**Object**

An object is an entity that has attributes and behaviour. For example, Ram is an object who has attributes such as height, weight, color etc. and has certain behaviours such as walking, talking, eating etc.

**Class**

A class is a blueprint for the objects. For example, Ram, Shyam, Steve, Rick are all objects so we can define a template (blueprint) class Human for these objects. The class can define the common attributes and behaviours of all the objects.

**Methods**

As we discussed above, an object has attributes and behaviours. These behaviours are called methods in programming.

## Example of Class and Objects

class Human:

# instance attributes

def \_\_init\_\_(self, name, height, weight):

self.name = name

self.height = height

self.weight = weight

# instance methods (behaviours)

def eating(self, food):

return "{} is eating {}".format(self.name, food)

# creating objects of class Human

ram = Human("Ram", 6, 60)

steve = Human("Steve", 5.9, 56)

# accessing object information

print("Height of {} is {}".format(ram.name, ram.height))

print("Weight of {} is {}".format(ram.name, ram.weight))

print(ram.eating("Pizza"))

print("Weight of {} is {}".format(steve.name, steve.height))

print("Weight of {} is {}".format(steve.name, steve.weight))

print(steve.eating("Big Kahuna Burger"))

## What is a Constructor in Python?

Constructor is used for initializing the instance members when we create the object of a class.

For example:  
Here we have a instance variable num which we are initializing in the constructor. The constructor is being invoked when we create the object of the class (obj in the following example).

class DemoClass:

# constructor

def \_\_init\_\_(self):

# initializing instance variable

self.num=100

# a method

def read\_number(self):

print(self.num)

# creating object of the class. This invokes constructor

obj = DemoClass()

# calling the instance method using the object obj

obj.read\_number()

### Syntax of constructor declaration

def \_\_init\_\_(self):

## Types of constructors in Python

We have two types of constructors in Python.

1. default constructor – this is the one, which we have seen in the above example. This constructor doesn’t accept any arguments.

2. parameterized constructor – constructor with parameters is known as parameterized constructor.

### 2.1 Python – default constructor example

Note: An object cannot be created if we don’t have a constructor in our program. This is why when we do not declare a constructor in our program, python does it for us. Lets have a look at the example below.

def \_\_init\_\_(self):

### 2.2 Python – Parameterized constructor example

def \_\_init\_\_(self,name=’name’,ag=’10’):

## Overview of OOP Terminology

* **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
* **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
* **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
* **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
* **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
* **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
* **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
* **Instantiation** − The creation of an instance of a class.
* **Method** − A special kind of function that is defined in a class definition.
* **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
* **Operator overloading** − The assignment of more than one function to a particular operator.

## Class Variables

Class variables are defined within the [class construction](https://www.digitalocean.com/community/tutorials/how-to-construct-classes-and-define-objects-in-python-3). Because they are owned by the class itself, class variables are shared by all instances of the class. They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.

**class** **Shark**:

animal\_type = "fish"

new\_shark = Shark()

print(new\_shark.animal\_type)

## Instance Variables

Unlike class variables, instance variables are defined within methods.

**class** **Shark**:

**def** **\_\_init\_\_**(self, name, age):

self.name = name

self.age = age

# **Inheritance in Python**

Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:

.

1. It provides **reusability** of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
2. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

class Person(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

    # To check if this person is employee

    def isEmployee(self):

        return False

class Person(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

    # To check if this person is employee

    def isEmployee(self):

        return False

**Output:**

Geek1 False

Geek2 True

|  |
| --- |
| class Person( object ):            # \_\_init\_\_ is known as the constructor          def \_\_init\_\_(self, name, idnumber):                  self.name = name                  self.idnumber = idnumber          def display(self):                  print(self.name)                  print(self.idnumber)    # child class  class Employee( Person ):          def \_\_init\_\_(self, name, idnumber, salary, post):                  self.salary = salary                  self.post = post                    # invoking the \_\_init\_\_ of the parent class                  Person.\_\_init\_\_(self, name, idnumber)      # creation of an object variable or an instance  a = Person('Rahul', 886012)    # calling a function of the class Person using its instance  a.display() |

**Output:**

Rahul

886012

‘a’ is the instance created for the class Person. It invokes the \_\_init\_\_() of the referred class. You can see ‘object’ written in the declaration of the class Person. In Python, every class inherits from a built-in basic class called as ‘object’. The constructor i.e. the ‘\_\_init\_\_’

**Different forms of Inheritance:**  
**1. Single inheritance**: When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.

