Functions

* A function is a block of statements.
* functions provide better modularity for user application and using high degree of code.
* It can be classified into two types.

1. predefined function (or) built-in functions

* These functions contain definition in compiler itself.

Ex:- type ( )
id ( )
print ( )

2. user defined function:-

* user can create its own function.

Syn:- def functionname (parameter):
block of statements
return [expression]

* function blocks begin with the keyword def followed by functionname and ( ).

* In paranthesis it contain Parameter (or) arguments.
* the code block in every function starts with : and is indented.
* The statement return exist in a function, it returns, Expression.
* A return statement with no argument it returns None.

Ex:-

1. def wish(name):

    Print("welcome to functions topic", name, "learners")

wish("Deepu")

wish("Ammu")

output:-

welcome to functions topic Deepu 'learners'

welcome to functions topic Ammu 'learners'

2. def My_functions():

    Print("Hello welcome to functions")

my_function()

output:- Hello welcome to functions.

calling function :-

once the basic structure of a function is finalized, we can execute it by calling from another function directly.

Ex:-

1. def printme(str):

    print(str)

    return;

printme("I'm first call to user define function!")

Printme("Again second call to the same function")

output:-

I'm first call to user define function.

Again Second call to the same function.

2. def foodname (str):

    Print(str)

foodname ("Tiffan is : chapathi")

foodname ("Lunch is : bissi bele bath")

foodname (" Dinner is : curd Rice")

output:-

Tiffan is : chapathi

Lunch is : bissi bele bath.

Dinner is : curd rice.

Pass by refeqence vs value:-

we can change what a parameteq "refeqce" to with in a function. The changes can also reflex "back" in the calling function.

Ex:-

1. def changeme (mylist):

    mylist.append ([1, 2, 3, 4]);

    Print ("values inside the function :", mylist)

    retqn.

mylist = [10, 20, 30];

changeme (mylist);

print ("values outside the function :", mylist)

output:-

Values inside the function : [10, 20, 30, [1, 2, 3, 4]]

Values outside the function : [10, 20, 30, [1, 2, 3, 4]]

Ex:-

2. def changeme (mylist):

    mylist = [1, 2, 3, 4];

```
    Print ("values inside the function : ", mylist)
    return.

mylist = [10, 20, 30];
changeme (mylist)
Print ("values outside the function:", mylist)
```

Output:-

values inside the function : [1, 2, 3, 4]

values outside the function : [10, 20, 30]

Function arguments:-

It can be classified into 4 categories-

* Required arguments
* keyword arguments.
* Default arguments.
* variable-length arguments.

1. Required arguments:-

The number of arguments in the function call should match exactly with the function definition.

Ex:-

```
1. def printme (str):
       print (str)
       return;

printme (str)
```

output:-

<class 'str'>

2. keyword arguments:-

keyword arguments are related to the function calls. when we use keyword arguments in function call, the

called identifies the arguments by the parameter name.

Ex:-

1. def printme (str):

    Print (str)

    return ;

Print me (str = "My string is Python hello")

output:-

My string is Python hello.

2. def printinfo (name , age):

    Print ("Name :", name)

    Print ("Age ", age)

    return;

Print info (age=50, name = "venkat")

output:-

Name: venkat

Age: 50.

3. default arguments:-

default argument is an argument that assumes default values, if a value is not provided in the function call for the argument automatically it take default value.

Ex:-

1. def printinfo (name = "ram", age=25):

    Print ("Name:" , name)

    Print ("Age ", age)

    return;

```
Print info (age = 20, name = "Deepu")
Print info (name = "Milky")
Print info()
```

Output:-

Name: Deepu

age: 20

Name : Milky

age: 25

Name: Ram

age : 25.

4. Variable - length arguments:-

To suply more number of arguments to the function defination.

Syn:- def . functionname (args , * var args tuple):

"functionblock"

retun [Expression]

* An asteqisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

* These tuple remains empty if no additional arguments are specified during the function call.

Ex:-

```
1. def  printinfo (arg1 , * vartuple ):
        Print ("output is:")
        Print (arg1)
        for var in vartuple:
            Print (var)
        retun;
```

print info (10)

print info (70, 40, 90)

output:-

output is :

10

output is:

70

40

90.

Anonymous Function:- (lambda)

* These function are called anonymous because they are not declared in the standard manner by using the def keyword.

* You can use the lambda keyword to create small anonymous function.

* lambda forms can take any number of arguments but return only one value in the form of expression. They cannot contain commands (or) multiple expression.

* A Anonymous function cannot be direct call to print because lambda required an expression.

* lambda functions have their own local name space and cannot access variable other than those in their parameter list and those in the global namespace.

syn:- lambda [arg1, arg2 .... argn]: expression.

Ex:-

1. sum = lambda arg1, arg2 : arg1 + arg2 ;

print ("value of total : ", sum (10, 20))

print ("value of total :", sum (20, 20))

output:-

value of total : 30

value of total : 40

return:-

whenever function call occur the function defination contain return statement. these can return expression.

Ex:-

```
1. def Sum (arg1, arg2 ):
        total = arg1+arg2
        Print ("Inside the function :", total)
        return total;

total =Sum (100, 20);
Print ("outside the function :", total)
```

OutPut:-

inside the function : 120

outside the function: 120.

Scope of variable:-

variable that are defined inside of the function body have a local scope and those define outside have a global scope.

Ex:-

```
1. total=0
def Sum (arg1, arg2 ):
        total = arg1 +arg2;
        Print ("Inside the function local total : ", total)
        return total;
Sum (10,20);
```

```
print ("outside the function global total : ", total
```

output:-

inside the function local total: 30

outside the function global total: 0

lambda Examples:-

1.
```
x = lambda a: a+10
print (x(5))
```

output:-

15.

2.
```
x= lambda a,b : a*b
Print (x (5,6))
```

output:-

30.

3.
```
x = lambda a,b,c :a +b+c
print (x(5 ,6,2))
```

output:-

13.

4.
```
def myfunc (n):
    return lambda a: a*n

mydoubleq = my func (2)
print (mydoubleq(11))
```

output:-

22.

Modules

What is Module?

* A Module to be the same as a code of library.
* A file containing set of functions you want to include in your application.

How to create Module:

To create a Module just save the code you want in a file with the file extension .py.

Save the code in the file name "mywon.py"

Ex:-

```
1. def greeting(Deepu):
       Print ("hello ," +Deepu)
```

use as a Module:

Now we can use the Module, just we created module by using the "import" statement.

* Module name . calling function.

Ex:- Myown . greeting (" welcome to my world")

Ex:-

```
1. import Myown
   Myown . greeting (" welcome to my world")
```

output:-

'Hello', welcome to my world.

Note:-

When using a function from module use the

Syn:- Modulename.functionname

Ex:- myown.add(10,20)

Variables in Module :-

The Module can contain functions, as already discribed (applications), but also variables of all types (arrays, dict, object ... etc).

Example- Create (Person) : Module - myown1

```
Person ={
        "name" : "Deepu",
        "country" : "India",
        "game" : "kabaddi",
        "age" : 21,
        "gender": "female"
    }
```

* Import the module name (myown1) and access the person dict.

