**Class**

Python is an object-oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A class is a user-defined blueprint or prototype from which objects are created. A Class is like an object constructor, or a "blueprint" for creating objects.

Classes provide a means of bundling data and functionality together

 Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

**Class Definition Syntax:**

class ClassName:

# Statement

**Object Definition Syntax:**

obj = ClassName()

print(obj.atrr)

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

* Classes are created by keyword class.
* Attributes are the variables that belong to a class.
* Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

**Object**

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.

An object consists of :

* **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
* **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
* **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

**Declaring Objects (Also called instantiating a class)**

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

**class** Dog:

    # A simple class

    # attribute

    attr1 **=** "mammal"

    attr2 **=** "dog"

    # A sample method

**def** fun(self):

**print**("I'm a", self.attr1)

        print("I'm a", self.attr2)

# Driver code

# Object instantiation

Rodger **=** Dog()

# Accessing class attributes

# and method through objects

print(Rodger.attr1)

Rodger.fun()

Output

mammal

I'm a mammal

I'm a dog

**\_\_init\_\_ method /Constructor**

Constructors are used to initializing the object’s state. Like methods, a constructor also contains a collection of statements (i.e., instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Python supports a special type of method called constructor for initializing the instance variable of a class.

A class constructor, if defined is called whenever a program creates an object of that class.

A constructor is called only once at the time of creating an instance.

If two instances are created for a class, the constructor will be called once for each instance.

|  |
| --- |
| # A Sample class with init method  **class** Person:        # init method or constructor  **def** \_\_init\_\_(self, name):          self.name **=** name        # Sample Method  **def** say\_hi(self):          print('Hello, my name is', self.name)      p **=** Person('Nikhil')  p.say\_hi() |

Output

Hello, my name is Nikhil

**Self**

self represents the instance of the class. By using the “self”  we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.

If we have a method that takes no arguments, then we still have to have one argument.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2)

# Write Python3 code here

class car():

# init method or constructor

def \_\_init\_\_(self, model, color):

self.model = model

self.color = color

def show(self):

print("Model is", self.model )

print("color is", self.color )

# both objects have different self which

# contain their attributes

audi = car("audi a4", "blue")

ferrari = car("ferrari 488", "green")

audi.show() # same output as car.show(audi)

ferrari.show() # same output as car.show(ferrari)

# Behind the scene, in every instance method

# call, python sends the instances also with

# that method call like car.show(audi)

**Difference between method and function**

**Python Method**

1. Method is called by its name, but it is **associated to an object** (dependent).
2. A method definition always includes **‘self’**as its first parameter.
3. A method is **implicitly passed the object** on which it is invoked.
4. It **may or may not return any data.**
5. A method **can operate on the data (instance variables) that is contained by the corresponding class**

**User defined**

**class** ABC :

**def** method\_abc (self):

        print("I am **in** method\_abc of ABC **class**. ")

class\_ref **=** ABC() # object of ABC class

class\_ref.method\_abc()

**Inbuilt method**

**import** math

ceil\_val **=** math.ceil(15.25)

print( "Ceiling value of 15.25 **is** : ", ceil\_val)

**Python Functions**

1. function is block of code that is also **called by its name**. (independent)
2. The function can have different parameters or may not have any at all. If **any data (parameters)** are passed, they are **passed explicitly**.
3. It **may or may not return any data.**
4. Function does not deal with Class and its instance concept.

|  |
| --- |
| **User defined**  **def** Subtract (a, b):  **return** (a**-**b)    **print**( Subtract(10, 12) ) # prints -2    **print**( Subtract(15, 6) ) # prints 9 |

**Built in**

s **=** sum([5, 15, 2])

**print**( s ) # prints 22

mx **=** max(15, 6)

print( mx ) # prints 15

*Functions can be called****only by its name****, as it is defined independently. But methods c****an’t be called by its name****only, we need to invoke the class by a reference of that class in which it is defined,*

**Types of variables in class**

There are 2 types of variables in class

1. Instance variable:

variables that are assigned values inside methods are instance variables.

Variables defined here are belongs to instance namespace

Instance variables are for data, unique to each instance.

Instance variables are variables whose value is assigned inside a constructor or method with self.

1. Class variable (static variable)

All variables which are assigned a value in the class declaration are class variables.

Variables defined here are belongs to class namespace.

class variables are for attributes and methods shared by all instances of the class.

class variables are variables whose value is assigned in the class.

