**Introduction**

Object oriented programming is a programming paradigm.

Programming paradigm is often misinterpreted with the word programming language

But the two words have a completely different meaning.

A programming language is a design specification of the syntax and semantic rules to be followed while writing a code to accomplish a specific task.

Programming paradigm refers to the style or way in which we write our program.

Python is a universal tool for both **object and procedural programming**.

The Procedure Oriented Programming(POP) mainly focuses on procedures. A procedure is a set of instructions used to accomplish a specific task. It can be routines, sub-routines, functions etc. Some examples of the procedure oriented programming languages are C, Fortran, Cobol, Algol etc. Procedure oriented programming is a dominant approach of software development for decades and it is still in use today for developing projects that are usually not very complex and large.

Object Oriented Programming(OOP) is a style of programming in which the main focus is on the data and the operations that manipulate it. Data are organized into classes and methods. The existence of some information is the existence of an instance of the class which is known as **objects** of the class. The objects are used to interact with real world entities. OOPs is mainly useful to develop big and complex projects carried out by large teams consisting of many developers. Some of the Object Oriented Programming languages are Java,C#,C++,Python etc.

In Object Oriented approach both the data and the code are grouped together into classes. Classes are like receipes and objects are the useful products developed from those receipes.

The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

For example a cookie may be a product developed from the receipe illustrated in a book. This receipe may be inadequate in order to develop a butter cookie. So a new receipe(class) can be designed inheriting properties from the existing receipe(cookie) and add additional steps as required.

Each cookie on the plate is an object with its own set of attributes and functions. Now let’s understand the concept of classes and objects in more detail.

So far you have learned about Python's core data types: strings, numbers, lists, tuples, and dictionaries which are all predefined classes. The last major user defined data structure are classes.

**Objects**

Python supports different kinds of data .The data can be integer, floating point number, string, lists, dictionaries, tuples etc.

Each data is considered as an object and each object has:

1. A name that uniquely identifies it
2. A set of individual properties which make it original, unique or outstanding

An object is an instance of a type.

1. A set of abilities, called methods to perform specific activities, able to change the object itself, or some of the other objects.

For example: the number **157** is an instance of type/class **int**

“**hello**” is an instance of type **string**

There is a hint. Whenever you describe an object you use

1. **a noun** – to define the object's name
2. **an adjective** –to define the object's property
3. **a verb** - to define the object's activity

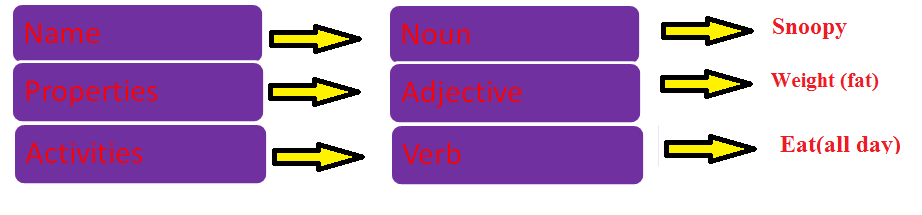
Let’s consider a real world example

**Snoopy is a fat dog which eats all day**

Object name = Snoopy

Class = Dog  
Property = Weight (fat)  
Activity = Eat(all day)  
**Snoopy is a fat dog which eats all day**

**Base Class Dog**

****

**Classes**

A language has only a few types of pre defined classes namely strings, numbers, lists, tuples, sets and dictionaries. It can not provide all possible types of classes we may want to have. For example in real life we may want to create a type called as employee, person, table, computer, dog etc. A language should provide a mechanism to make our own type. This can be achieved through a feature called as class. Classes allow you to define the information and behavior that characterize anything you want to model in your program i.e a class by itself is a **type and implementation**. A **variable** of the class type is called an **object**.

A class is a model or a blueprint of a very specific part of reality, reflecting properties and activities found in the real world.

A class specifies the set of instance variables and methods that are “bundled together” for defining a type of object.

