesk application to get bitcoin price:

If we send http request to coindesk application it will provide bitcoin current price information.

We can send http request from python application by using requests module. We have to install this module separately.

pip install requests

We can send request to coindesk application by using the following url:

https://api.coindesk.com/v1/bpi/currentprice.json

It will provide the following response in json format.

{
"time":{"updated":"Sep 18, 2013 17:27:00 UTC","updatedISO":"2013-09-18T17:27:00+00:00"},

"disclaimer":"This data was produced from the CoinDesk Bitcoin Price Index. Non-USD currency data converted using hourly conversion rate from openexchangerates.org",

"bpi":{"USD":{"code":"USD","symbol":"$","rate":"126.5235","description":"United States Dollar","rate_float":126.5235},

"GBP":{"code":"GBP","symbol":"£","rate":"79.2495","description":"British Pound Sterling","rate_float":79.2495},

"EUR":{"code":"EUR","symbol":"€","rate":"94.7398","description":"Euro","rate_float":94.7398}}}

Demo Program

```
1) import requests
2) response=requests.get('https://api.coindesk.com/v1/bpi/currentprice.json')
3) binfo=response.json() # provides python's dict object
4) #print(type(binfo))
```

```
#print(binfo)
print('Bitcoin Price as on',binfo['time']['updated'])
print('1 BitCoin = $',binfo['bpi']['USD']['rate'])
```

How to perform serialization and deserialization of customized class objects:

json.dumps() → python dict to json string
json.dump() → python dict to json file
dump(),dumps() functions will work only for python dict objects, and we cannot use for our customized class objects like Employee, Customer etc.

json.loads() → json string to python dict
json.load() → json file to python dict

load() and loads() functions will always provide python dict objects as return type and we won't get our customized class objects directly.

The required conversions we have to take care explicitly.

Demo Program

```
import json
class Employee:
    def __init__(self,eno,ename,esal,eaddr):
        self.eno=eno
        self.ename=ename
        self.esal=esal
        self.eaddr=eaddr
    def display(self):
        print('ENO:{}, ENAME:{}, ESAL:{}, EADDR:{}'.format(self.eno,self.ename,self.esal,self.eaddr))

e=Employee(100,'Durga',1000.0,'Hyderabad')

#emp_dict={'eno':e.eno,'ename':e.ename,'esal':e.esal,'eaddr':e.eaddr}

emp_dict=e.__dict__

with open('emp.json','w') as f:
    json.dump(emp_dict,f,indent=4)

with open('emp.json','r') as f:
```

```
21)    edict = json.load(f)
22) #print(type(edict))
23)
24) newE=Employee(edict['eno'],edict['ename'],edict['esal'],edict['eaddr'])
25)
26) newE.display()
```

Note: In the above program, we converted Employee object to dict object explicitly to perform serialization, because dump() function will always accept dict type only.

load() function always returns dict type only and hence we have to convert that dict object to Employee object explicitly.

Short-cut:

```
e=Employee(100,'Durga',1000.0,'Hyderabad')

with open('emp.json','w') as f:
        json.dump(e.__dict__,f,indent=4)
```

jsonpickle module:

By using jsonpickle module we can serialize our custom class objects directly to json and we can deserialize json to our custom class objects directly.

jsonpickle module is not available bydefault and we have to install explicitly.

```
pip install jsonpickle
```

This module contains

1. encode() → To convert any object to json_string directly
2. decode() → To convert json_string to our original object

Demo Program for serialization and deserialization by using jsonpickle

```
1)  import jsonpickle
2)  class Employee:
3)     def __init__(self,eno,ename,esal,eaddr,isMarried):
4)        self.eno=eno
5)        self.ename=ename
6)        self.esal=esal
7)        self.eaddr=eaddr
8)        self.isMarried=isMarried
```

```
9)     def display(self):
10)        print('ENO:{}, ENAME:{}, ESAL:{}, EADDR:{}, Is Married:{}'.format(self.eno,self.
    ename,self.esal,self.eaddr,self.isMarried))
11)
12) #Serialization to String
13) e=Employee(100,'Durga',1000.0,'Hyderabad',True)
14) json_string = jsonpickle.encode(e)
15) print(json_string)
16)
17) #Serialization to file
18) with open('emp.json','w') as f:
19)     f.write(json_string)
20)
21) #Deserialization
22) newEmp=jsonpickle.decode(json_string)
23) #print(type(newEmp))
24) newEmp.display()
25)
26) #Deserialization From the file
27) with open('emp.json','r') as f:
28)     json_string=f.readline()
29) newEmp=jsonpickle.decode(json_string)
30) newEmp.display()
```

Object Serialization with YAML

YAML: A retronym for YAML Ain't Markup Language that meant originally Yet Another Markup Language.

It is alternative to JSON.
It is also light weight and human understandable form.
It is more readable than JSON.

To serialize and deserialize our python data to yaml, we have to go for pyaml libray. This library by default not available and we have to install separately.

pip install pyaml

pyaml library contains yaml module.

yaml module contains the following functions to perform serialization and deserialization.

For Serialization

1.dump() → To serialize python dict object to yaml string or yaml file.

For Deserialization

2. load() → To deserialize from yaml string or yaml file to python dict object

Note: load() is deprecated and we have to use safe_load() function.

Demo Program for serialization and deserialization by using yaml

```
1)  from pyaml import yaml
2)  emp_dict={'name':'Durga','age':35,'salary':1000.0,'isMarried':True}
3)
4)  #Serialization to yaml string
5)  yaml_string=yaml.dump(emp_dict)
6)  print(yaml_string)
7)
8)  #Serialization to yaml file
9)  with open('emp.yaml','w') as f:
10)     yaml.dump(emp_dict,f)
11)
12) #Deserialization from yaml string
13) ed=yaml.safe_load(yaml_string)
14) print(ed)
15) for k,v in ed.items():
16)     print(k,':',v)
17)
18) #Deserialization from yaml file
19) with open('emp.yaml','r') as f:
20)     edf=yaml.safe_load(f)
21) print(edf)
```

Learn Complete Python In Simple Way

OOP's Part - 2

STUDY MATERIAL

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

- ❖ Method Resolution Order (MRO)
- ❖ super() Method

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
```

Demo Program-2:

```
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("Car Name:{} , Model:{} and Color:{}".format(self.name,self.model,self.c
   olor))
8)
9)  class Employee:
10)    def __init__(self,ename,eno,car):
11)        self.ename=ename
12)        self.eno=eno
13)        self.car=car
14)    def empinfo(self):
15)        print("Employee Name:",self.ename)
16)        print("Employee Number:",self.eno)
17)        print("Employee Car Info:")
18)        self.car.getinfo()
19) c=Car("Innova","2.5V","Grey")
20) e=Employee('Durga',10000,c)
21)         e.empinfo()
```

Output:

Employee Name: Durga
Employee Number: 10000
Employee Car Info:
Car Name: Innova, Model: 2.5V and Color:Grey

In the above program Employee class Has-A Car reference and hence Employee class can access all members of Car class.

Demo Program-3:

```
1)  class X:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      def m1(self):
6)              print("m1 method of X class")
7)
8)  class Y:
9)      c=30
10)     def __init__(self):
11)         self.d=40
12)     def m2(self):
13)         print("m2 method of Y class")
14)
15)     def m3(self):
16)         x1=X()
17)         print(x1.a)
18)         print(x1.b)
19)         x1.m1()
20)         print(Y.c)
21)         print(self.d)
22)         self.m2()
23)         print("m3 method of Y class")
24) y1=Y()
25) y1.m3()
```

Output:

```
10
20
m1 method of X class
30
40
```

m2 method of Y class
m3 method of Y class

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)

```
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
```

```
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.

```
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()
```

Output:

```
Parent class method
Child class method
```

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.

Similarly variables also

```
1)  class P:
2)     a=10
3)     def __init__(self):
4)        self.b=20
5)  class C(P):
6)     c=30
7)     def __init__(self):
8)        super().__init__()===>Line-1
9)        self.d=30
10)
```

```
11) c1=C()
12) print(c1.a,c1.b,c1.c,c1.d)
```

If we comment Line-1 then variable b is not available to the child class.

Demo program for Inheritance:

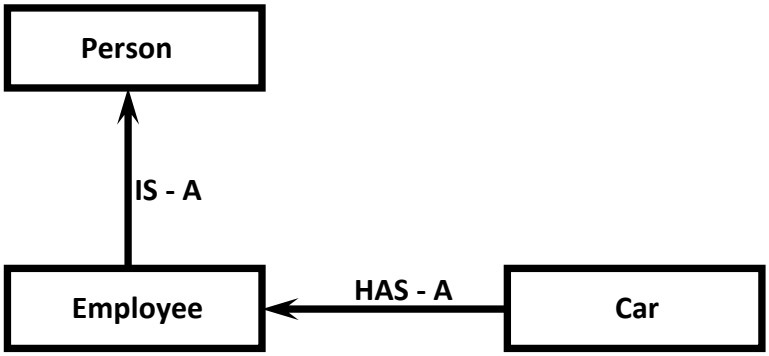
```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)      def  eatndrink(self):
6)          print('Eat Biryani and Drink Beer')
7)
8)  class Employee(Person):
9)      def __init__(self,name,age,eno,esal):
10)         super().__init__(name,age)
11)         self.eno=eno
12)         self.esal=esal
13)
14)     def work(self):
15)         print("Coding Python is very easy just like drinking Chilled Beer")
16)     def empinfo(self):
17)         print("Employee Name:",self.name)
18)         print("Employee Age:",self.age)
19)         print("Employee Number:",self.eno)
20)         print("Employee Salary:",self.esal)
21)
22) e=Employee('Durga', 48, 100, 10000)
23) e.eatndrink()
24) e.work()
25) e.empinfo()
```