**2. Multiple inheritance**: When a child class inherits from multiple parent classes, it is called as multiple inheritance.  
Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as comma separated list in bracket.

|  |
| --- |
| # Python example to show working of multiple  # inheritance  class Base1(object):      def \_\_init\_\_(self):          self.str1 = "Geek1"          print "Base1"    class Base2(object):      def \_\_init\_\_(self):          self.str2 = "Geek2"          print "Base2"    class Derived(Base1, Base2):      def \_\_init\_\_(self):            # Calling constructors of Base1          # and Base2 classes          Base1.\_\_init\_\_(self)          Base2.\_\_init\_\_(self)          print "Derived"        def printStrs(self):          print(self.str1, self.str2)      ob = Derived()  ob.printStrs() |

**Output:**

Base1

Base2

Derived

('Geek1', 'Geek2')

Output :

Base1

Base2

Derived

Geek1 True E101

**3. Multilevel inheritance**: When we have child and grand child relationship.

|  |
| --- |
| # A Python program to demonstrate inheritance    # Base or Super class. Note object in bracket.  # (Generally, object is made ancestor of all classes)  # In Python 3.x "class Person" is  # equivalent to "class Person(object)"  class Base(object):        # Constructor      def \_\_init\_\_(self, name):          self.name = name        # To get name      def getName(self):          return self.name      # Inherited or Sub class (Note Person in bracket)  class Child(Base):        # Constructor      def \_\_init\_\_(self, name, age):          Base.\_\_init\_\_(self, name)          self.age = age        # To get name      def getAge(self):          return self.age    # Inherited or Sub class (Note Person in bracket)  class GrandChild(Child):        # Constructor      def \_\_init\_\_(self, name, age, address):          Child.\_\_init\_\_(self, name, age)          self.address = address        # To get address      def getAddress(self):          return self.address    # Driver code  g = GrandChild("Geek1", 23, "Noida")  print(g.getName(), g.getAge(), g.getAddress()) |

**Output:**

Geek1 23 Noida

**4. Hierarchical inheritance** More than one derived classes are created from a single base.

**4. Hybrid inheritance**: This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

**Private members of parent class**  
We don’t always want the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won’t be available to the child class.  
We can make an instance variable by adding double underscores before its name. For example,

|  |
| --- |
| # Python program to demonstrate private members  # of the parent class  class C(object):         def \_\_init\_\_(self):                self.c = 21                  # d is private instance variable                self.\_\_d = 42  class D(C):         def \_\_init\_\_(self):                self.e = 84                C.\_\_init\_\_(self)  object1 = D()    # produces an error as d is private instance variable  print D.d |

Output :

File "/home/993bb61c3e76cda5bb67bd9ea05956a1.py", line 16, in

print (D.d)

AttributeError: type object 'D' has no attribute 'd'

Since ‘d’ is made private by those underscores, it is not available to the child class ‘D’ and hence the error

# **Polymorphism**

Sometimes an object comes in many types or forms. If we have a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick().  We access them using the same method . This idea is called Polymorphism

**class** Pdf(Document):

**def** show(self):

**return** 'Show pdf contents!'

**class** Word(Document):

**def** show(self):

**return** 'Show word contents!'

**def** showdocument(doctype):

doctype.show()

ob1 = Pdf()

ob2 = Word()

showdocumnet(ob1)

showdocumnet(ob2)

OUTPUT

'Show pdf contents!'

'Show word contents!'

# **Encapsulation**

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

## Private methods

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

**class** Car:

**def** \_\_init\_\_(self):

self.\_\_updateSoftware()

**def** drive(self):

**print** 'driving'

**def** \_\_updateSoftware(self):

**print** 'updating software'

redcar = Car()

redcar.drive()

## Encapsulation prevents from accessing accidentally, but not intentionally. The private attributes and methods are not really hidden, they’re renamed adding “\_Car” in the beginning of their name. The method can actually be called using redcar.\_Car\_\_updateSoftware()

Variables can be private which can be useful on many occasions. A private variable can only be changed within a class method and not outside of the class.

Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in the code.  
An example

**class** Car:

\_\_maxspeed = 0

\_\_name = ""

**def** \_\_init\_\_(self):

self.\_\_maxspeed = 200

self.\_\_name = "Supercar"

**def** drive(self):

**print** 'driving. maxspeed ' + str(self.\_\_maxspeed)

redcar = Car()

redcar.drive()

redcar.\_\_maxspeed = 10 *# will not change variable because its private*

redcar.drive()

# **Method overloading**

In Python you can define a method in such a way that there are multiple ways to call it.

Given a single method or function, we can specify the number of parameters ourself.

Depending on the function definition, it can be called with zero, one, two or more parameters.

This is known as *method overloading.* Not all programming languages support method overloading, but Python does

**class** Human:

**def** sayHello(self, name=None):

**if** name **is** **not** None:

**print** 'Hello ' + name

**else**:

**print** 'Hello '

*# Create instance*

obj = Human()

*# Call the method*

obj.sayHello()

*# Call the method with a parameter*

obj.sayHello('Guido')

**Method overriding**

**Python method overriding** occurs simply defining in the child class  with the same name of a **method** in the parent class.

**Inhertiace Quick View**

**#Single inheritance**

**class Parent():**

**pass**

**class Child(Parent):**

**pass**

**#Mltiple inheritance**

**class Base():**

**pass**

**class child():**

**pass**

**class multiple(base,child):**

**pass**

**#Multi Level inheritance**

**class A():**

**pass**

**class B(A):**

**pass**

**class C(B):**

**#Hybrid Inheritance**

**#** Hybrid Python inheritance is a combination of any two kinds of inheritance.

**class A():**

**pass**

**class B(A):**

**pass**

**class C(A):**

**pass**

**class D(C):**

**pass**

**#Hierarchical Inheritance**

**#More Than one class inherit from the any one of class**

**class A():**

**pass**

**class B(A)**

**pass**

**class C(A):**

**pass**

**Decorator:**

**Adding some extra functionality to existing function**

**def val\_checker(func):**

**def inner(a,b):**

**if b == 0:**

**print ("the val of b cannot be zero")**

**return ""**

**return func(a,b)**

**return inner**

**@val\_checker**

**def division(a,b):**

**return a/b**

**print divison()**

**Iterators:**

An [**iterator**](https://en.wikipedia.org/wiki/Iterator) is an object that can be iterated (looped) upon.,strings, lists, and dictionaries are iterators

An iterator is defined by a class that implements the [**Iterator Protocol**](https://docs.python.org/3/c-api/iter.html). This protocol looks for two methods within the class: \_\_iter\_\_ and \_\_next\_\_.

#### Saving memory space

Iterators don’t compute the value of each item when instantiated. They only compute it when you ask for it. This is known as [lazy evaluation](https://en.wikipedia.org/wiki/Lazy_evaluation).

Lazy evaluation is useful when you have a very large data set to compute.

def check\_prime(number):  
 for divisor in range(2, int(number \*\* 0.5) + 1):  
 if number % divisor == 0:  
 return False  
 return True

Then, we define the iterator class that will include the \_\_iter\_\_ and \_\_next\_\_ methods:

class Primes:  
 def \_\_init\_\_(self, max):  
 self.max = max  
 self.number = 1

def \_\_iter\_\_(self):  
 return self

def \_\_next\_\_(self):  
 self.number += 1  
 if self.number >= self.max:  
 raise StopIteration  
 elif check\_prime(self.number):  
 return self.number  
 else:  
 return self.\_\_next\_\_()

Primes is instantiated with a maximum value. If the next prime is greater or equal than the max, the iterator will raise a StopIteration exception, which ends the iterator.

When we request the next element in the iterator, it will increment number by 1 and check if it’s a prime number. If it’s not, it will call \_\_next\_\_ again until number is prime. Once it is, the iterator returns the number.

By using an iterator, we’re not creating a list of prime numbers in our memory. Instead, we’re generating the next prime number every time we request for it.