In Python, almost everything is an object - strings, numbers, booleans, functions, and modules are all examples of objects. We design new kinds of objects with classes. A class is a kind of blue print for an object. The class describes what state and behavior should make an object but it is not the object itself.For example, strings in Python are object instances of the built-in String class.

Now let us understand with an example how to create our own class(type) .In order to create our own class we use a reserved keyword called as **class**. A class statement defines a new type and gives the type a name. Following the class keyword is the name of the class . The name of the class should be followed by a colon.

The simplest form of class definition looks like this

**class ClassName:**

**<statement-1>**

**.**

**.**

**.**

**<statement-N>**

The class below has neither properties nor activities.

**Example 1:**

**class animal:**

**pass**

**print(animal,type(animal))**

In the above example we create a class called as animal. Here class is the keyword used to create a new class(type) called as animal. Since at this point I have no attributes or methods defined we use the keyword **pass**.

The next thing to be discussed is what is the significance of **pass** in the program. In  Python,  pass is a null statement. The interpreter does not ignore a pass statement, but nothing happens and the statement results into no operation. The pass statement is useful when we don't write the implementation of a function but you want to implement it in the future.

**Output:**

**<class '\_\_main\_\_.animal'><class 'type'>**

So animal is a type of package \_\_main\_\_.animal

The next question to be answered is what is **\_\_main\_\_ package**?

Since there is no main function in python ,when the command to run a python program is given to the interpreter ,it will define few special variables.One of the special variable is \_\_name\_\_ .If the source file is executed as the main program,the interpreter sets the \_\_name\_\_ variable to have a value \_\_main\_\_. If the file is being imported from another module then it will be set to module’s name.

A class is defined by its attributes .The class attributes are defined in an indented code block

**Example 2:**

**# A class can have attributes(felds or variables)**

**class Animal:**

**a="dog"**

**print(Animal.a)**

**Output:**

dog

The attribute a with the value “dog” belongs to Animal and is accessed using the class name.

**Example 3:**

**# A class can have functions(behaviour)**

**class Animal:**

**def sound():**

**print("barking")**

**Animal.sound()**

**Output:**

**Output:**

**barking**

The function(behaviour) sound belongs to Animal and is invoked using the class name.

The class you define has nothing to do with the object.The existence of a class does not mean that any of the compatible objects will automatically be created. The class itself isn't able to create an object - you have to create it yourself, and Python allows you to do this.

**Example 4:**

class A:

a=10

def display():

print("This is a class behaviour")

print(A.a) ## display attribute value

A.display() ## display behaviour of the class invoked by classname

**Output:**

10

This is a class behaviour

Classes support two kinds of operations:

**Instantiation:**

Creating a variables of a given type(animal)is called as instantiation. of a class Animal. In order to create an object of a specific class , you need to call a class as if it was a function.For example the statement a= Animal() will create an instance or an object of class Animal.

**Example 5:**

**class Animal:**

**pass**

**a=Animal()**

**print(a)**

In the above example an object ‘a’ is created by calling the class Animal( ). The newly created object is provided with everything the class brings as this class is completely empty, the object is empty, too.

**Output:**

**<\_\_main\_\_.Animal object at 0x0043F750>**

Hence by looking at the output we can make out that an object of class Animal has got created at a particular location, 0x0043F750 in this case .

**Attribute References**

Attribute references use dot notation to access attributes associated with the class. For example, a.name refers to the method member associated with the instances of type Animal.

**class Animal:**

**name='dog'**

**a=Animal()**

**print(a.name)**

**Output:**

**dog**

**Constructor**

When an object is created, a special function of the class is called for initializing. This method is called a constructor. The name of the constructor in Python is \_\_init\_\_.

If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

**“The constructor in Python is used to define the attributes of an instance and assign values to them”**

**Example 6:**

**class Sample5:**

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

**print("constructor called")**

**a = Sample5()**

**Output**

**TypeError: \_\_init\_\_() takes 0 positional arguments but 1 was given**

As we see in the output, The constructor is required to have the a parameter called **self** and it is set automatically. It might take more parameters than just self, but self has to always be the first parameter in the function definition. If the latter is considered, then the way in which the class name is used to create the object must get reflected in the  \_\_init\_\_ definition.