Output:

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
```

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.color))
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 Beer')
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 Python is very easy just like drinking Chilled Beer")
24)     def empinfo(self):
25)         print("Employee Name:",self.name)
```

```
        print("Employee Age:",self.age)
        print("Employee Number:",self.eno)
        print("Employee Salary:",self.esal)
        print("Employee Car Info:")
        self.car.getinfo()

c=Car("Innova","2.5V","Grey")
e=Employee('Durga',48,100,10000,c)
e.eatndrink()
e.work()
e.empinfo()
```

Output:

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
Employee Car Info:
      Car Name:Innova
       Model:2.5V
       Color:Grey
```

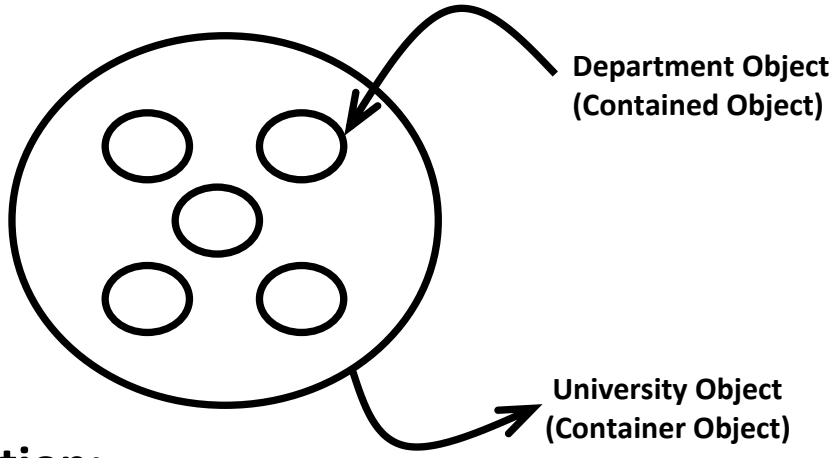
In the above example Employee class extends Person class functionality but just uses Car class functionality.

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.

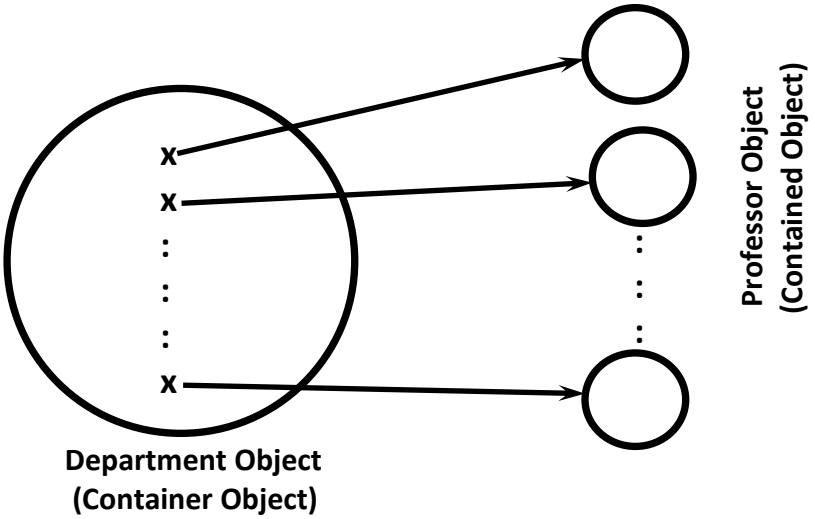
Eg: University contains several Departments and without existing university object there is no chance of existing Department object. Hence University and Department objects are strongly associated and this strong association is nothing but Composition.



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='DURGASOFT'
3)     def __init__(self,name):
4)         self.name=name
5) print(Student.collegeName)
6) s=Student('Durga')
```

```
7)  print(s.name)
```

Output:

DURGASOFT
Durga

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.

```
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

```
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,sel
      f.rollno,self.marks)
12) s1=Student('durga',48,101,90)
13) print(s1)
```

Output:

```
Name=durga
Age=48
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:

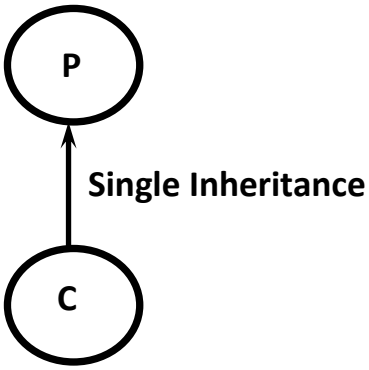
1) Single Inheritance:

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

```
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
```



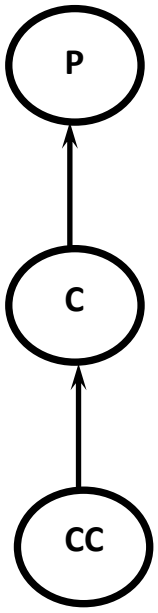
2) Multi Level Inheritance:

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

```
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:

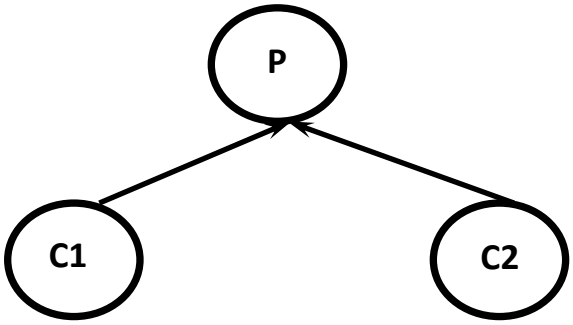
```
Parent Method
Child Method
Sub Child Method
```



Multi – Level Inheritance

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



Hierarchical Inheritance

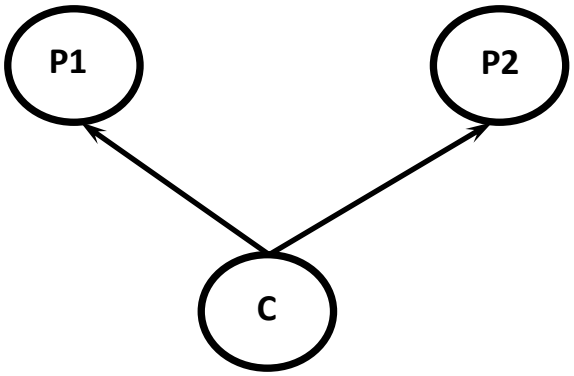
```
1)  class P:
2)      def m1(self):
3)          print("Parent Method")
4)  class C1(P):
5)      def m2(self):
6)          print("Child1 Method")
7)  class C2(P):
8)      def m3(self):
9)          print("Child2 Method")
10) c1=C1()
11) c1.m1()
12) c1.m2()
13) c2=C2()
14) c2.m1()
15) c2.m3()
```

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.



Multiple Inheritance

```
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

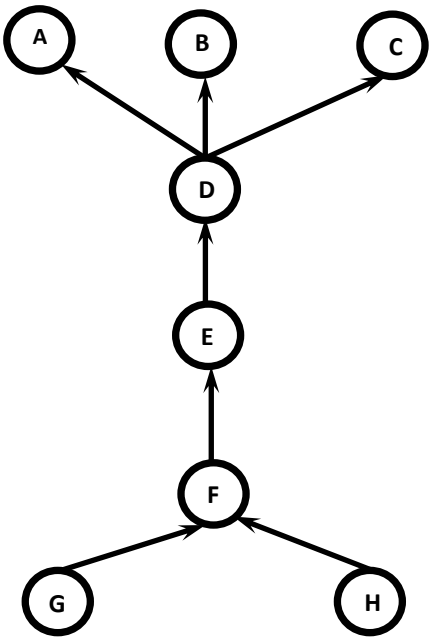
```
1)  class P1:
2)      def m1(self):
3)          print("Parent1 Method")
4)  class P2:
5)      def m1(self):
6)          print("Parent2 Method")
7)  class C(P1,P2):
8)      def m2(self):
9)          print("Child Method")
10) c=C()
11) c.m1()
12) c.m2()
```

Output:

Parent1 Method
Child Method

5) Hybrid Inheritance:

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

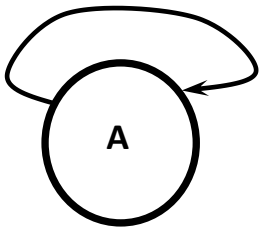


6) Cyclic Inheritance:

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

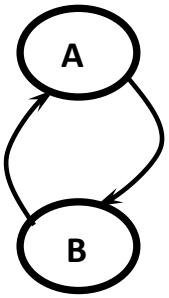
NameError: name 'A' is not defined



Eg - 2:

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

NameError: name 'B' is not defined



Method Resolution Order (MRO):

- In Hybrid Inheritance the method resolution order is decided based on MRO algorithm.
- This algorithm is also known as C3 algorithm.
- Samuele Pedroni proposed this algorithm.
- It follows DLR (Depth First Left to Right) i.e Child will get more priority than Parent.
- Left Parent will get more priority than Right Parent.
- MRO(X) = X+Merge(MRO(P1),MRO(P2),...,ParentList)

Head Element vs Tail Terminology:

- Assume C1,C2,C3,...are classes.
- In the list: C1C2C3C4C5....
- C1 is considered as Head Element and remaining is considered as Tail.