Ex:-

1. &

```
import myown1
a= myown1.person["name"]
print ("name :",a)
b= myown1.person["country"]
Print ("country:", b)
c= myown1.person["game"]
Print ("game :", c)
d= myown1.person["age"]
```

```
print ("age :", d)
e = myown1.Person["gender"]
print ("gender:", e)
```

output :-

name : Deepu

country : India

game : kabadd:

age : 21

gender : female.

Re-naming a module :

We can create an alias name for module by using the "as" keyword.

Create an alias name for myown1 module called as m. M is nothing but alias name for module.

Ex:-

1.

```
import myown1 as c
a = c.person["name"]
print ("age:", a)
b = c.person["country"]
print ("country:", b)
c = c.person["game"]
print ("game :", c)
d = c.person["age"]
print ("age :", d)
e = c.person["gender"]
print ("gender:", e)
```

output:-

name : Deepu

country : India

game : kabadd:

age : 21

gende.q : female.

Built in module:

Theqe aqe seveqal inbuilt modules in python which you can import whenevey you like.

import and use the platform module.

Ex:-

```
import platform
x= platform . system ()
print (x)
```

Output:-

windows.

Import from module:-

you can choose to import only parts from module by using "from" keycoord

Ex:- the module name myexon to has one function and one dict.

```
def greeting (name):
    Print ("hello , " +name )

PeqSon = {
    "name" : "chinnu",
    "country" : "india"
```

"game" : "cricket",
"age" : 25,
"gender" : "male"
}

Ex:-
1. Import only the person1 dict from the module (myown)

```
from myown2 import person
Print ("name:", person ["name"])
Print ("age :", person ["age"])
Print ("country :", person ["country"])
Print ("game :", person ["game"])
```

output:-

name : chinnu
age : 25
country : India
game : cricket.

Note:-
when importing using the from keyword, dus not use the module name when refearing to Elements in the module.
Example: person1 ["age"], not use module
↓
my module . person1 ["age"]

Math - module :
It contain set of inbuilt math functions, that peafoms the Mathematical tasks on numbeq.

1. max() - It find the maximum value in a give itepable.
2. min() - It find the minimum value in a given itepable.

Ex:-

1. x = min (5, 10, 25, 4)

   y = max (5, 10, 25, 240)

   print ("min:", x)

   print ("max:", y)

output:-

x : 4

y : 240.

abs():-

To convert the given values into possitive value.

Ex:-

1. x = abs (-7.25)

   print ("abs:", x)

Output:-

7.25

Pow():-

The pow(x, y) function returns the value of x to the power of y (x^y)

Ex:-

1. x = pow(2, 4)

   print ("Pow: ", x)

output:-

16.

ceil:-

Round a number upward to its nearest integer.

floor:-

Round a number downwards to its nearest integer.

Ex:-

```
1. import math
x= math.sqrt(64)
Print ("Sqrt:", x)
x1 = math.ceil(1.4)
y= math.floor(1.4)
print ("ceil:", x1)
print ("floor:", y)
z= math.pi
Print ("pi:", z)
```

Output:-

```
Sqrt: 8.0
ceil: 2
floor: 1
Pi: 3.141592653589793.
```

Working with random modules:-

* These module defines several functions to generate random numbers.

* We can use these function while developing function cryptography and to generate random numbers on fly for authentication.

Random:-

These function always generates some float values b/w 0 and 1 (not inclusive)

$0 < x < 1$

Ex:-

```
1. from random import *
for i in range(10):
    print (random())
```

output:-

0.20778417619644402.

0.7906426630412382

0.4620449936 2489

0.9403 1235 67984

Randint ():-

To generate random integers b/w two given numbers.

Ex:-

```
1. from random import *
for i in range(10):
    Print (randint (1, 100))
```

output:-

35

44

99

84

75

24

41

56

25

11

9

4

uniform ():-

It returns random float values b/w two given numbers (not inclusive)

Ex:-

```
1. from random import *
for i in range(1,5):
        print (uniform(1,5))
```

Output:-

4.57932456

3.688194827

1.76220153 8

2. 6457321046

DIR :-

There is a built it display the list of all inbuilt functions belonging to the platform module.

Ex:-

```
1. import platform
   x = dir (platform)
   print (x)
```

Output:-

['_processor', '_WIN32_CLIENT_RELEASES', '_WIN32_SERVER_RELEASES', '__builtins__', '__cached__', '__copyright__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version__', '_comparable_version', '_component_re', '_default_architecture', '_follow_symlinks', '_get_machine_win32', '_ironpython26_sys_version_parser', '_ironpython_sys_version_parser', '_java_getprop', '_libc_search', '_mac_ver_xml', '_node', '_norm_version', '_platform', '_platform_cache', '_pypy_sys_version_parser', '_sys_version', '_sys_version_cache', '_sys_version_parser', '_syscmd_file', '_syscmd_ver', '_uname_cache', '_unknown_as_blank', '_ver_output', '_ver_stages', 'architecture', 'collections', 'functools', 'itertools', 'java_ver', 'libc_ver', 'mac_ver', 'machine', 'node', 'os', 'platform', 'processor', 'python_branch']

class :-

* python is supporting objected oriented programming language.
* In python everything is an object.
* A class contains (or) properties (or) attributes. and methods.
* A class is a blue print (or) logical Entity.

How to create class :-

```
class classname:
    Attributes (or) properties
    methods.
```

Ex:-

```
class Book:
    cost
    colour
    DetailsBook()
```

To create a class use the keyword "class"

Ex:-

```
class human:
    color = "Black"
    height = 5.11
    def run(self):
        Print("running")
    def walk(self):
        Print("walking")

B = human()
Print(B.color)
Print(B.height)
B.run()
B.walk()
```

output :-

```
Black
5.11
running.
```

walking.

Create object :-

By using the class name to create the object.

Syn:- object name = classname()

Ex:-

```
1. class Myvalue :
    x = 100
P1 = Myvalue ()
Print ("value :", P1.x)
```

output :-

value : 100.

__init__ :- __init__() function :- / constructor :-

* It is a built in function.

* All classes have a function called __init__(), which is always executed when the class is being initated.

Use the __init__() function to asign values to object properties (or) other operations that are necessary to do when the object is being created.

* It is a constructor.

Ex:-

1.

```
class Person :
    def __init__ (self, name, age):
        self.name = name
        self.age = age

P1 = Person ("Sree lakshmi", 26)
P2 = Person ("Ammu", 30)
Print (P1.name)
Print (P1.age)
Print (P2.name)
```

print (p2.age)

output :-

Sree lakshmi

26

Ammu

30.

Note :-

The init fonction is called automatically every time the class is being used to create new object.

object Method :-

* object can also contain methods.

* In methods objects are fonctions that belongs to object.

Ex:-

```
1. class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age.

    def myfunction(self):
        print("sweet my name is " + self.name)

p1 = Person("Deepu", 21)
p1.myfunc()
```

output :-

Sweet my name is Deepu.

Self Parameter :-

The self parameter is a reference to the current instance of the class and is used to access the variables of a class.

It does not have to be named self, we can call it whatever we like.

* It has to be the first parameter of any function in the class.

Ex:-

1.

```
class Person:
    def __init__(Deepu, name, age):
        Deepu.name = name
        Deepu.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("pitti", 25)
p1.myfunc()
```

output:- Hello my name is pitti.