**Self** is a parameter that refers to the object that is used to access the class attributes and methods. It can have any other name, self is just a good naming convention, to indicate that the object it**self** is invoking the methods.

The constructor can be used to set up the object, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed.

**a = Sample5() conceptually becomes Sample5.\_\_init\_\_(a)**

The instance is passed as the frst argument to the constructor (a.k.a **ctor**).

Even though the argument is implicit, in Python, we require an explicit parameter.

This can be given any name - is normally called self.

**Example 7:**

**class Sample6:**

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

**print("constructor called")**

**print('self : ', self)**

**a = Sample6()**

**print('a : ', a)**

**Output:**

**constructor called**

**self : <\_\_main\_\_.Sample6 object at 0x0069F730>**

**a : <\_\_main\_\_.Sample6 object at 0x0069F730>**

Both the outputs have the same value indicating that self and object **a** refer to the same object.

\_\_init\_\_ () method is called every time an object is instantiated .This can be illustrated with the example below

**Example 8:** Python code to count number of objects created

**class Sample7:**

**count=0**

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

**print("constructor called")**

**Sample7.count=Sample7.count+1**

**a =Sample7()**

**b=Sample7()**

**c=Sample7()**

**print(Sample7.count)**

**Output:**

**constructor called**

**constructor called**

**constructor called**

**3**

In the above example 3 objects have been created of class Sample7 ,hence the constructor is called three times. This indicates that whenever an object is created the constructor is invoked implicitly.

Now let us understand how we can specify the values of instance attributes through the constructor. The following constructor includes the name and age parameters, other than the self parameter.

**Example 9:**

class Person:

age=20

name="Shaun"

def display(self):

print(self.age,self.name)

p = Person()

p.display()

**Output:**

20 Shaun

**Types Of Constructors**

There are 2 types of constructors namely

**Example 10:Non-parameterized constructors**

#When no parameters are passed for invoking constructor

class Person:

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

print("without parameters")

p= Person()

**Output:**

without parameters

**Example 11:Parameterized constructors**

**#**Parameters are passed for invoking the constructor

class Person:

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

self.name = name

self.age = age

def display(self):

print(self.name,self.age)

p = Person("joe",30) # value of self will be implicitly assigned

p.display()

**Output:**

joe 30

**Class and Instance Variables**

**Instance variables** are for data unique to each instance and **class variables** are for attributes and methods shared by all instances of the class.

**Example 12:**

**class Animal:**

**kind = 'dog' # class variable shared by all instances**

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

**self.name = name # instance variable unique to each instance**

**a= Animal('pinky')**

**b = Animal('snow')**

**print( a.kind ) # shared by all animals**

**print( b.kind ) # shared by all animals**

**print( a.name) # unique to a**

**print( b.name ) # unique to b**

Output:

dog

dog

pinky

snow

Shared data can have possibly surprising effects with [mutable](https://docs.python.org/3/glossary.html#term-mutable) objects such as lists and dictionaries. For example, the L1 list in the following code should not be used as a class variable because just a single list would be shared by all Animal instances.

This can be explained with the following example given below.

**Example 13:**

class Animal:

L1=[]

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

self.name = name

def add(self,activity):

self.L1.append(activity)

a=Animal('pinky')

b=Animal('snow')

a.add('roll over')

b.add('play dead')

print( a.L1 )

print( b.L1 )

**Output:**

['roll over', 'play dead']

['roll over', 'play dead']

**Correct design of the class should use an instance variable instead**

class Animal:

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

self.name = name

self.L1 = []

def add(self,activity):

self.L1.append(activity)

a=Animal('pinky')

b=Animal('snow')

a.add('roll over')

b.add('play dead')

print( a.L1 )

print( b.L1 )

**Output:**

['roll over']

['play dead']