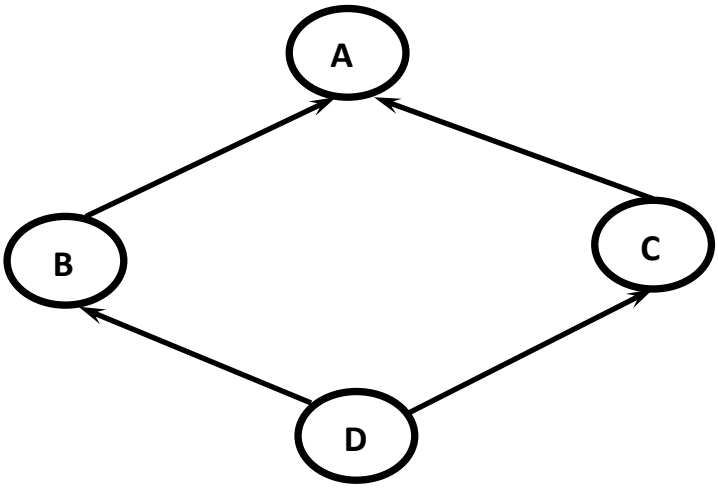
How to find Merge:

- Take the head of first list
- If the head is not in the tail part of any other list, then add this head to the result and remove it from the lists in the merge.
- If the head is present in the tail part of any other list, then consider the head element of the next list and continue the same process.

Note: We can find MRO of any class by using mro() function.

```
print(ClassName.mro())
```

Demo Program-1 for Method Resolution Order:



mro(A) = A, object
mro(B) = B, A, object
mro(C) = C, A, object
mro(D) = D, B, C, A, object

test.py

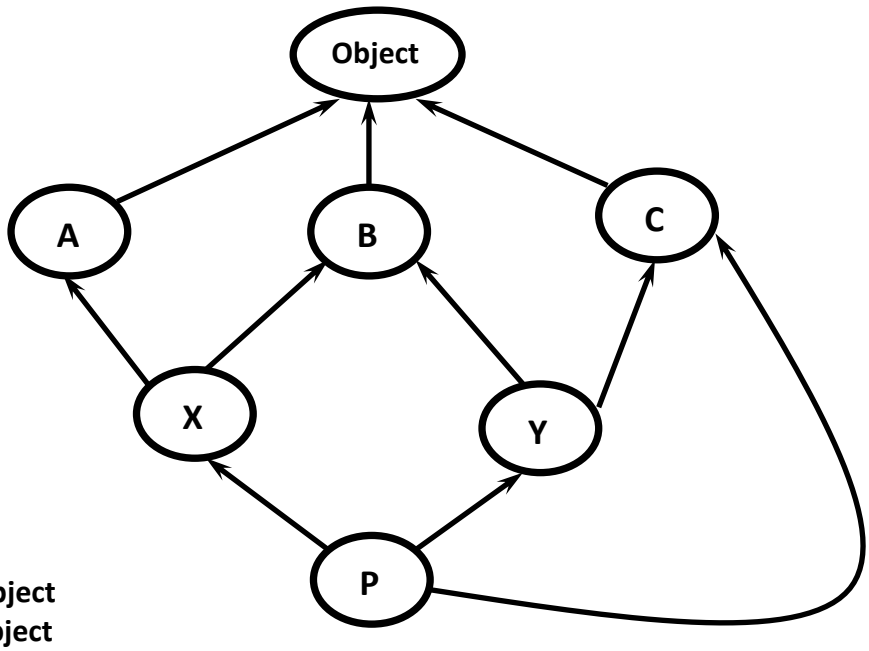
```
1) class A:pass
2) class B(A):pass
3) class C(A):pass
4) class D(B,C):pass
5) print(A.mro())
```

```
6)  print(B.mro())
7)  print(C.mro())
8)  print(D.mro())
```

Output:

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

Demo Program-2 for Method Resolution Order:



mro(A)=A,object
mro(B)=B,object
mro(C)=C,object
mro(X)=X,A,B,object
mro(Y)=Y,B,C,object
mro(P)=P,X,A,Y,B,C,object

Finding mro(P) by using C3 Algorithm:

Formula: MRO(X) = X+Merge(MRO(P1),MRO(P2),...,ParentList)

mro(p) = P+Merge(mro(X),mro(Y),mro(C),XYC)
= P+Merge(XABO,YBCO,CO,XYC)
= P+X+Merge(ABO,YBCO,CO,YC)
= P+X+A+Merge(BO,YBCO,CO,YC)

= P+X+A+Y+Merge(BO,BCO,CO,C)
= P+X+A+Y+B+Merge(O,CO,CO,C)
= P+X+A+Y+B+C+Merge(O,O,O)
= P+X+A+Y+B+C+O

test.py

```
1)  class A:pass
2)  class B:pass
3)  class C:pass
4)  class X(A,B):pass
5)  class Y(B,C):pass
6)  class P(X,Y,C):pass
7)  print(A.mro())#AO
8)  print(X.mro())#XABO
9)  print(Y.mro())#YBCO
10) print(P.mro())#PXAYBCO
```

Output:

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__main__.Y'>,
<class '__main__.B'>,
 <class '__main__.C'>, <class 'object'>]
```

test.py

```
1)  class A:
2)      def m1(self):
3)          print('A class Method')
4)  class B:
5)      def m1(self):
6)          print('B class Method')
7)  class C:
8)      def m1(self):
9)          print('C class Method')
10) class X(A,B):
11)     def m1(self):
12)         print('X class Method')
13) class Y(B,C):
14)     def m1(self):
15)         print('Y class Method')
```

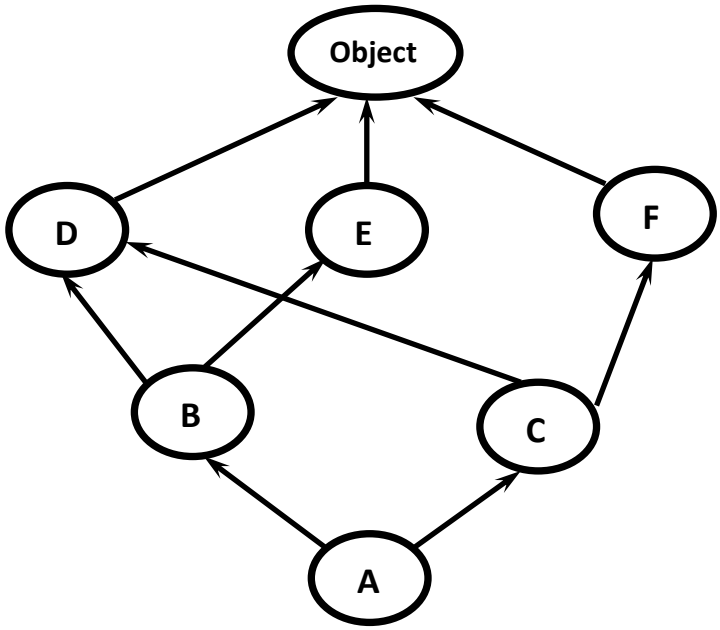
```
16) class P(X,Y,C):
17)     def m1(self):
18)         print('P class Method')
19) p=P()
20) p.m1()
```

Output: P class Method

In the above example P class m1() method will be considered.If P class does not contain m1() method then as per MRO, X class method will be considered. If X class does not contain then A class method will be considered and this process will be continued.

The method resolution in the following order: PXAYBCO

Demo Program-3 for Method Resolution Order:



mro(o) = object
mro(D) = D,object
mro(E) = E,object
mro(F) = F,object
mro(B) = B,D,E,object
mro(C) = C,D,F,object
mro(A) = A+Merge(mro(B),mro(C),BC)
= A+Merge(BDEO,CDFO,BC)
= A+B+Merge(DEO,CDFO,C)
= A+B+C+Merge(DEO,DFO)

= A+B+C+D+Merge(EO,FO)
= A+B+C+D+E+Merge(O,FO)
= A+B+C+D+E+F+Merge(O,O)
= A+B+C+D+E+F+O

test.py

```
1)  class D:pass
2)  class E:pass
3)  class F:pass
4)  class B(D,E):pass
5)  class C(D,F):pass
6)  class A(B,C):pass
7)  print(D.mro())
8)  print(B.mro())
9)  print(C.mro())
10) print(A.mro())
```

Output:

```
[<class '__main__.D'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.D'>, <class '__main__.E'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.D'>, <class '__main__.F'>, <class 'object'>]
[<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.D'>,
<class '__main__.E'>,
 <class '__main__.F'>, <class 'object'>]
```

super() Method:

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

Demo Program-1 for super():

```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)      def display(self):
6)          print('Name:',self.name)
7)          print('Age:',self.age)
8)
9)  class Student(Person):
10)     def __init__(self,name,age,rollno,marks):
11)         super().__init__(name,age)
```

```
12)       self.rollno=rollno
13)       self.marks=marks
14)
15)    def display(self):
16)       super().display()
17)       print('Roll No:',self.rollno)
18)       print('Marks:',self.marks)
19)
20) s1=Student('Durga',22,101,90)
21) s1.display()
```

Output:

Name: Durga
Age: 22
Roll No: 101
Marks: 90
In the above program we are using super() method to call parent class constructor and display() method

Demo Program-2 for super():

```
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)    a=888
16)    def __init__(self):
17)       self.b=999
18)       super().__init__()
19)       print(super().a)
20)       super().m1()
21)       super().m2()
22)       super().m3()
```

```
23)
24) c=C()
```

Output:

```
10
Parent instance method
Parent class method
Parent static method
```

In the above example we are using super() to call various members of Parent class.

How to Call Method of a Particular Super Class:

We can use the following approaches

1) super(D, self).m1()

It will call m1() method of super class of D.

2) A.m1(self)

It will call A class m1() method

```
1)      class A:
2)      def m1(self):
3)          print('A class Method')
4)  class B(A):
5)      def m1(self):
6)          print('B class Method')
7)  class C(B):
8)      def m1(self):
9)          print('C class Method')
10) class D(C):
11)     def m1(self):
12)         print('D class Method')
13) class E(D):
14)     def m1(self):
15)         A.m1(self)
16)
17) e=E()
18) e.m1()
```

Output: A class Method

Various Important Points about super():

Case-1: From child class we are not allowed to access parent class instance variables by using super(), Compulsory we should use self only.
But we can access parent class static variables by using super().