Let’s try it out:

primes = Primes(100000000000)

print(primes)

for x in primes:  
 print(x)

---------

<\_\_main\_\_.Primes object at 0x1021834a8>  
2  
3  
5  
7  
11  
...

GENERATORS

Generators are special functions having yeild keyword.

if any function having yeild keyword that functions are called as generators

Generator function will return generator-iterator object.

https://www.youtube.com/watch?v=D54x8uSra3Q

def fib(max):

a,b = 0,1

c = a+b

while True:

if c<max:

yield c

a,b = b,c

else:

break

**Difference B/W List and Touple**

Difference between list and tuple

1. **Literal**

someTuple = (1,2)

someList = [1,2]

1. **Size**

a = tuple(range(1000))

b = list(range(1000))

a.\_\_sizeof\_\_() # 8024

b.\_\_sizeof\_\_() # 9088

Due to the smaller size of a tuple operation, it becomes a bit faster, but not that much to mention about until you have a huge number of elements.

1. **Permitted operations**

b = [1,2]

b[0] = 3 # [3, 2]

a = (1,2)

a[0] = 3 # Error

That also means that you can't delete an element or sort a tuple. However, you could add new element to both list and tuple with the only difference that you will change id of the tuple by adding element

a = (1,2)

b = [1,2]

id(a) # 140230916716520

id(b) # 748527696

a += (3,) # (1, 2, 3)

b += [3] # [1, 2, 3]

id(a) # 140230916878160

id(b) # 748527696

1. **Usage**

As a list is mutable, it can't be used as a key in a dictionary, whereas a tuple can be used.

a = (1,2)

b = [1,2]

c = {a: 1} # OK

c = {b: 1} # Error

**1.Cannot Add Elements to touple**

**2.cannot remove elements from touple**

**3.cannot sort the touple**

**Difference B/W Range and Xange**

**1.Return type:** range will return list and range will return generator object

**2.Memory:** range will take more memory when compared to xrnage bcouse range retrun list and xrange return object

**3.Operations** : As range() returns the list, all the operations that **can** be applied on the list can be used on it. On the other hand, as xrange() returns the xrange object, operations associated to list **cannot** be applied on them, hence a disadvantage.

**4.Speed:** Because of the fact that xrange() evaluates only the generator object containing only the values that are required by lazy evaluation, therefore is**faster** in implementation than range().

**If you want to write code that will run on both Python 2 and Python 3, use range() as the xrange funtion is deprecated in Python 3**

**Difference B/W Python 2.x and Python 3.x**

**1.Division operator: in python 2.x 7/5 will return 1, in python 3.x it will return 1.4**

**2.print : python 2.x print “hello”,python 3.x print (“dsdsd”)**

**3.Unicode :** In Python 2, implicit str type is ASCII. But in Python 3.x implicit str type is Unicode.

4.**Xrange()**:python 3.x x range deprecated

**Python CSV**

Csv have following methods:

Import csv

Csv.reader()

Csv.writer()

Csv.writerow()

Csv.writerows()

Csv.dictwrite()

Csv.writeheader()

**Reading CSV**

import csv

filename = "E:\VINOD\PYTHON\\test.csv"

with open(filename,'r') as csvfile:

csvreader = csv.reader(csvfile)

for i in csvreader:

print (i)

**Output:**

['SNO', 'SHOT', 'ASSET', 'Name']

['1', 'sh', 'ass', 'test']

['1', 'sh', 'ass', 'test']

**Writing CSV – List of Lists**

import csv

fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file

rows = [ ['Nikhil', 'COE', '2', '9.0'],

         ['Sanchit', 'COE', '2', '9.1'],

         ['Aditya', 'IT', '2', '9.3'],

         ['Sagar', 'SE', '1', '9.5'],

         ['Prateek', 'MCE', '3', '7.8'],

         ['Sahil', 'EP', '2', '9.1']]

# name of csv file

filename = "university\_records.csv"

# writing to csv file

with open(filename, 'w') as csvfile:

    # creating a csv writer object

    csvwriter = csv.writer(csvfile)

    # writing the fields

    csvwriter.writerow(fields)

    # writing the data rows

    csvwriter.writerows(rows)