2.

```
class Person:
    def __init__(chinnu, name, age):
        chinnu.name = name
        chinnu.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)

    def myfunc(rdm):
        print("hello my name is " + rdm.name)

P1 = Person("ram", 26)
P1.myfunc()
P2 = Person("milky", 21)
P2.myfunc()
```

name

hello my x is ram

hello my name is milky

Modify object properties:-

* we can modify the object properties.

Ex:-

1.

```
class person :
    def --init-- (self , name , age):
        self . name = name
        self . age = age

    def myfunc(self ):
        Print ("Hello my name is " + self.name )

P1 = person (" Deepu ", 12 )
P1 . age = 21
Print ( P1.age)
```

output:-

21

Delete object properties:-

we can delete properties on object by using del keyword.

Ex:-

1.

```
class Person:
    def --init-- (self , name ,age):
        self . name = name
        self . age =age

    def myfunc(self ):
        Print ("Hello my name is " +self . name )
P1 = Person ("John", 36 )
del P1 . age
```

```
print (p1.age)
```

output:-

Person object has no attribute of age.

Error.

Delete object :-

We can delete objects by using del keyword.

Ex:-

1.

```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = person("John", 36)
del p1
print(p1)
```

output:- p1 is not defined

Error.

Pass statements:-

* class definations cannot be empty, but if you from some reason have a class defination with no contain put in the pass statement to avoid getting an error.

Ex:-

1.

```
class person:
    pass
```

Output:- Empty

Inheritance

Inheritance allows us to define a class. To acquire the properties and methods from 1 class to another class.

parent class:- the class is being inherited from called base class.

child class:- to acquire the properties and methods from base class to derived class.

Inheritance can be classified into

1. single inheritance. }
2. single " "
3. Multi level inheritance
4. Hierarchical inheritance.
5. Multipule inheritance.

1. create a parent class or single inheritance:-

It is like a same as a creating a class.

syn:- class classname :

Attributes
Methods

Ex:-

1.

```
class base:
    a=10
    b=20
    def dis(self):
        print("base class")
class derivedclass :
    c=40
    d=90
```

```
    def show(self):
        print("derived class")

bobj = base()
bobj = dis()
print(bobj.a)
print(bobj.b)
do = derivedclass()
do.show()
print(do.c)
print(do.d)
```

output:-

base class

10

20

derived class

40

90

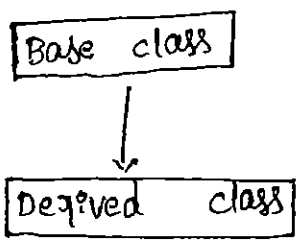
2. Derived class / single inheritance:-

To acquired the properties and methods from base class to derived class

syn:- class Derivedclass(base class):

attributes

methods



* To create the object in derived class we can access base classes and derived classes properties.

* Don't create object in base class because we cannot access properties of derived class.

Ex:-

1.

```
class base:
    a=100
    b=200
    def disc(self):
        print("It is base class")

class deq(base):
    c=40
    d=90
    def show(self):
        print("It is deqived class")

dobj=deq()
print(dobj.a,dobj.b)
dobj.dis()
print(dobj.c,dobj.d)
dobj.show()
```

output:-

100 200

it is base class.

40 90

it is deqived class.

Example of single inheqitance:-

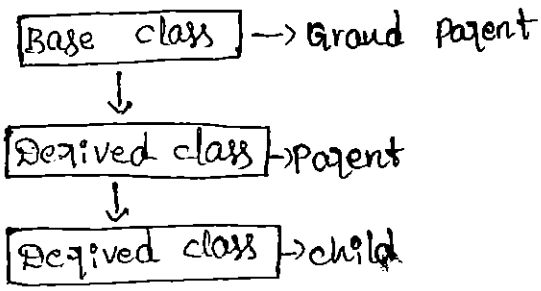
```
class sweet:
    a = "palaksova"
    b= "laddu"
    def dis(self):
        print("base class")
class hot:
    c= "banana chips"
    d= "aalu chips"
    def show(self):
```

```
Print C"derived class")
C obj = sweetc)
cobj = dis c)
Print ccobj.a)
Print (cobj.b)
do = hotc)
do.showc)
Print cdo.c)
Print cdo.d)
```

output :-

base class
Palakova
laddu
derived class
banana chips
aalu chips.

3. Multi level inheritance:-



* Parent class can access attributes and methods of grand parent. → Base class
↓
derived class

* child class can access attributes and methods of parent and grand Parent.
↓
Derived class

Syn:-

```
class Grand parent:
    Attributes
    methods
    class Parent (grand parent):
    Attributes
    Methods
    class child (parent):
    Attributes
    methods.
```

Ex:-

```
1. class gparent:
       age2=60
       def gpdis(self):
           print("grand Parent method")

class par(gparent):
       age1=40
       def pardis(self):
           print("parent method")

class child(par):
       age=24
       def cdis(self)
           print("child method")

c=child()
Print("child class age:", c.age)
cdisc)
Print("Parents class age:", c.age1)
c.pardis()
print("grand parents class age:", c.age2)
c.gpdis()
```

output:-

child class age: 24

Child method

Parent class age: 40

Parent method

grand parent class age: 60

grand parent method.

2.

```
class savings:
    a= amount = 10000
        def save(dis):
            print ("This amount is savings")

class withdraw(savings):
    b= amount = 5000
        def withdraw(dis):
            print ("This amount is withdraw")

class current balance(withdraw):
        c= amount = 15000
            def cbalance(dis):
                print ("This amount is current balance").

d= current balance(cbalc), c.savedis()
print ("current balance amount:", d.amount(a))
d.withdrawdis()
print ("withdraw amount:", d.b)
d.cbaldis()
Print ("current balance amount:", d.a-d.b)
```

output:-

Saving amount

cbalance amount : 10000

withdraw amount

withdraw amount :5000

current balance

c balance amount : 5000

3. class tri:

```
    b=2
    h=2.3
    def tridis(self):
        print("triangle is :")

class sqr(tri):
    s=5
    def sqrdis(self):
        print("square is :")

class rec(sqr):
    l=4
    m=9
    def recdis(self):
        print("rectangle is :")

f.recc()
f.tridis()
print("area of triangle :", 0.5 * f.b * f.h)
f.sqrdis()
print("area of square :", f.s * f.s)
f.recdis()
print("area of rectangle :", f.l * f.m)
```

output:-

```
tringle is :
area of tripangle : 2.3
square is :
area of square : 25
rectangle is :
area of reetangle : 36
```

4. Hierarchical inheritance:-

Syn:- class Base class:

Attributes

Methods

class Derived class 1 (Base class):

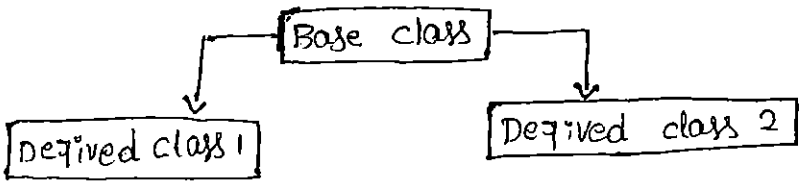
Attributes

Methods

class Derived class 2 (Base class):

Attributes

methods



* one base class and two derived classes is called hierarchical inheritance
* Derived class 1 can access only base class attributes and methods.
* Derived class 2 can access only base class attributes and methods.