```
1)  class P:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)
6)  class C(P):
7)      def m1(self):
8)          print(super().a)#valid
9)          print(self.b)#valid
10)         print(super().b)#invalid
11) c=C()
12) c.m1()
```

Output:

```
10
20
AttributeError: 'super' object has no attribute 'b'
```

Case-2: From child class constructor and instance method, we can access parent class instance method, static method and class method by using super()

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)      def m1(self):
5)          print('Parent instance method')
6)      @classmethod
7)      def m2(cls):
8)          print('Parent class method')
9)      @staticmethod
10)     def m3():
11)         print('Parent static method')
12)
13) class C(P):
14)     def __init__(self):
15)         super().__init__()
16)         super().m1()
```

```
17)       super().m2()
18)      super().m3()
19)
20)    def m1(self):
21)       super().__init__()
22)       super().m1()
23)       super().m2()
24)       super().m3()
25)
26) c=C()
27) c.m1()
```

Output:

```
Parent Constructor
Parent instance method
Parent class method
Parent static method
Parent Constructor
Parent instance method
Parent class method
Parent static method
```

Case-3: From child class, class method we cannot access parent class instance methods and constructors by using super() directly(but indirectly possible). But we can access parent class static and class methods.

```
1)  class P:
2)     def __init__(self):
3)        print('Parent Constructor')
4)     def m1(self):
5)        print('Parent instance method')
6)     @classmethod
7)     def m2(cls):
8)        print('Parent class method')
9)     @staticmethod
10)    def m3():
11)       print('Parent static method')
12)
13) class C(P):
14)    @classmethod
15)    def m1(cls):
16)       #super().__init__()--->invalid
17)       #super().m1()--->invalid
18)       super().m2()
```

```
19)       super().m3()
20)
21) C.m1()
```

Output:

```
Parent class method
Parent static method
```

From Class Method of Child Class, how to call Parent Class Instance Methods and Constructors:

```
1)  class A:
2)      def __init__(self):
3)          print('Parent constructor')
4)
5)      def m1(self):
6)          print('Parent instance method')
7)
8)  class B(A):
9)      @classmethod
10)     def m2(cls):
11)         super(B,cls).__init__(cls)
12)         super(B,cls).m1(cls)
13)
14) B.m2()
```

Output:

```
Parent constructor
Parent instance method
```

Case-4: In child class static method we are not allowed to use super() generally (But in special way we can use)

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)      def m1(self):
5)          print('Parent instance method')
6)      @classmethod
7)      def m2(cls):
8)          print('Parent class method')
9)      @staticmethod
10)     def m3():
```

```
11)       print('Parent static method')
12)
13) class C(P):
14)     @staticmethod
15)     def m1():
16)         super().m1()-->invalid
17)         super().m2()--->invalid
18)         super().m3()--->invalid
19)
20) C.m1()
```

RuntimeError: super(): no arguments

How to Call Parent Class Static Method from Child Class Static Method by using super():

```
1)  class A:
2)
3)      @staticmethod
4)      def m1():
5)          print('Parent static method')
6)
7)  class B(A):
8)      @staticmethod
9)      def m2():
10)         super(B,B).m1()
11)
12) B.m2()
```

Output: Parent static method

Learn Complete Python In Simple Way

OOP's Part - 3

STUDY MATERIAL

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 4 topics are important

1) **Duck Typing Philosophy of Python**

2) **Overloading**
   1) **Operator Overloading**
   2) **Method Overloading**
   3) **Constructor Overloading**

3) **Overriding**
   1) **Method Overriding**
   2) **Constructor Overriding**

1) Duck Typing Philosophy of Python:

In Python we cannot specify the type explicitly. Based on provided value at runtime the type will be considered automatically. Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):
obj.talk()
```

What is the Type of obj? We cannot decide at the Beginning. At Runtime we can Pass any Type. Then how we can decide the Type?

At runtime if 'it walks like a duck and talks like a duck,it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

```
1)  class Duck:
2)      def talk(self):
3)          print('Quack.. Quack..')
4)
5)  class Dog:
6)      def talk(self):
7)          print('Bow Bow..')
8)
9)  class Cat:
10)     def talk(self):
11)         print('Moew Moew ..')
12)
13) class Goat:
14)     def talk(self):
15)         print('Myaah Myaah ..')
16)
17) def  f1(obj):
18)     obj.talk()
19)
20) l=[Duck(),Cat(),Dog(),Goat()]
21) for obj in l:
22)     f1(obj)
```

Output:

```
Quack.. Quack..
Moew Moew ..
Bow Bow..
Myaah Myaah ..
```

The problem in this approach is if obj does not contain talk() method then we will get AttributeError.

```
1)  class Duck:
2)      def talk(self):
3)          print('Quack.. Quack..')
4)
5)  class Dog:
6)      def bark(self):
```

```
        print('Bow Bow..')
def  f1(obj):
    obj.talk()

d=Duck()
f1(d)

d=Dog()
f1(d)
```

Output:

```
D:\durga_classes>py test.py
Quack.. Quack..
Traceback (most recent call last):
  File "test.py", line 22, in <module>
    f1(d)
  File "test.py", line 13, in f1
    obj.talk()
AttributeError: 'Dog' object has no attribute 'talk'
```

But we can solve this problem by using hasattr() function.

`hasattr(obj,'attributename')` ➔ attributename can be Method Name OR Variable Name

Demo Program with hasattr() Function:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Human:
    def talk(self):
        print('Hello Hi...')

class Dog:
    def bark(self):
        print('Bow Bow..')

def  f1(obj):
    if hasattr(obj,'talk'):
        obj.talk()
    elif hasattr(obj,'bark'):
        obj.bark()
```

```
18)
19) d=Duck()
20) f1(d)
21)
22) h=Human()
23) f1(h)
24)
25) d=Dog()
26) f1(d)
27) Myaah Myaah Myaah...
```

2) Overloading

We can use same operator or methods for different purposes.

Eg 1: + operator can be used for Arithmetic addition and String concatenation

```
print(10+20)#30
print('durga'+'soft')#durgasoft
```

Eg 2: * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200
print('durga'*3)#durgadurgadurga
```

Eg 3: 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.

Eg 1: + operator can be used for Arithmetic addition and String concatenation

```
print(10+20)#30
print('durga'+'soft')#durgasoft
```

Eg 2: * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200
print('durga'*3)#durgadurgadurga
```

Demo program to use + operator for our class objects:

```
1)  class Book:
2)      def __init__(self,pages):
3)          self.pages=pages
4)
5)  b1=Book(100)
6)  b2=Book(200)
7)  print(b1+b2)
```

```
D:\durga_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

- We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.
- For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.
- Internally + operator is implemented by using __add__() method.This method is called magic method for + operator. We have to override this method in our class.

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.

1) +	→	object.__add__(self,other)
2) -	→	object.__sub__(self,other)
3) *	→	object.__mul__(self,other)
4) /	→	object.__div__(self,other)
5) //	→	object.__floordiv__(self,other)
6) %	→	object.__mod__(self,other)
7) **	→	object.__pow__(self,other)
8) +=	→	object.__iadd__(self,other)
9) -=	→	object.__isub__(self,other)
10) *=	→	object.__imul__(self,other)
11) /=	→	object.__idiv__(self,other)
12) //=	→	object.__ifloordiv__(self,other)
13) %=	→	object.__imod__(self,other)
14) **=	→	object.__ipow__(self,other)
15) <	→	object.__lt__(self,other)
16) <=	→	object.__le__(self,other)
17) >	→	object.__gt__(self,other)
18) >=	→	object.__ge__(self,other)
19) ==	→	object.__eq__(self,other)
20) !=	→	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) print("10>20 =",10>20)
11) s1=Student("Durga",100)
12) s2=Student("Ravi",200)
13) print("s1>s2=",s1>s2)
14) print("s1<s2=",s1<s2)
15) print("s1<=s2=",s1<=s2)
16) print("s1>=s2=",s1>=s2)
```

Output

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

Program to Overload Multiplication Operator to Work on Employee Objects:

```
1)  class Employee:
2)      def __init__(self,name,salary):
3)          self.name=name
4)          self.salary=salary
5)      def __mul__(self,other):
6)          return self.salary*other.days
7)
8)  class TimeSheet:
9)      def __init__(self,name,days):
10)         self.name=name
11)         self.days=days
12)
13) e=Employee('Durga',500)
14) t=TimeSheet('Durga',25)
15) print('This Month Salary:',e*t)
```

Output: This Month Salary: 12500

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')
```

```
    def m1(self,a):
        print('one-arg method')
    def m1(self,a,b):
        print('two-arg method')

t=Test()
#t.m1()
#t.m1(10)
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:

```
class Test:
    def sum(self,a=None,b=None,c=None):
        if a!=None and b!= None and c!= None:
            print('The Sum of 3 Numbers:',a+b+c)
        elif a!=None and b!= None:
            print('The Sum of 2 Numbers:',a+b)
        else:
            print('Please provide 2 or 3 arguments')
t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
```

Output

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

Demo Program with Variable Number of Arguments:

```
class Test:
    def sum(self,*a):
        total=0
```

```
4)        for x in a:
5)            total=total+x
6)        print('The Sum:',total)
7)
8)  t=Test()
9)  t.sum(10,20)
10) t.sum(10,20,30)
11) t.sum(10)
12) t.sum()
```

3) Constructor Overloading:

- Constructor overloading is not possible in Python.
- If we define multiple constructors then the last constructor will be considered.