**Writing CSV ----- Writing a dictionary to a CSV file**

|  |
| --- |
| import csv    # my data rows as dictionary objects  mydict =[{'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},           {'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},           {'branch': 'IT', 'cgpa': '9.3', 'name': 'Aditya', 'year': '2'},           {'branch': 'SE', 'cgpa': '9.5', 'name': 'Sagar', 'year': '1'},           {'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},           {'branch': 'EP', 'cgpa': '9.1', 'name': 'Sahil', 'year': '2'}]    # field names  fields = ['name', 'branch', 'year', 'cgpa']    # name of csv file  filename = "university\_records.csv"    # writing to csv file  with open(filename, 'w') as csvfile:      # creating a csv dict writer object      writer = csv.DictWriter(csvfile, fieldnames = fields)        # writing headers (field names)      writer.writeheader()        # writing data rows      writer.writerows(mydict) |

**Diff b/w WITH and OPEN In Files**

1. With will provide you Better Syntax and Exception Handling
2. It will automatically Cloase The File.

**File With WITH OPEN**

**w**ith open(filename,'r') as file:

file.read()

#No Need To Coase The File

**File With Without WITH OPEN**

File = open(filename,’r’)

File.read()

File.close() #important to close the file becasue of security issues abd memeory issues

**Reading Text Files:**

myFile = open(filename,’r’)

d = myfile.read()

print (d) # it will print total file content in string format

ex :

hello world

hello world

hello world

myFile = open(filename,’r’)

d = myfile.readlines()

print (d) # it will return a list having each row as one element in list

ex:

[hello world, hello world, hello world, hello world,]

**Writng To Text File**

filename = "E:\VINOD\PYTHON\demo.txt.txt"

myfile = open(filename,'w')

d = myfile.write("hello testing") #it will clear entire file data and writing newly

**Appending Content To Existing File:**

filename = "E:\VINOD\PYTHON\demo.txt.txt"

myfile = open(filename,'a')

d = myfile.write("hello testing \n")

Triangle Programs:

def traingle(n):

for i in range(0,n):

for j in range(0,i+1):

print (j+1,end="")

print ("\n")

output:

1

12

123

1234

12345

def traingle(n):

for i in range(0,n):

for j in range(0,i+1):

print (\*,end="")

print ("\n")

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

def pypart2(n):

k = 2\*n - 2

for i in range(0, n):

for j in range(0, k):

print(end=" ")

k = k - 1

for j in range(0, i+1):

print("\* ", end="")

print("\r")

OUTPUT:

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**Lenear Search:**

search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned

data = [5,3,4,8,2]

def linear\_search(data,key):

c = 0

while c<len(data):

if data[c] == key:

print (c)

print ("found")

return True

else:

c+=1

#Other Way

For I in data:

If i==key:

Print (“Found”)

Else:

Pass

linear\_search(data,8)

**Binary Search:**

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half
5. def binarySearch(arr, l, r, x):
7. while l <= r:
9. mid = l + (r - l)/2;
11. # Check if x is present at mid
12. if arr[mid] == x:
13. return mid
15. # If x is greater, ignore left half
16. elif arr[mid] < x:
17. l = mid + 1
19. # If x is smaller, ignore right half
20. else:
21. r = mid - 1
23. # If we reach here, then the element
24. # was not present
25. return -1

**BUBBLE SORT**

def bubs(data):

n = len(data)

for i in range(n):

for j in range(0,n-i-1):

if arr[j]>arr[j+1]:

arr[j],arr[j+1] = arr[j+1],arr[j]

return data

**Merge Sort**

def ms(data):

mid = len(data)//2

lh = data[:mid]

rh = data[mid:]

ms(lh)

ms(rh)

i,j,k = 0

while i<len(lh) and j < len(rh):

if lh[i]<rh[j]:

data[k] = lh[i]

i=i+1

else:

data[k] = rh[j]

j=j+1

k=k+1

while i<len(lh):

data[k] = lh[i]

k = k+1

i=i+1

while j<len(rh):

data[k] = rh[i]

k = k+1

i=i+1

**INSERTION SORT**

def IS(data):

n = len(data)

for i in range(1,n):

key = data[i]

j = i-1

while j>=0 and k<arr[j]:

arr[j+1] = arr[j]

j-=1

arr[j+1] = key