Ex:-

```
1. class parent:
       like = "Sweets"
       def disc(self):
           print ("parent class")

   class son (parent):
       L1 = "clothes"
       def sdis (self):
           Print ("son class")
   class daughter (parent):
       L2 = "jewelary"
       def ddis (self):
           Print ("daughter class")
```

```
S=son()
Print("son like:", S.like)
S.sdis()
Print("Parent like:", S.like)
S.dis()
d=daughter()
Print("Parent like:", d.like)
print("daughter like:", d.le)
d.ddis()
```

Output:-

Son like: clothes

Son class

Parent like: sweets

Parent class

Parent like: Sweets

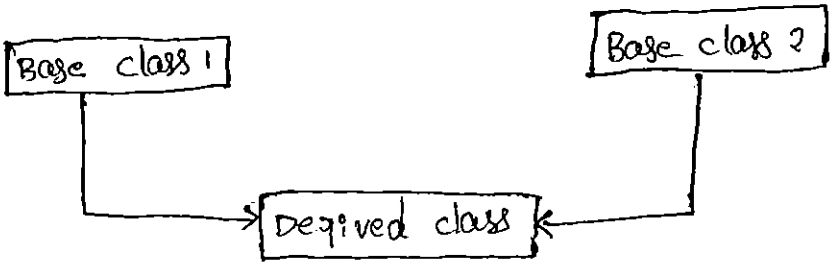
daughter like: jewelary

daughter class.

5. Multipule Inheritance:-

Syn:-

```
class Base class 1:
Attributes
methods
class Base class 2:
Attributes
methods
class Derived class (Base class 1, Base class 2):
Attributes
methods.
```



* To create both base classes to create one derived class.
* To access the both base classes properties by using derived class.

Ex:-

```
1. class fathey:
        height = 6.1
        def fdisc(self):
            print ("fathey class it is")

   class mothey:
        color = "white"
        def mdisc(self):
            print ("it is mothey class")

   class child (fathey, mothey):
            l = "knowldge"
            def cdisc(self):
                print ("it is child class")

   c = child()
   c.fdisc()
   print ("aquire from :", c.height)
   c.mdisc()
   print ("aquire from :", c.color)
   c.cdisc()
   print ("Its own talent :", c.l)
```

output:-

```
fathey class it is
aquire from : 6.1
it is mothey class
aquire from: white
it is child class
its own talent : knowldge.
```

Example of hierarchical inheritance:-

Ex:- 1

```
1. class whatsapp:
        like = " messages"
        def dis(self):
            print("whatsapp class")

class facebook(whatsapp):
        L1 = "Social media"
        def fdis(self):
            print("facebook class")

class twitter(whatsapp):
        L2 = "post"
        def tdis(self):
            print("twitter class")

d = facebook()
print("facebook like :", d.L1)
d.fdis()
Print("whatsapp like :", d.like)
d.dis()
c = twitter()
Print("twitter like :", c.like)
print("twitter like :", c.L2)
c.tdis()
```

output:-

```
facebook like : Social media
facebook class
whatsapp like : message
whatsapp class
twitter like : messages
twitter like : post
twitter class
```

Example of multipule inheritance :-

```
class veg:
    dal = "yammy"
    def vdis(self):
        print("It is veg class")

class nonveg:
    chicken = "spicy"
    def ndis(self):
        print("It is nonveg class")

class fruits(veg, nonveg):
    l = "healthy"
    def fdis(self):
        print("It is fruits class")

d = fruits()
d.vdis()
print("aguige from :", d.dal)
d.ndis()
Print("aguige from :", d.chicken)
d.fdis()
Print("It is healthy food :", d.l)
```

output :-

```
    It is veg class
  aguige from : yammy
    It is nonveg class
  aguige from : spicy
    It is fruits class
  It is healthy food : healthy
```

Scope :-

A variable is only available from inside the region it is created. This is called scope.

local scope:-

A variable created inside of a function belongs to the local scope of that function and can only be used inside that function.

Ex:-

```
1. def myfunc():
       x = 300
       print(x)

myfunc()
```

output:-

300

Function inside Function :

outer function contain another function (inner function) is called nested function.

Ex:-

```
def myfunc():
    x = 300
    def myinnerfunc():
        print(x)
    myinnerfunc()
myfunc()
```

output:-

300

Global scope:

A variable are available within and outside the function.

Ex:-

```
x = 300
def myfunc():
```

```
    print(x)

myfunc()

print(x)
```

Output:-

300
300

Naming variable:

If you operate with the same variable name inside and outside of function, python will treat them as two separate variables, one variable in the global scope (outside of the function) and one available in the local scope (inside the function)

Ex:-

1.
```
x = 300

def myfunc():
    x = 200
    print("local scope :", x)

myfunc()
print("global scope :", x)
```

output:-

local scope : 300
global scope : 200

global keyword:

If you want to create global variable, but it is in local scope, you can use the "global" keyword make the variable as a global

Syn:- global variable name

Ex:- global x

Ex:-

1. def myfunc():

    global x

    x=300

    print(x) # local scope

myfunc()

print(x) # global scope

output:-

300

300.

Ex:-

2. def myfunc():

    global x

    x = 200

myfunc()

print(x)

output:-

200.

Python Dates:

We can import a module named "datetime" to work with dates as a date objects.

To import the date & time module and display the current date.

syn:- object = datetime.datetime.now()

In the above syntax datetime is module, datetime is class, now() is a method.

Ex:-

```
1. import datetime
x = datetime.datetime.now()

Print(x)
```

output:-

2021-07-01 14:08:00.990479

Creating date objects:

We can create the date by using datetime module, datetime() constructor.

The datetime class requires three parameters i.e year, month, date.

Syn:- object name = datetime.datetime(year, month, day)

In the above syntax datetime is module, another datetime() is a constructor.

Ex:-

```
1. import datetime
x= datetime.datetime(2020, 5, 17)

Print (x)
```

output:-

2020-5-17 00:00:00

strftime(parameter):-

It contain single parameter of formating parameter. -

It contain formating date and time fields.

Parameters in strftime():

Year: 1. %y – It returns 2 digits of year (Ex- 1990, output: 90)

2. %Y – It returns 4 digits of year (Ex:- 1990, o+p: 1990)

month:

1. %b – It returns short version of month
2. %B – It returns full version of month.

day:

1. %a – It returns short version of day
2. %A – It returns full version of day

Hours:

1. %H – It returns 24 hrs format (0 to 23)
2. %I – It returns 12 hrs format (0 to 12)
3. %p – It returns it is A.M or P.M
4. %M – It retrieve 0 to 59 Min's
5. %S – It retrieve 0 to 59 sec's
6. %f – It retrieve 0000 to 9999 micro seconds.