```
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.

Constructor with Default Arguments:

```
1)  class Test:
2)      def __init__(self,a=None,b=None,c=None):
3)          print('Constructor with 0|1|2|3 number of arguments')
4)
5)  t1=Test()
6)  t2=Test(10)
7)  t3=Test(10,20)
```

```
8) t4=Test(10,20,30)
```

Output

```
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
```

Constructor with Variable Number of Arguments:

```
1) class Test:
2)     def __init__(self,*a):
3)         print('Constructor with variable number of arguments')
4)
5) t1=Test()
6) t2=Test(10)
7) t3=Test(10,20)
8) t4=Test(10,20,30)
9) t5=Test(10,20,30,40,50,60)
```

Output:

```
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
```

3) Overriding

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('Appalamma')
6)  class C(P):
7)      def marry(self):
8)          print('Katrina Kaif')
9)
10) c=C()
11) c.property()
12)         c.marry()
```

Output

```
Gold+Land+Cash+Power
Katrina Kaif
```

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

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

Output

```
Gold+Land+Cash+Power
Appalamma
Katrina Kaif
```

Demo Program for Constructor Overriding:

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)
5)  class C(P):
6)      def __init__(self):
7)          print('Child Constructor')
8)
9)  c=C()
```

Output: Child Constructor

In the above example,if child class does not contain constructor then parent class constructor will be executed

From child class constuctor we can call parent class constructor by using super() method.

Demo Program to call Parent Class Constructor by using super():

```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)
6)  class Employee(Person):
7)      def __init__(self,name,age,eno,esal):
8)          super().__init__(name,age)
9)          self.eno=eno
10)         self.esal=esal
11)
12)     def display(self):
13)         print('Employee Name:',self.name)
14)         print('Employee Age:',self.age)
15)         print('Employee Number:',self.eno)
16)         print('Employee Salary:',self.esal)
17)
18) e1=Employee('Durga',48,872425,26000)
19) e1.display()
20) e2=Employee('Sunny',39,872426,36000)
21) e2.display()
```

Output

Employee Name: Durga
Employee Age: 48
Employee Number: 872425
Employee Salary: 26000

Employee Name: Sunny
Employee Age: 39
Employee Number: 872426
Employee Salary: 36000

Learn Complete Python In Simple Way

OOP's Part - 4

STUDY MATERIAL

Agenda

1) Abstract Method
2) Abstract class
3) Interface
4) Public,Private and Protected Members
5) __str__() Method
6) Difference between str() and repr() functions
7) Small Banking Application

Abstract Method:

- Sometimes we don't know about implementation, still we can declare a method. Such types of methods are called abstract methods.i.e abstract method has only declaration but not implementation.
- In python we can declare abstract method by using @abstractmethod decorator as follows.

- @abstractmethod
- def m1(self): pass

- @abstractmethod decorator present in abc module. Hence compulsory we should import abc module,otherwise we will get error.
- abc → abstract base class module

```
1) class Test:
2)     @abstractmethod
3)     def m1(self):
4)         pass
```

NameError: name 'abstractmethod' is not defined

Eg:

```
1) from abc import *
2) class Test:
3)     @abstractmethod
4)     def m1(self):
5)         pass
```

Eg:

```
1)  from abc import *
2)  class Fruit:
3)      @abstractmethod
4)      def taste(self):
5)          pass
```

Child classes are responsible to provide implemention for parent class abstract methods.

Abstract class:

Some times implementation of a class is not complete,such type of partially implementation classes are called abstract classes. Every abstract class in Python should be derived from ABC class which is present in abc module.

Case-1:

```
1)  from abc import *
2)  class Test:
3)      pass
4)
5)  t=Test()
```

In the above code we can create object for Test class b'z it is concrete class and it does not conatin any abstract method.

Case-2:

```
1)  from abc import *
2)  class Test(ABC):
3)      pass
4)
5)  t=Test()
```

In the above code we can create object, even it is derived from ABC class,b'z it does not contain any abstract method.

Case-3:

```
1)  from abc import *
2)  class Test(ABC):
3)      @abstractmethod
4)      def m1(self):
```

```
5)         pass
6)  t=Test()
```

TypeError: Can't instantiate abstract class Test with abstract methods m1

Case-4:

```
1)  from abc import *
2)  class Test:
3)      @abstractmethod
4)      def m1(self):
5)          pass
6)
7)  t=Test()
```

We can create object even class contains abstract method b'z we are not extending ABC class.

Case-5:

```
1)  from abc import *
2)  class Test:
3)      @abstractmethod
4)      def m1(self):
5)          print('Hello')
6)
7)  t=Test()
8)  t.m1()
```

Output: Hello

Conclusion: If a class contains atleast one abstract method and if we are extending ABC class then instantiation is not possible.

"abstract class with abstract method instantiation is not possible"

Parent class abstract methods should be implemented in the child classes. Otherwise we cannot instantiate child class.If we are not creating child class object then we won't get any error.

Case-1:

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle): pass
```

It is valid because we are not creating Child class object.

Case-2:

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle): pass
8)  b=Bus()
```

TypeError: Can't instantiate abstract class Bus with abstract methods noofwheels

Note: If we are extending abstract class and does not override its abstract method then child class is also abstract and instantiation is not possible.

```
1)  from abc import *
2)  class Vehicle(ABC):
3)      @abstractmethod
4)      def noofwheels(self):
5)          pass
6)
7)  class Bus(Vehicle):
8)      def noofwheels(self):
9)          return 7
10)
11) class Auto(Vehicle):
12)     def noofwheels(self):
13)         return 3
14) b=Bus()
15) print(b.noofwheels())#7
16)
```

```
17) a=Auto()
18) print(a.noofwheels())#3
```

Note: Abstract class can contain both abstract and non-abstract methods also.

Interfaces In Python:

In general if an abstract class contains only abstract methods such type of abstract class is considered as interface.

```
1)  from abc import *
2)  class DBInterface(ABC):
3)      @abstractmethod
4)      def connect(self):pass
5)
6)      @abstractmethod
7)      def disconnect(self):pass
8)
9)  class Oracle(DBInterface):
10)     def connect(self):
11)         print('Connecting to Oracle Database...')
12)     def disconnect(self):
13)         print('Disconnecting to Oracle Database...')
14)
15) class Sybase(DBInterface):
16)     def connect(self):
17)         print('Connecting to Sybase Database...')
18)     def disconnect(self):
19)         print('Disconnecting to Sybase Database...')
20)
21) dbname=input('Enter Database Name:')
22) classname=globals()[dbname]
23) x=classname()
24) x.connect()
25) x.disconnect()
```

```
D:\durga_classes>py test.py
Enter Database Name:Oracle
Connecting to Oracle Database...
Disconnecting to Oracle Database...

D:\durga_classes>py test.py
Enter Database Name:Sybase
```

Connecting to Sybase Database...
Disconnecting to Sybase Database...

Note: The inbuilt function globals()[str] converts the string 'str' into a class name and returns the classname.

Demo Program-2: Reading class name from the file

config.txt

EPSON

test.py

```
1)  from abc import *
2)  class Printer(ABC):
3)      @abstractmethod
4)      def printit(self,text):pass
5)
6)      @abstractmethod
7)      def disconnect(self):pass
8)
9)  class EPSON(Printer):
10)     def printit(self,text):
11)         print('Printing from EPSON Printer...')
12)         print(text)
13)     def disconnect(self):
14)         print('Printing completed on EPSON Printer...')
15)
16) class HP(Printer):
17)     def printit(self,text):
18)         print('Printing from HP Printer...')
19)         print(text)
20)     def disconnect(self):
21)         print('Printing completed on HP Printer...')
22)
23) with open('config.txt','r') as f:
24)     pname=f.readline()
25)
26) classname=globals()[pname]
27) x=classname()
28) x.printit('This data has to print...')
29) x.disconnect()
```

Output:

```
Printing from EPSON Printer...
This data has to print...
Printing completed on EPSON Printer...
```

Concreate class vs Abstract Class vs Inteface:

1) If we dont know anything about implementation just we have requirement specification then we should go for interface.
2) If we are talking about implementation but not completely then we should go for abstract class. (partially implemented class).
3) If we are talking about implementation completely and ready to provide service then we should go for concrete class.

```
1)  from abc import *
2)  class CollegeAutomation(ABC):
3)      @abstractmethod
4)      def m1(self): pass
5)      @abstractmethod
6)      def m2(self): pass
7)      @abstractmethod
8)      def m3(self): pass
9)  class AbsCls(CollegeAutomation):
10)     def m1(self):
11)         print('m1 method implementation')
12)     def m2(self):
13)         print('m2 method implementation')
14)
15) class ConcreteCls(AbsCls):
16)     def m3(self):
17)         print('m3 method implemnentation')
18)
19) c=ConcreteCls()
20) c.m1()
21) c.m2()
22) c.m3()
```

Public, Protected and Private Attributes:

By default every attribute is public. We can access from anywhere either within the class or from outside of the class.

Eg: name = 'durga'

Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefexing with _ symbol.

Syntax: _variablename = value

Eg: _name='durga'

But is is just convention and in reality does not exists protected attributes.

private attributes can be accessed only within the class.i.e from outside of the class we cannot access. We can declare a variable as private explicitly by prefexing with 2 underscore symbols.

syntax: __variablename=value

Eg: __name='durga'

```
1)  class Test:
2)      x=10
3)      _y=20
4)      __z=30
5)      def m1(self):
6)          print(Test.x)
7)          print(Test._y)
8)          print(Test.__z)
9)
10) t=Test()
11) t.m1()
12) print(Test.x)
13) print(Test._y)
14) print(Test.__z)
```

Output:

```
10
20
30
10
```

```
20
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    print(Test.__z)
AttributeError: type object 'Test' has no attribute '__z'
```

How to Access Private Variables from Outside of the Class:

We cannot access private variables directly from outside of the class.
But we can access indirectly as follows `objectreference._classname__variablename`

```
1)  class Test:
2)      def __init__(self):
3)          self.__x=10
4)
5)  t=Test()
6)  print(t._Test__x)#10
```

__str__() method:

- Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format
  `<__main__.classname object at 0x022144B0>`

- To return meaningful string representation we have to override __str__() method.

```
1)  class Student:
2)      def __init__(self,name,rollno):
3)          self.name=name
4)          self.rollno=rollno
5)
6)      def __str__(self):
7)          return 'This is Student with Name:{} and Rollno:{}'.format(self.name,self.rollno)
8)
9)  s1=Student('Durga',101)
10) s2=Student('Ravi',102)
11) print(s1)
12) print(s2)
```

Output without Overriding str():

```
<__main__.Student object at 0x022144B0>
<__main__.Student object at 0x022144D0>
```

Output with Overriding str():

```
This is Student with Name: Durga and Rollno: 101
This is Student with Name: Ravi and Rollno: 102
```

Difference between str() and repr()
OR
Difference between __str__() and __repr__()

- str() internally calls __str__() function and hence functionality of both is same.
- Similarly,repr() internally calls __repr__() function and hence functionality of both is same.
- str() returns a string containing a nicely printable representation object.
- The main purpose of str() is for readability.It may not possible to convert result string to original object.

```
1) import datetime
2) today=datetime.datetime.now()
3) s=str(today)#converting datetime object to str
4) print(s)
5) d=eval(s)#converting str object to datetime
```

```
D:\durgaclasses>py test.py
2018-05-18 22:48:19.890888
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    d=eval(s)#converting str object to datetime
  File "<string>", line 1
    2018-05-18 22:48:19.890888
          ^
SyntaxError: invalid token
```

But repr() returns a string containing a printable representation of object.
The main goal of repr() is unambigouous. We can convert result string to original object by using eval() function,which may not possible in str() function.

```
1) import datetime
2) today=datetime.datetime.now()
3) s=repr(today)#converting datetime object to str
4) print(s)
5) d=eval(s)#converting str object to datetime
6) print(d)
```

Output:

```
datetime.datetime(2018, 5, 18, 22, 51, 10, 875838)
2018-05-18 22:51:10.875838
```

Note: It is recommended to use repr() instead of str()

Mini Project: Banking Application

```
1)  class Account:
2)      def __init__(self,name,balance,min_balance):
3)          self.name=name
4)          self.balance=balance
5)          self.min_balance=min_balance
6)
7)      def deposit(self,amount):
8)          self.balance +=amount
9)
10)     def withdraw(self,amount):
11)         if self.balance-amount >= self.min_balance:
12)             self.balance -=amount
13)         else:
14)             print("Sorry, Insufficient Funds")
15)
16)     def printStatement(self):
17)         print("Account Balance:",self.balance)
18)
19) class Current(Account):
20)     def __init__(self,name,balance):
21)         super().__init__(name,balance,min_balance=-1000)
22)     def __str__(self):
23)         return "{}'s Current Account with Balance :{}".format(self.name,self.balance)
24)
25) class Savings(Account):
26)     def __init__(self,name,balance):
27)         super().__init__(name,balance,min_balance=0)
28)     def __str__(self):
29)         return "{}'s Savings Account with Balance :{}".format(self.name,self.balance)
30)
31) c=Savings("Durga",10000)
32) print(c)
33) c.deposit(5000)
34) c.printStatement()
35) c.withdraw(16000)
```

```
36) c.withdraw(15000)
37) print(c)
38)
39) c2=Current('Ravi',20000)
40) c2.deposit(6000)
41) print(c2)
42) c2.withdraw(27000)
43) print(c2)
```

Output:

```
D:\durgaclasses>py test.py
Durga's Savings Account with Balance :10000
Account Balance: 15000
Sorry, Insufficient Funds
Durga's Savings Account with Balance :0
Ravi's Current Account with Balance :26000
Ravi's Current Account with Balance :-1000
```

Python Logging Module

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Python Logging

It is highly recommended to store complete application flow and exceptions information to a file. This process is called logging.

The main advanatages of logging are:

1. We can use log files while performing debugging
2. We can provide statistics like number of requests per day etc

To implement logging, Python provides inbuilt module logging.

Logging Levels:

Depending on type of information, logging data is divided according to the following 6 levels in python

1. CRITICAL===>50
Represents a very serious problem that needs high attention

2. ERROR ===>40
Represents a serious error

3. WARNING ==>30
Represents a warning message, some caution needed. It is alert to the programmer.

4. INFO==>20
Represents a message with some important information

5. DEBUG ===>10
Represents a message with debugging information

6. NOTSET==>0
Represents that level is not set

By default while executing Python program only WARNING and higher level messages will be displayed.

How to implement Logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages required to store.

We can do this by using basicConfig() function of logging module.

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods

```
logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

Q. Write a Python Program to create a log file and write WARNING and Higher level messages?

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.WARNING)
3) print('Logging Demo')
4) logging.debug('Debug Information')
5) logging.info('info Information')
6) logging.warning('warning Information')
7) logging.error('error Information')
8) logging.critical('critical Information')
```

log.txt:

```
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

Note:

In the above program only WARNING and higher level messages will be written to the log file. If we set level as DEBUG then all messages will be written to the log file.

test.py:

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.DEBUG)
3) print('Logging Demo')
4) logging.debug('Debug Information')
5) logging.info('info Information')
6) logging.warning('warning Information')
```

```
7)  logging.error('error Information')
8)  logging.critical('critical Information')
```

log.txt:

```
DEBUG:root:Debug Information
INFO:root:info Information
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

How to configure log file in over writing mode:

In the above program by default data will be appended to the log file.i.e append is the default mode. Instead of appending if we want to over write data then we have to use filemode property.

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING)
```
meant for appending

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING,filemode='a')
```
explicitly we are specifying appending.

```
logging.basicConfig(filename='log786.txt',level=logging.WARNING,filemode='w')
```
meant for over writing of previous data.

Note:

```
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
```

If we are not specifying level then the default level is WARNING(30)

If we are not specifying file name then the messages will be printed to the console.

test.py:

```
1)  import logging
2)  logging.basicConfig()
3)  print('Logging Demo')
4)  logging.debug('Debug Information')
5)  logging.info('info Information')
6)  logging.warning('warning Information')
7)  logging.error('error Information')
8)  logging.critical('critical Information')
```

```
D:\durgaclasses>py test.py
Logging Demo
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

How to Format log messages:

By using format keyword argument, we can format messages.

1. To display only level name:

```
logging.basicConfig(format='%(levelname)s')
```

Output:
WARNING
ERROR
CRITICAL

2. To display levelname and message:

```
logging.basicConfig(format='%(levelname)s:%(message)s')
```

Output:
WARNING:warning Information
ERROR:error Information
CRITICAL:critical Information

How to add timestamp in the log messages:

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s')
```

Output:
2018-06-15 11:50:08,325:WARNING:warning Information
2018-06-15 11:50:08,372:ERROR:error Information
2018-06-15 11:50:08,372:CRITICAL:critical Information

How to change date and time format:

We have to use special keyword argument: datefmt

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s', datefmt='%d/%m/%Y
%I:%M:%S %p')
```

datefmt='%d/%m/%Y %I:%M:%S %p' ===>case is important

Output:
15/06/2018 12:04:31 PM:WARNING:warning Information
15/06/2018 12:04:31 PM:ERROR:error Information
15/06/2018 12:04:31 PM:CRITICAL:critical Information

Note:
%I--->means 12 Hours time scale
%H--->means 24 Hours time scale

Eg:
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s', datefmt='%d/%m/%Y %H:%M:%S')

Output:
15/06/2018 12:06:28:WARNING:warning Information
15/06/2018 12:06:28:ERROR:error Information
15/06/2018 12:06:28:CRITICAL:critical Information

https://docs.python.org/3/library/logging.html#logrecord-attributes

https://docs.python.org/3/library/time.html#time.strftime

How to write Python program exceptions to the log file:

By using the following function we can write exception information to the log file.

logging.exception(msg)

Q. Python Program to write exception information to the log file:

```
1)  import logging
2)  logging.basicConfig(filename='mylog.txt',level=logging.INFO,format='%(asctime)s:%(levelname)s:%(message)s',datefmt='%d/%m/%Y  %I:%M:%S %p')
3)  logging.info('A new Request Came')
4)  try:
5)      x=int(input('Enter First Number:'))
6)      y=int(input('Enter Second Number:'))
7)      print('The Result:',x/y)
8)
9)  except ZeroDivisionError as msg:
10)     print('cannot divide with zero')
11)     logging.exception(msg)
12)
13) except ValueError as msg:
14)     print('Please provide int values only')
15)     logging.exception(msg)
16)
17) logging.info('Request Processing Completed')
```

```
D:\durgaclasses>py test.py
Enter First Number:10
Enter Second Number:2
The Result: 5.0
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
D:\durgaclasses>py test.py
Enter First Number:20
Enter Second Number:2
The Result: 10.0

D:\durgaclasses>py test.py
Enter First Number:10
Enter Second Number:0
cannot divide with zero

D:\durgaclasses>py test.py
Enter First Number:ten
Please provide int values only
```

mylog.txt:

```
15/06/2018  12:30:51 PM:INFO:A new Request Came
15/06/2018  12:30:53 PM:INFO:Request Processing Completed
15/06/2018  12:30:55 PM:INFO:A new Request Came
15/06/2018  12:31:00 PM:INFO:Request Processing Completed
15/06/2018  12:31:02 PM:INFO:A new Request Came
15/06/2018  12:31:05 PM:ERROR:division by zero
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print('The Result:',x/y)
ZeroDivisionError: division by zero
15/06/2018  12:31:05 PM:INFO:Request Processing Completed
15/06/2018  12:31:06 PM:INFO:A new Request Came
15/06/2018  12:31:10 PM:ERROR:invalid literal for int() with base 10: 'ten'
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    x=int(input('Enter First Number:'))
ValueError: invalid literal for int() with base 10: 'ten'
15/06/2018  12:31:10 PM:INFO:Request Processing Completed
```

Problems with root logger:

If we are not defining our own logger,then bydefault root logger will be considered.
Once we perform basic configuration to root logger then the configurations are fixed and we cannot change.