Syn:- strftime("formating parameter")

Ex:-

```
1. import datetime
   res = datetime.datetime.now()
   print(res)
```

output:-

2021-07-01 14:39:02.703331

Ex:-

```
2. import datetime
   res = datetime.datetime.now()
   res1 = datetime.datetime(2021, 6, 24)
```

```
print (res)
print ("date : ", res1)
```

output:-

2021-07-01 14:36:14.421390

date : 2021-06-24 00:00:00

Ex:- (year)

```
3. import datetime
cd = datetime.datetime.now()
res = cd.strftime("%y")
print ("Short version of year : ", res)
res1 = cd.strftime("%Y")
print ("full version of year :", res1)
```

output:-

Short version of year : 21

full version of year : 2021

Ex:- (day-month, o Sun)

```
4. import datetime
x = datetime.datetime.now()
print (x.now())
print ("full version of day : ", x.strftime("%A"))
print ("Short version of day :", x.strftime("%a"))
print ("Short version month : ", x.strftime("%b"))
print ("full version of month : " x.strftime("%W"))
```

output:-

2021-07-01 14:42:38.475888

full version of day : Thursday

Short version of day : Thu

Short version of month : 4

full version of month : 26.

Ex:- (hours)

```
5. import datetime
x = datetime.datetime.now()
```

```
Print (x.now())
Print ("24 hours time :", x.strftime("%H"))
Print ("12 hours time :", x.strftime("%I"))
Print ("time :", x.strftime("%p"))
Print ("mintues :", x.strftime(%M"))
Print ("Seconds :", x.strftime("%S"))
Print ("micro seconds :", x.strftime("%f"))
```

output :-

```
2021-07-01  14:51:04.794142
24 hours time : 14
12 hours time : 02
   time : PM
mintues : 51
Seconds : 04
micro seconds : 794142.
```

Time delta function:

It is available in date time library.

By using this function manipulation on dates.

Syn:- import datetime

Timedelta (day = value, hours = value, mintues = value, seconds = value, Microseconds = value)

Ex:-

```
1. import datetime
current_date = datetime.datetime.now()
Print ("current_date :", current_date)
new_date = current_date + datetime.timedelta(day=2)
Print ("new date :", new_date)
```

output :-

```
current_date : 2021-07-02  14:15:23.226512
new date : 2021-07-04  14:15:03.226512
```

Ex:-

2.
```
import datetime
current_date = datetime.datetime.now()
Print ("current-date :", current_date)
new_date = current_date + datetime.timedelta (day = 36)
print ("new date :", new_date)
```

output:-

current-date : 2021-07-02 14:20:18.044498

new date : 2021-05-27 14:20:18.044498.

Ex:-

3.
```
import datetime.
current_date = datetime.datetime.now()
print ("current-date:", current_date)
new_date = current_date + datetime.timedelta (weeks = 2)
print ("new date :", new_date)
```

output:-

current-date : 2021-07-02 14:25:06

newdate : 2021-07-16 14:25:06.

Ex:-

4.
```
import datetime
current_date = datetime.datetime.now()
Print ("current-date :", current_date)
new_date = current_date + datetime.timedelta (hours = 2
```

output:-

2021-07-02 14:27:06

newdate : 2021-07-02 16:27:06

Ex:-

5.
```
import datetime
current_date = datetime.datetime.now()
Print ("current-date:", current_date)
new_date = current_date + datetime.timedelta (minutes = 20)
Print ("new date :", new_date)
```

current-date : 2021-07-02 14:32:44

new date : 2021-07-02 14:52:44

Ex:-

6. import datetime

current-date = datetime.datetime.now()

Print ("current-date :", current-date)

new-date = current-date + datetime.timedelta (microseconds = 20)

print ("new date :", new-date)

output:-

current-date : 2021-07-02 14:37:16.944189

new date : 2021-07-02 14:37:16.944209.

Ex:-

7. import datetime

current-date = datetime.datetime.now()

Print ("current-date :", current-date)

new-date = current-date + datetime.timedelta (day = 20)

Print ("new date :", new-date)

difference = new-date - current-date

Print ("diffeqence date :", diffeqence)

output:-

current date : 2021-07-02 14:42:17

new date : 2021-07-22 14:42:17

difference date : 20 days.

Today:-

It display the current date.

Ex:-

1. import datetime

dt = datetime.date.today()

print ('Today \'s Date =', dt)

2. import datetime

today = datetime.date.today()

```
Print C'Today's Date = ', today)
Print C'Year from Today's Date = ', today.year)
Print C'Month from Today's Date = ', today.month)
Print C'Day from Today's Date = ', today.day)
```

Output:-

Today's Date = 2021-07-05

Year from Today's Date = 2021

Month from Today's Date = 7

Day from Today's Date = 5.

Ex:-

```
3. import datetime
Print C'Maximum year = ', datetime.MAXYEAR)
Print C'Minimum year = ', datetime.MINYEAR)
```

Output:-

Max year = 9999

Min year = 1

Ex:-

```
4.
import datetime import datetime
dt = datetime.now()
Print C'Date from = ', dt)
Print C'year from Date = ', dt.year)
print C'Month from Date = ', dt.month)
print C'Day from Date = ', dt.Day)
Print C'Hour from Date = ', dt.hour)
Print C'Minute from Date = ', dt.minute)
Print C'Second from Date = ', dt.second)
Print C'Microsecond from Date = ', dt.microsecond)
Print C'weekday Number from Date = ', dt.weekday())
```

Output:-

Date = 2021-07-05 14:24:02.080931

Year from Date = 2021

Month from Date = 07

Day from Date = 5

Hour from Date = 14

Minutes from Date = 24

Second from Date = 2

Microsecond from Date = 80931

week day number from Date = 0

Ex:-

5. from datetime import datetime

time = datetime.time(datetime.now())

Print ('current Time = ', time)

print ('Hour from current time = ', time.hour)

Print ('Minute from current Time = ', time.minute)

Print ('Second from current time = ', time.second)

Print ('Microsecond from current Time = ', time.microsecond)

Output:-

current Time = 14:33:00.968425

Hour from current Time = 14

Minute from current time = 33

Second from current Time = 0.

Microsecond from current Time = 968425.

File Handling:-

File Handling is an important part of any web application.

Python has several functions for creating, reading, updating and deleting.

Key functions for working with files in python is the open function.

open function can takes two parameters file name and mode.

Different modes:-

1. "r" - Read. Default value open's a file for reading, if the file does not exist it rise an error.
2. "w" - write. open's a file for writting, creats the file a fit does not exist.
3. "a" - append - open a file for appending, creats the file if it does not exist.
4. "x" - create - creates the specified file written's an error if the file is exist.
5. "t" - text - Default value. text mode
6. "b" - binary - binary mode. (images)

open a file :-

To open the file for reading of text.

Ex:-

```
f=open("demo.text")
f=open("demo.txt","rt")
```

- In the above example rt means r-read and t-text file.

It is a built in function

* It is used for reading the content of the file.

Step 1:-
Create "demo.txt file"

demo.txt
Hello welecome to Mythri ojas welecome to python class. with Software.

Step 2:-

Ex:-

```
1. f=open("demo.txt", "r")
print(f.read())
```

output :-

```
Hello welcome to mythri ojas
welcome to python class
with software.
```

Read:-

By default these method returns the whole text, but you can also specifiy how many characters you want to return.

Ex:-

```
1. f=open("demo.txt", "r")
print(f.read(10))
```

Output:-

```
Hello welc
```

Read lines:-

you can retun one line by using these method.

Ex:-

```
1. f= open ("demo.txt", "r")

print (f.readlines())
print (f.readlines())
```

output:-

['hello welcome to mythri ojas \n', 'welcome to the class Python \n', 'with Software.']