Demo Application:

student.py:

```
1) import logging
2) logging.basicConfig(filename='student.log',level=logging.INFO)
3) logging.info('info message from student module')
```

test.py:

```
1) import logging
2) import student
3) logging.basicConfig(filename='test.log',level=logging.DEBUG)
4) logging.debug('debug message from test module')
```

student.log:

INFO:root:info message from student module

In the above application the configurations performed in test module won't be reflected,b'z root logger is already configured in student module.

Need of Our own customized logger:

The problems with root logger are:

1. Once we set basic configuration then that configuration is final and we cannot change

2. It will always work for only one handler at a time, either console or file, but not both simultaneously

3. It is not possible to configure logger with different configurations at different levels

4. We cannot specify multiple log files for multiple modules/classes/methods.

To overcome these problems we should go for our own customized loggers

Advanced logging Module Features: Logger:

Logger is more advanced than basic logging.
It is highly recommended to use and it provides several extra features.

Steps for Advanced Logging:

1. Creation of Logger object and set log level

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

2. Creation of Handler object and set log level
There are several types of Handlers like StreamHandler, FileHandler etc

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.INFO)
```

Note: If we use StreamHandler then log messages will be printed to console

3. Creation of Formatter object

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s: %(message)s',
datefmt='%d/%m/%Y %I:%M:%S %p')
```

4. Add Formatter to Handler

```
consoleHandler.setFormatter(formatter)
```

5. Add Handler to Logger

```
logger.addHandler(consoleHandler)
```

6. Write messages by using logger object and the following methods

```
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Note: Bydefault logger will set to WARNING level. But we can set our own level based on our requirement.

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

logger log level by default available to console and file handlers. If we are not satisfied with logger level, then we can set log level explicitly at console level and file levels.

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.WARNING)
```

```
fileHandler=logging.FileHandler('abc.log',mode='a')
fileHandler.setLevel(logging.ERROR)
```

Note:

console and file log levels should be supported by logger. i.e logger log level should be lower than console and file levels. Otherwise only logger log level will be considered.

Eg:

logger==>DEBUG console===>INFO -------->Valid and INFO will be considered
logger==>INFO console===>DEBUG -------->Invalid and only INFO will be considered to the console.

Demo Program for Console Handler

test.py:

```
1) import logging
2) logger=logging.getLogger('demologger')
3) logger.setLevel(logging.DEBUG)
4)
5) consoleHandler = logging.StreamHandler()
6)
7) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
8)
9) consoleHandler.setFormatter(formatter)
10) logger.addHandler(consoleHandler)
11)
12) logger.critical('It is critical message')
13) logger.error('It is error message')
14) logger.warning('It is warning message')
15) logger.info('It is info message')
16) logger.debug('It is debug message')
```

output

```
D:\durgaclasses>py test.py
27/07/2019  11:13:20 PM:CRITICAL:demologger:It is critical message
27/07/2019  11:13:20 PM:ERROR:demologger:It is error message
27/07/2019  11:13:20 PM:WARNING:demologger:It is warning message
27/07/2019  11:13:20 PM:INFO:demologger:It is info message
27/07/2019  11:13:20 PM:DEBUG:demologger:It is debug message
```

Demo Program for File Handler:

test.py

```
1) import logging
2) logger=logging.getLogger('demologger')
3) logger.setLevel(logging.DEBUG)
4)
5) fileHandler=logging.FileHandler('custtest.log',mode='w')
6)
7) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
8)
9) fileHandler.setFormatter(formatter)
```

```
10) logger.addHandler(fileHandler)
11)
12) logger.critical('It is critical message')
13) logger.error('It is error message')
14) logger.warning('It is warning message')
15) logger.info('It is info message')
16) logger.debug('It is debug message')
```

custtest.log

```
27/07/2019  11:16:39 PM:CRITICAL:demologger:It is critical message
27/07/2019  11:16:39 PM:ERROR:demologger:It is error message
27/07/2019  11:16:39 PM:WARNING:demologger:It is warning message
27/07/2019  11:16:39 PM:INFO:demologger:It is info message
27/07/2019  11:16:39 PM:DEBUG:demologger:It is debug message
```

Demo Program to use both Console and File Handlers:

test.py

```
1)  import logging
2)  logger=logging.getLogger('demo logger')
3)  logger.setLevel(logging.INFO)
4)
5)  consoleHandler = logging.StreamHandler()
6)  fileHandler=logging.FileHandler('abc.log',mode='w')
7)
8)  formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
    datefmt='%d/%m/%Y  %I:%M:%S %p')
9)
10) consoleHandler.setFormatter(formatter)
11) fileHandler.setFormatter(formatter)
12)
13) logger.addHandler(consoleHandler)
14) logger.addHandler(fileHandler)
15)
16) logger.critical('It is critical message')
17) logger.error('It is error message')
18) logger.warning('It is warning message')
19) logger.info('It is info message')
20) logger.debug('It is debug message')
```

output on console

```
D:\durgaclasses>py test.py
27/07/2019  11:19:41 PM:CRITICAL:demo logger:It is critical message
```

27/07/2019 11:19:41 PM:ERROR:demo logger:It is error message
27/07/2019 11:19:41 PM:WARNING:demo logger:It is warning message
27/07/2019 11:19:41 PM:INFO:demo logger:It is info message

abc.log

27/07/2019 11:19:41 PM:CRITICAL:demo logger:It is critical message
27/07/2019 11:19:41 PM:ERROR:demo logger:It is error message
27/07/2019 11:19:41 PM:WARNING:demo logger:It is warning message
27/07/2019 11:19:41 PM:INFO:demo logger:It is info message

Demo program to define and use custom logger with different modules and with different log files:

test.py

```
1) import logging
2) import student
3) logger=logging.getLogger('testlogger')
4) logger.setLevel(logging.DEBUG)
5)
6) fileHandler=logging.FileHandler('test.log',mode='a')
7)
8) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %I:%M:%S %p')
9)
10) fileHandler.setFormatter(formatter)
11) logger.addHandler(fileHandler)
12)
13) logger.critical('critical message from test module')
14) logger.error('error message from test module')
15) logger.warning('warning message from test module')
16) logger.info('info message from test module')
17) logger.debug('debug message from test module')
```

student.py

```
1) import logging
2) logger=logging.getLogger('studentlogger')
3) logger.setLevel(logging.DEBUG)
4)
5) fileHandler=logging.FileHandler('student.log',mode='a')
6) fileHandler.setLevel(logging.ERROR)
7)
```

```
8) formatter=logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s',
   datefmt='%d/%m/%Y  %H:%M:%S')
9)
10) fileHandler.setFormatter(formatter)
11) logger.addHandler(fileHandler)
12)
13) logger.critical('critical message from student module')
14) logger.error('error message from student module')
15) logger.warning('warning message student test module')
16) logger.info('info message from student module')
17) logger.debug('debug message from student module')
```

test.log

```
24/07/2019 01:24:09 PM:demologger:CRITICAL:It is critical message
24/07/2019 01:24:09 PM:demologger:ERROR:It is error message
27/07/2019  11:27:06 PM:CRITICAL:testlogger:critical message from test module
27/07/2019  11:27:06 PM:ERROR:testlogger:error message from test module
27/07/2019  11:27:06 PM:WARNING:testlogger:warning message from test module
27/07/2019  11:27:06 PM:INFO:testlogger:info message from test module
27/07/2019  11:27:06 PM:DEBUG:testlogger:debug message from test module
```

student.log

```
2019-07-22 13:52:21,343:CRITICAL:studentlogger:critical message from student module
2019-07-22 13:52:21,343:ERROR:studentlogger:error message from student module
27/07/2019  23:26:35:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:26:35:ERROR:studentlogger:error message from student module
27/07/2019  23:26:59:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:26:59:ERROR:studentlogger:error message from student module
27/07/2019  23:27:06:CRITICAL:studentlogger:critical message from student module
27/07/2019  23:27:06:ERROR:studentlogger:error message from student module
```

Note: In the above program we are maintaining different log files for different modules, which is not possible by root logger.

Creation of generic custom logger and usage Demo Program-1:

custlogger.py

```
1) import logging
2) import inspect
3) def get_custom_logger(level):
4)     function_name =inspect.stack()[1][3]
5)     logger_name=function_name+" logger"
6)
```

```
7)    logger=logging.getLogger(logger_name)
8)    logger.setLevel(level)
9)
10)   fileHandler = logging.FileHandler('abc.log',mode='a')
11)   fileHandler.setLevel(level)
12)   formatter=logging.Formatter(
13)       '%(asctime)s:%(levelname)s:%(name)s:%(message)s',
14)        datefmt='%d/%m/%Y  %I:%M:%S %p')
15)   fileHandler.setFormatter(formatter)
16)   logger.addHandler(fileHandler)
17)   return logger
```

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def logtest():
4)     logger=get_custom_logger(logging.DEBUG)
5)     logger.critical('critical message from test module')
6)     logger.error('error message from test module')
7)     logger.warning('warning message from test module')
8)     logger.info('info message from test module')
9)     logger.debug('debug message from test module')
10) logtest()
```

student.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def logstudent():
4)     logger=get_custom_logger(logging.ERROR)
5)     logger.critical('critical message from student module')
6)     logger.error('error message from student module')
7)     logger.warning('warning message from student module')
8)     logger.info('info message from student module')
9)     logger.debug('debug message from student module')
10) logstudent()
```

abc.log

```
27/07/2019  11:35:53 PM:CRITICAL:logtest logger:critical message from test module
27/07/2019  11:35:53 PM:ERROR:logtest logger:error message from test module
27/07/2019  11:35:53 PM:WARNING:logtest logger:warning message from test module
27/07/2019  11:35:53 PM:INFO:logtest logger:info message from test module
27/07/2019  11:35:53 PM:DEBUG:logtest logger:debug message from test module
```

27/07/2019 11:35:54 PM:CRITICAL:logstudent logger:critical message from student module
27/07/2019 11:35:54 PM:ERROR:logstudent logger:error message from student module

Creation of generic custom logger and usage Demo Program-2:

custlogger.py

```
1)  import logging
2)  import inspect
3)  def get_custom_logger(level):
4)      function_name =inspect.stack()[1][3]
5)      logger_name=function_name+" logger"
6)
7)      logger=logging.getLogger(logger_name)
8)      logger.setLevel(level)
9)
10)     fileHandler = logging.FileHandler('abc.log',mode='a')
11)     fileHandler.setLevel(level)
12)     formatter=logging.Formatter(
13)         '%(asctime)s:%(levelname)s:%(name)s:%(message)s',
14)          datefmt='%d/%m/%Y  %I:%M:%S %p')
15)     fileHandler.setFormatter(formatter)
16)     logger.addHandler(fileHandler)
17)     return logger
```

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def f1():
4)      logger=get_custom_logger(logging.DEBUG)
5)      logger.critical('critical message from f1')
6)      logger.error('error message from f1')
7)      logger.warning('warning message from f1')
8)      logger.info('info message from f1')
9)      logger.debug('debug message from f1')
10) def f2():
11)     logger=get_custom_logger(logging.WARNING)
12)     logger.critical('critical message from f2')
13)     logger.error('error message from f2')
14)     logger.warning('warning message from f2')
15)     logger.info('info message from f2')
16)     logger.debug('debug message from f2')
```

```
17) def f3():
18)     logger=get_custom_logger(logging.ERROR)
19)     logger.critical('critical message from f3')
20)     logger.error('error message from f3')
21)     logger.warning('warning message from f3')
22)     logger.info('info message from f3')
23)     logger.debug('debug message from f3')
24) f1()
25) f2()
26) f3()
```

abc.log

```
27/07/2019  11:38:56 PM:CRITICAL:f1 logger:critical message from f1
27/07/2019  11:38:56 PM:ERROR:f1 logger:error message from f1
27/07/2019  11:38:56 PM:WARNING:f1 logger:warning message from f1
27/07/2019  11:38:56 PM:INFO:f1 logger:info message from f1
27/07/2019  11:38:56 PM:DEBUG:f1 logger:debug message from f1
27/07/2019  11:38:56 PM:CRITICAL:f2 logger:critical message from f2
27/07/2019  11:38:56 PM:ERROR:f2 logger:error message from f2
27/07/2019  11:38:56 PM:WARNING:f2 logger:warning message from f2
27/07/2019  11:38:56 PM:CRITICAL:f3 logger:critical message from f3
27/07/2019  11:38:56 PM:ERROR:f3 logger:error message from f3
```

How to create separate log file based on caller dynamically?

custlogger.py

```
1)  import logging
2)  import inspect
3)  def get_custom_logger(level):
4)      function_name =inspect.stack()[1][3]
5)      logger_name=function_name+" logger"
6)
7)      logger=logging.getLogger(logger_name)
8)      logger.setLevel(level)
9)
10)     fileHandler = logging.FileHandler('{}.log'.format(function_name),mode='a')
11)     fileHandler.setLevel(level)
12)     formatter=logging.Formatter(
13)         '%(asctime)s:%(levelname)s:%(name)s:%(message)s',
14)          datefmt='%d/%m/%Y  %I:%M:%S %p')
15)     fileHandler.setFormatter(formatter)
16)     logger.addHandler(fileHandler)
17)     return logger
```

test.py

```
1)  from custlogger import get_custom_logger
2)  import logging
3)  def f1():
4)      logger=get_custom_logger(logging.DEBUG)
5)      logger.critical('critical message from f1')
6)      logger.error('error message from f1')
7)      logger.warning('warning message from f1')
8)      logger.info('info message from f1')
9)      logger.debug('debug message from f1')
10) def f2():
11)     logger=get_custom_logger(logging.WARNING)
12)     logger.critical('critical message from f2')
13)     logger.error('error message from f2')
14)     logger.warning('warning message from f2')
15)     logger.info('info message from f2')
16)     logger.debug('debug message from f2')
17) def f3():
18)     logger=get_custom_logger(logging.ERROR)
19)     logger.critical('critical message from f3')
20)     logger.error('error message from f3')
21)     logger.warning('warning message from f3')
22)     logger.info('info message from f3')
23)     logger.debug('debug message from f3')
24) f1()
25) f2()
26) f3()
```

f1.log

```
27/07/2019  11:41:33 PM:CRITICAL:f1 logger:critical message from f1
27/07/2019  11:41:33 PM:ERROR:f1 logger:error message from f1
27/07/2019  11:41:33 PM:WARNING:f1 logger:warning message from f1
27/07/2019  11:41:33 PM:INFO:f1 logger:info message from f1
27/07/2019  11:41:33 PM:DEBUG:f1 logger:debug message from f1
```

f2.log

```
27/07/2019  11:41:33 PM:CRITICAL:f2 logger:critical message from f2
27/07/2019  11:41:33 PM:ERROR:f2 logger:error message from f2
27/07/2019  11:41:33 PM:WARNING:f2 logger:warning message from f2
```

f3.log

```
27/07/2019  11:41:33 PM:CRITICAL:f3 logger:critical message from f3
27/07/2019  11:41:33 PM:ERROR:f3 logger:error message from f3
```

Need of separating logger configurations into a file or dict or JSON or YAML?

Instead of hard coding logging configurations inside our application, we can separate into into a file or dict or JSON or YAML.

Advantages:

1. Modifications will become very easy.
2. We can reuse same configurations in different modules.
3. Length of the code will be reduced and readability will be improved.

Demo Program for Logger configurations into a separate config file

logging_config.init: For File Handler

```
1) [loggers]
2) keys=root,demologger
3)
4) [handlers]
5) keys=fileHandler
6)
7) [formatters]
8) keys=sampleFormatter
9)
10) [logger_root]
11) level=DEBUG
12) handlers=fileHandler
13)
14) [logger_demologger]
15) level=DEBUG
16) handlers=fileHandler
17) qualname=demoLogger
18)
19) [handler_fileHandler]
20) class=FileHandler
21) level=DEBUG
22) formatter=sampleFormatter
23) args=('test.log','w')
24)
25) [formatter_sampleFormatter]
26) format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
27) datefmt=%d/%m/%Y %I:%M:%S %p
```

logging_config.init: For Console Handler

```
1) [loggers]
2) keys=root,demologger
3)
4) [handlers]
5) keys=consoleHandler
6)
7) [formatters]
8) keys=sampleFormatter
9)
10) [logger_root]
11) level=DEBUG
12) handlers=consoleHandler
13)
14) [logger_demologger]
15) level=DEBUG
16) handlers=consoleHandler
17) qualname=demoLogger
18)
19)
20) [handler_consoleHandler]
21) class=StreamHandler
22) level=DEBUG
23) formatter=sampleFormatter
24) args=(sys.stdout,)
25)
26) [formatter_sampleFormatter]
27) format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
28) datefmt=%d/%m/%Y %I:%M:%S %p
```

test.py

```
1) import logging
2) import logging.config
3) logging.config.fileConfig("logging_config.init")
4) logger=logging.getLogger('demologger')
5) logger.critical('It is critical message')
6) logger.error('It is error message')
7) logger.warning('It is warning message')
8) logger.info('It is info message')
9) logger.debug('It is debug message')
```

Logger configurations into a dictionary

test.py

```
1)  import logging
2)  from logging.config import dictConfig
3)
4)  logging_config = dict(
5)     version = 1,
6)     formatters = {
7)        'f': {'format':
8)             '%(asctime)s:%(name)s:%(levelname)s:%(message)s',
9)             'datefmt':'%d/%m/%Y %I:%M:%S %p'}
10)       },
11)    handlers = {
12)       'h': {'class': 'logging.StreamHandler',
13)            'formatter': 'f',
14)            'level': logging.DEBUG}
15)       },
16)    root = {
17)       'handlers': ['h'],
18)       'level': logging.DEBUG,
19)       },
20) )
21)
22) dictConfig(logging_config)
23)
24) logger = logging.getLogger()
25) logger.critical('it is critical message')
26) logger.error('it is error message')
27) logger.warning('it is warning message')
28) logger.info('it is info message')
29) logger.debug('it is debug message')
```