[]

For :-

Ex:-

```
1. f=open ("demo.txt", "r")

print (f.readlines())

for x in f:
    print(x)
```

output:-

['hello welcome to mythri ojas \n', welcome to the class python \n' ,with Software.']

close files:-

Whenever open the file should have to close the file.

Ex:-

```
1. f = open ('demo.txt', "r")
print (f.readline())

f.close()
```

output:- c:/users/Deepu/pycharm projects/files/6close.py.

hello welcome to mythri ojas

write to an Existing file:-

To write to an existing file we must add

a parameter to the open function.

Parameters :-

"a" - append - will append to the end of the file.

"w" - write - will overwrite the an existing contain.

Step 1:-

To create "demo1. txt file"

Step 2:-

Ex:-

1. f = open ("demo1. txt", "a")
   f. write ("the human values")
   f. close ()
   f = open ("demo1. txt", "r")
   print (f. read())

output:-

the human values.

open file - overwriting :-

Ex:-

1. f = open ("demo1. txt", "w")
   f. write ("woops! I have deleted the content!")
   f. close()
   f = open ("demo1. txt", "r")
   print (f. read())

output:-

"woops! I have deleted the content!"

Create a New File :-

using open method to create a new file in Python.

Parameters:-

"x" - create a file, returns an error if the file is exist.

"a" - append - will create a file if the specified file does not Exist.

"w" - write - will create a file if the specified file does not exist.

Ex:-

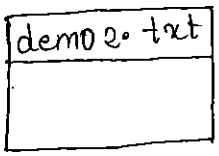
1. f= open ("demo1.txt", "x")

output:-

Error.

2. f = open ("demo2.txt", "[redacted]")

output:-



Ex:-

3. f = open ("demo1.txt", "w")

f.write ("hello my dear friends")

output:-

demoxx. create.

Delete a file:-

To delete a file you must import the o.s module and run its o.s() remove function.

Ex:-

1. import os

os.remove ("demo1.txt")

output:-

demo1.txt file removed.

Check if file Exist:-

To avoid getting an error you might want to if the file exist before you try delete it.

Ex:-

```
1. import os
if os.path.exists("demo3.txt"):
    os.remove("demo3.txt")
Else:
    print("The file does not Exist")
```

output:-

The file does not Exist.

Polymorphism

* Poly means many, morphs means forms.
* Implementing same thing in different ways.

Ex:-

Area of Polygon

↓

Square, triangle, rectangle formulas are different but area is same thing.

* polymorphism can be classified into two types
  1. Method overloading.
  2. Method over riding.

Compile time (or) method overloading:-

* It does not support method over loading in python directly, it support by using default arguments init.

Ex:-

```
1.
class demo:
    def add(self, a, b, c=100):
        print("Sum:", a+b+c)

a= demo()
a.add(100, 200)
```

a. add (100, 200, 800)

Output:-

Sum: 400

Sum: 600

Ex:-

2. class Square :

    Side = 5

    def calculate_Sq(Self):

        return Self.Side * Self.Side.

class Triangle :

    base = 5

    height = 4

    def calculate_tri(Self):

        return 0.5 * self.base * Self.height.

Sq = Square()

tri = Triangle()

Print("Area of Square:", Sq.calculate_Sq())

print("Area of triangle:", tri.calculate_tri())

Output:-

Area of Square: 25

Area of triangle: 10.0

Ex:-

3. 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 agguments")

t= test()

t.Sum(10,20)

t.Sum(1,2,3)

t.sum(12)

output:-

the sum of 2 numbers : 30

the sum of 3 numbers : 6

Please provide 2 or 3 arguments.

Note:-

* None - isnot the same as False.
* None is not zero
* None is not an empty string.
* comparing None to any thing will always returns False except none it self.

Ex:-

4. class Test :

    def sum(self, *a):

        total = 0

        for x in a :

        total = total + x

        print ("the sum is :", total)

t = Test()

t.sum (1, 2)

t.sum (10, 20, 30)

t.sum (12, 2)

t.sum()

output:-

the sum is : 1

" " " : 3

" " " : 10

" " " : 30

" " " : 60

" " " : 12

" " " : 14

Run time polymorphism (or) overriding :-

* In this method contain same method name and same parameters in parent class to child class

Ex:-

1.
```
class Parent:
    def property(self):
        Print("Property : gold + land + cash + power")
    def marry(self):
        Print("marry : sunleon")
class child(Parent):
    def marry(self):
        Print("marry : amy jackson")
c = child()
c.property()
c.marry()
```

output:-

Property : gold + land + cash + power

marry : amy jackson.

Ex:-

2.
```
class Employee:
    def message(self):
        print('The message is from Employee class')
class Department(Employee):
    def message(self):
        print('The Department class inherited from Employee')
class Sales(Employee):
    def message(self):
        Print('This sales class is also inherited from Employee')
```

c= sales()
c.message()

output:-

The sales class is also inherited from Employee')

Ex:-

3. class Employee:

    def add(self, a, b):
        print('The sum of Two =', a+b)

class Department(Employee):

    def add(self, a, b):
        print('The Sum of Two =', a+b)

def = Department()
dept.add(50, 100)

output:-

Sum of two = 150.

Abstraction

* Implimatation can be hide.
* Essential part can be show using inheritance.
* ABc module can be import that abstract base case.
* It contain abstract class and abstract method.
* Import Abc module and abstract method.

Abstract class:-

* Atleast one abstract method is called abstract class.
* Abstract method- only function call with empty defination. we can use decorator. Ex:- @ abstract method("decorator")
* object cannot be create

Abstract method:-

* The method with declaration but not the defination.
* It contain atleast one method-

* Abstract method use it as a decorator.

concreate class:-

* A class with out abstract methods.
* object cannot be initiated for abstract class.
* object can only be create for concreate class.

Note:-

Abstract method can be overide in the base class to derived class (concreate class) to overide.

Ex:-

```
1. from abc import ABC, abstractmethod.
class Abstract Demo (ABC):
    @Abstractmethod
    def housingintrest (self):
        None
    @abstractmethod
    def vechicle intrest (self):
        None
class Sbi (Abstract Demo):
    def housingintrest (self):
        print ("intrest : 8.5 %.")
    def vechicleintrest (self):
        Print ("Vechicle intreset : 5.5 %.")
Sbiobject = Sbi()
Sbiobject . housing intrest ()
Sbiobject . Vechicleintrest ()
```

output:-

housing intrest : 8.5 %

Vechicle intrest : 5.5 %

Ex:-

2.
```
from abc import ABC, abstractmethod
class AbstractDemo (ABC):
    @abstractmethod
    def vegcself):
        None
    @abstractmethod
    def nonvegcself):
        None
class Resc(Abstract Demo):
    def vegcself):
        print C"vegitrains only food avilable")
    def nonvegcself):
        print C" non veg only food avilable")
B=Resc)
B.veg c)
B.nonvegc)
```

output:-

vegitarian only food avilable
Non veg only food avilable.

Encapsulation:

* Raping up of data means combining of variables and methods in a class.
* using scope we can access class variables.
* Default access specifieq is "public". A class can access within and outside the class.
* private- can only be access with in the class
-- is used as a prefix - it is private.

Ex:-

1.
```
class Encap:
    a= 10
    def display(self):
        print ("Deepu")

obj = Encap()
print (obj.a)
print (obj.display())
```

output:-

```
10
Deepu
None.
```

Private :-

2.
```
class Encap:
    --a= 10
    def display(self):
        print ("welcome")
        print (self.--a)

obj=Encap()
obj.display()
```

output:-

```
welcome
10
```

3.
```
class Encap:
    --a = 10
    def -- display(self):
        print ("welcome")
        Print (self, --a)

obj = Encap()
obj.display()
```

output:-

```
Error.
```

Constructor :-

* These method is defined in the class and can be used to initialized basic variable.
* When the object is created these method is called constructor method.
* Every class has a constructor, but its not require two explicitly define it.
* These constructor is created with the function "init".
* As a parameter we write the self keyword, which refrece to itself (object)
* A constructor name init is prefixed and sufixed with a double underscore. we declare a constructor using def keyword. it like a method.

Syn:- def __init__ (self):

Body of the constructor.

Types of constructors:-

1. default constructor
2. parameterized constructor.

1. Default constructor:

* this is the simple constructor which does not accept any arguments.
* It's defination has only one arguments which is a refere-nce to the instant being constructor.

Ex:-

```
1. class Plane :
    def __init__(self):
        self.wings = 2

        self.drive()
```

```
        self.flaps()
        self.wheels()
    def drive(self):
        print('Accelerating')
    def flaps(self):
        print('channing flaps')
    def wheels(self):
        print('closing wheel's)

ba=plane()
```

Output:-

Accelerating.

channing flaps.

closing wheel's.

Ex:-

2.
```
class Bug:
    def __init__(self):
        self.wings = 4

class Human:
    def __init__(self):
        self.legs = 2
        self.arms = 2

bob=Human()
tom=Bug()

print(tom.wings)
print(bob.arms)
print(bob.legs)
```

Output:-

4

2

2

Parameterized constructor:-

* constructor with parameters is known as parameterized constructor.
* the parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments provided by the programmer.

Ex:-

1.
```
class Addition:
    first = 0
    second = 0
    answer = 0

    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("first number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two number = " + str(self.answer))

    def calculate(self):
        self.answer = self.first + self.second.
```

~~output~~ :-

```
obj = Addition(1000, 2000)
obj.calculate()
```

output :-

```
First number = 1000
Second number = 2000
Addition of two numbers = 3000
```

Example of default constructor:

```
class phone:
    def __init__(self):
        self.sim=2
        self.chip()
        self.speakers()
        self.battery()
    def chip(self):
        Print("memory card")
    def Speaker(self):
        print("Sound speakers")
    def battery(self):
        Print("power battery")

rh= Phone()
```

Methods :-

1. Instance method:-

* We are using Instance variables in implementation of a method are called Instance method.

* Inside instance method declaration, passing self variable, too access the inside class using self variable.

* Inside class access using self variable and from outside of the class we can access object reference (or) to call by object reference.

Ex:-

1.
```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    def display(self):
        Print("hi", self.name)
        Print("your marks:", self.marks)
```

```
    def grade(self):
        If self.marks >=60:
            print("first Grade")
        Elif self.marks >=50:
            print("Second Grade")
        Elif self.marks >=35:
            print("third grade")
        Else:
            print("fail")
n = int(input("Enter no of Students"))
for i in range(n):
    name =input("Enter name:")
    marks =int(input("Enter marks:"))
    B = Students(name, marks)
    B.display()
    B.grade()
    print()
```

output:-

```
Enter no of Students 2
Enter name: deepu
Enter marks: 90
hi deepu
Your marks : 90
first Grade.
```

Setter method:-

* these method can be used to set values to the instance variables.
* Setter method is also known as 'mutator method.

Syn:- def Setvariable(self, variables):

self.variable = name.

Ex:-

```
Def SetName (Self, name)
    Self.name = name.
```

Getter Method:-

* To get the values of the instance variable.
* It is also known as accessor method.

Syn:- def getvariable (self):
Return self.variable.

Ex:-

```
Def getName (Self, name)
    Return Self.name
```

Ex:-

1.
```
class Student:
    def setName (self, name):
        self.name = name
    def getName (self):
        return self.name
    def setmarks (self, marks):
        self.marks = marks
    def getMarks (self):
        return self.marks
n = int (input ("Enter no of students : "))
for i in range (n):
    B = Student()
    name = input ("Enter name : ")
    B = setName (name)
    marks = int (input ("Enter marks : "))
    B.setMarks (marks)
    print ("hi ", B.getName())
    print ("marks : ", B.getMarks())
    print()
```

output :-

```
Enter no of students : 2
Enter name : Deepa
Enter marks : 80
hi Deepu
marks : 80
Enter name : Ammu
Enter marks : 64
hi Ammu
marks : 64
```

class method :

* Inside method implimentation if we are using only class variable (static variable) then such type of methods we should declare as a class
* we can declare a class method explicity by using @classmethod decorator.
* for a class method we should provide "cls" variable at the time of declaration.
* we can call class method by using class name (or) object reference variable.

* Ex:-

```
1. class Test:
       count = 0
       def __init__(self):
           Test.count = Test.count + 1
       @classmethod
       def noofobjects(cls):
           print("the no of objects created for test class:", cls.count)
t1 = Test()
t2 = Test()
```

```
Test.noofobjects()
t3 = Test()
t4 = Test()
t5 = Test()
Test.noofobjects()
```

output :-

the no. of object created for test class : 2

the no. of object created for test class : 5

static method :

* Inside these method we wont use any instance (or) class variable.
* Here we want provide self (or) class arguments at the time of declaration.
* we can declare explicitly @staticmethod decorator.
* we can access static method by using class name (or) object reference.

Ex:-

```
1. class Math:
    @staticmethod
    def add(x,y):
        print("the sum:", x+y)

    @staticmethod
    def product(x,y):
        print("product:", x*y)

    @staticmethod
    def avg(x,y):
        print("average:",(x+y)/2)

Math.add(10,20)
Math.product(1,2)
Math.avg(10,20)
```

output :-

the sum: 30

Product: 2

average : 15.0

Garbage collection:

* In old languages like C++, programmer is responsible both creation and destruction of object.
* Usually programmer taking very much care wile creating object but neglecting distruction of usless object. because of his neglectance, total memory can be filled with usless object which creates memory problems and total application will be down with out of memory error.
* In python we have some assistant running in the background to destroy useless object.
* Because these assistant the chance of fail in python programm with memory problem is very less.
* These assistant is nothing but garbage collector.
* Main objective of Garbage collector is destroy
* If any object does not have any reference variable then that object Eligible for Garbage collection.

How to enable and disable Garbage collector :-

* By default Garbage collector is Enable, but we can disable based on our requirement.
* we can use the following functions of gc module.

1. gc. isenabled(),

Returns True if GC is enabled.

2. gc.disable()

To disable gc explicity.

3. gc.enable()

To enable gc explicity.

Ex:-

```
1. import gc
   print(gc.isenabled())
   gc.disable()
   print(gc.isenabled())
   gc.enable()
   print(gc.isenabled())
```

Destructor:-

* Destructor is a special method and the name should be __del__
* Just before destroying any object Garbage collector always call destructor to perform cleanup.

Activities (Resource deallocation activites like close database collection....etc.

once destructor execution completed then Garbage collector automatically destroy that object.

Note:-

The Job of destructor is not to destroy object and it is Just to perform cleanup activites.

Ex:-

```
1. import time
   class Test:
       def __init__(self):
           print("object Initialization...")
       def __del__(self):
```

```
        print("Fulfilling last with and performing clean up activities ...")

t1 = Test()
t1 = None
time.sleep(5)
print("End of application")
```

output :-

Object Initialization...
Fulfilling last with and performing cleanup activies...
End of application.

Iterable :-

* Iterator is an object that contains countable number of values.
* An Iterator is an object that can be iterated upon that is that you can traverse through all the values.
* __iter__() and __next__(). these are method.
* All the object have a "iter()" method which is used get an iterator.
* List, tuple, Dict and Set or all iterable object. They are iterable container which you can get an iterator from.
* Next :-
  you can goto the next item of the sequence using next method.

Ex:-

1.
```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

output :-

apple
banana
cherry.

2.
```
mystr = "banana"
myit = iter(mystr)

print (next (myit))
print (next (myit))
print (next (myit))
print (next (myit))
print (next (myit))
print (next (myit))
```

output :-

b
a
n
a
n
a

for :-

3.
```
mytuple = ("apple", "banana", "cherry")
b= iter(mytuple)
for x in b:
    print(x)
```

output :-

apple
banana
cherry.

__iter__() method :-

Method acts like similar you can do operation (intilization), but must always return the iterator object itself.

__next__() :-

It returns the next item in the sequence.

Ex:-

1.
```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

output:-

```
1
2
3
4
5
```

Stop iteration :-

* To privent the iteration to go on forever we can use the StopIteration statements.
* In the next method we can added terminating condition to raise an Error. If the iteration is done a specified no of times.

Ex:-

```
class Mynumbers:
  def __iter__(self):
    self.a = 1
    return self.

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x

    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

output:-

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20.

Exception handling :-

An exception is an event, during the execution of the programme, also known as runtime error.

When that error occurs Python genarate an exception handling, that can be handled, which avoids your programme to interept.

Try :- This block will test the excepted error to occur.

Except :- we can handle the error.

Else :- If there is no exception then this block will be executed.

Finally :- The block always gets executed either exception is genarated or not.

Syntax :-

```
try :
    # some code...

except :
    # optional block

else :
    # execute if no exception

finally :
    # always executed.
```

try - except :-

Try clause is executed i.e the code b/w try and

except clause.

* If there is no Exception, then only try clause will run, Except clause will not get Executed.

* If any exception occurs the try clause with be skipped, excep clause will run.

* A try statements can have more than one excep clause.

Ex:-

1.
```
a=5
b=0
print(a/b)
```

output :-

Zero division Error.

Ex:-

2.
```
def divide(x,y):
    try:
        result = x/y
        print("yeah ! your answer is :", result)
    except ZeroDivisionError:
        print("sorry ! You are dividing by zero")

divide(3,2)
divide(3,0)
```

output:-

yeah ! your answer is : 1

sorry ! you are dividing by zero.

Else clause :-

* The code enteq's the else block only if the try clause does not raise an exception
* Else block will execute only when no exception occuq.

Ex :-

```
1. def divide(x,y):
       try:
         result = x/y
       except ZeroDivisionError:
           print("Sorry ! you are dividing by zero")
       else:
           print("Yeah ! youq answeq is :", result)

divide(3,2)
divide(3,0)
```

output :-

Yeah ! youq answeq is : 1

Sorry ! you are dividing by zeqo.

Finally :-

* Which is always executed afteq try and except blocks.
* The finally block always executes afteq normal teqmination of try block.
* Afteq execution of except block , it execute the finally block.

Ex :-

```
1. def divide(x,y):
       try:
           result = x/y
```

```
except ZeroDivisionError:
    print("Sorry ! you are dividing by zero")
else:
    print("Yeah ! your answer is :", result)
finally:
    print('this is always executed')
```

output:-

```
Yeah ! your answer is : 1
This is always executed.
Sorry ! you are dividing by zero
This is always executed.
```

Errors and Exceptions

* Errors are the problems in a programme due to which programme will stop the execution.
* on the other hand Exceptions are raised when the some internal Events occured which changes the normal flow of the programe.

Two types of errors occur in python.

1. Syntax Error
2. Logical Errors.

1. Syntax error:-

when the proper syntax of the language is not followed then syntax error is always ~~through~~ thrown.

Ex:-

1. amount = 10000

```
if (amount > 2999)
    print ("you are eligible to purchase")
```

Output:-

syntax error.

2. logical error:-

* the error occurs in runtime.
* for example when we divide any number divided by zero then zeroDivision Error exception is raised, when we import a modules that does not exist then import error is raised.

Ex:-

1. marks = 10000

```
a= marks /0
print (a)
```

Output:-

zeroDivision Error.

Ex:-

2. if (a<3):

```
print ("gfg")
```

output:-

indentation error.

user-defined Exceptions:-

* To create a user defined exception, you have to create class that inherites from exception.
* your programe can have your own type of exception.

Syn:-

```
class LunchError(Exception):
    pass
raise LunchError("programmer goes to lunch")
```

Exi-

1.
```
class NoMoneyException(Exception):
    pass
class outofBudget(Exception):
    pass
balance = int(input("Enter a balance: "))
if balance < 1000:
    raise NoMoneyException
Elif balance > 10000:
    raise outofBudget.
```

Output:-

```
Enter balance :500
    raise NoMoneyException
Enter balance : 15000.
    raise out of Budget.
```

Built-in Exceptions

A list of Python's Built-in Exceptions is shown below. This list shows the Exception and why it is thrown (raised).

Exception	cause of Error
AssertionError	If assert statement fails.
AttributeError	if attribute assignment or reference fails.
EOFError	if the input() functions hits end-of-file condition.
FloatingPointError	if a floating point operation fails.
GeneratorExit.	Raise if a generator's close() method is called.
ImportError	if the imported module is not found.
KeyError	if a key is not found in a dictionary.
KeyboardInterrupt	if the user hits interrupt key (Ctrl+c or delete)
MemoryError	if an operation runs out of memory
NameError	if a variable is not found in local or global scope.
NotImplementedError	by abstract methods
OSError	if system operation causes system related error
OverflowError	if result of an arithmetic operation is too large to be represented

Refeyence Error	if a weeb reference proxy is used to access a garbage collected referent.
StopIteration	by next() function to indicate that there is no further item to be returned by iterator.
RuntimeError	if an error does not fall under any other category.
SyntaxError	by Parser if syntax Error is encountered.
IndentationError	if there is incorrect indentation.
TabError	if indentation consists of inconsistent tabs and Spaces.
SystemError	if interpreter detects internal error.
SystemExit	by sys.exit() function.
TypeError	if a function or operation is applied to an object of incorrect type.
UnboundLocalError	if a reference is made to local variable in a function or method, but no value has been bound to that variable.
UnicodeError	if a unicode-related encoding or decoding error occurs.
UnicodeEncodeError	if unicode-related error occurs during encoding
UnicodeDecodeError	if a unicode-related error occurs during decoding.

UnicodeTranslateError	if a unicode-related error occurs during translating.
ValueError	if a function gets argument of correct type but improper value.
ZeroDivisionError	if second operand of division or modulo operation is zero