Example

*#instance variable*

*class car:*

*wheels=4*

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

*self.milege=10*

*self.company="BMW"*

*c1=car()*

*c2=car()*

*#we can chnge instance varibale value here*

*c1.milege=12*

*c2.company="skoda"*

*#suppose u want to chnge class variable value*

*car.wheels=6*

*print(c1.milege,c1.company,c1.wheels)*

*print(c2.milege,c2.company,c2.wheels)*

**Types of Methods**

Python offers three types of methods namely instance, class and static methods

1. **Instance Method**
2. They are most widely used methods.
3. The purpose of instance methods is to set or get details about instances (objects)
4. Instance method receives the instance of the class, self as the first argument and points to the instance of our class.
5. However it can take any number of arguments
6. Using the self parameter, we can access the other attributes and methods on the same object and can change the object state
7. Also, using the self.\_\_class\_\_ attribute, we can access the class attributes, and can change the class state as well
8. **instance methods gives us control of changing the object as well as the class state.**

*#instance method example*

*class My\_class:*

*def instance\_method(self):*

*return "This is an instance method."*

*obj = My\_class()*

*obj.instance\_method()*

*Explanation*

When the instance method is called, Python replaces the **self** argument with the instance object, **obj**. That is why we should add one default parameter while defining the instance methods. Notice that when **instance\_method()** is called, you don’t have to pass self.  Python does this for you.

Along with the default parameter self, you can add other parameters of your choice as well:

***Note****that the \_\_init\_\_() method is a special type of method known as a****constructor****. This method is called when an object is created from the class and it allows the class to initialize the attributes of a class.*

*#instance method with \_\_init\_\_()*

*class My\_class:*

*def \_\_init\_\_(self,a,b):*

*self.a=a*

*self.b=b*

*def instance\_method(self):*

*print("This is an instance method with constructor in it",self.a,self.b)*

*obj = My\_class(10,20)*

*obj.instance\_method()*

#instance method self.\_\_class\_\_.\_\_name\_\_

#it returns Class name

*class My\_class:*

*def \_\_init\_\_(self,a,b):*

*self.a=a*

*self.b=b*

*def instance\_method(self):*

*print("This is an instance method using self\_\_class\_\_.\_\_name\_\_ :",self.\_\_class\_\_.\_\_name\_\_)*

*obj = My\_class(10,20)*

*obj.instance\_method()*

With the help of the “self” keyword- self.a and self.b, we have accessed the variables present in the \_\_init\_\_() method of the same class.

Along with the objects of a class, an instance method can access the class itself with the help of **self.\_\_class\_\_**attribute

The self.\_\_class\_\_.\_\_name\_\_ attribute returns the name of the class to which class instance(self) is related.

1. **Class Method**
2. The purpose of the class methods is to set or get the details (status) of the class.
3. They can’t access or modify specific instance data.
4. They are bound to the class instead of their objects.
5. In order to define a class method, you have to specify that it is a class method with the help of the @classmethod decorator
6. Class methods also take one default parameter- **cls,**which points to the class. Again, this not mandatory to name the default parameter “cls”. But it is always better to go with the conventions

Example

*#class method*

*class My\_class:*

*@classmethod*

*def class\_method(cls):*

*return "This is a class method."*

*#obj = My\_class()*

*#obj.class\_method()*

*#We can access the class methods with the help of a class instance/object.*

*#But we can access the class methods directly without creating an instance or object of the class.*

*My\_class.class\_method()*

**When to use class methods?**

**The most common use of the class methods is for creating factory methods. Factory methods are those methods that return a class object (like a constructor) for different use cases**

*Example*

*#when to use classmethods*

*from datetime import date*

*class Dog:*

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

*self.name = name*

*self.age = age*

*# a class method to create a Dog object with birth year.*

*#here we can see classmethod is modifying the status of the class by asking for birthyears*

*@classmethod*

*def Year(cls, name, year):*

*return cls(name, date.today().year - year) #returung the constructor cls #equivalent Dog(name, date.today().year – year).*

*Dog1 = Dog('Bruno', 1)*

*Dog1.name , Dog1.age*

*Dog2 = Dog.Year('Dobby', 2017)*

*#Dog2.name , Dog2.age*

1. **Static Method**
2. Static methods cannot access the class data.Actually they do not need to access the class data
3. They are self-sufficient and can work on their own.
4. they are not attached to any class attribute, they cannot get or set the instance state or class state.
5. In order to define a static method,we can use the @staticmethod decorator .
6. Unlike instance methods and class methods, we do not need to pass any special or default parameters.

*Example*

*#static method*

*class My\_class:*

*@staticmethod*

*def static\_method():*

*return "This is a static method."*

*#obj = My\_class()*

*#obj.static\_method()*

*#And we can call static methods directly, without creating an object/instance of the class:*

*My\_class.static\_method()*

**When to use static methods?**

**They are used for creating utility functions. For accomplishing routine programming tasks, we use utility functions.**

**Example:**

*#when to use static method*

*class Calculator:*

*@staticmethod*

*def add(x, y):*

*return x + y*

*print('The sum is:', Calculator.add(15, 10))*

*#You can see that the use-case of this static method is very clear,*

*#adding the two numbers given as parameters. Whenever you have to add two numbers,*

*#you can call this method directly without worrying about object construction.*

***In short***

* ***An instance method knows its instance (and from that, it’s class)***
* ***A class method knows its class***
* ***A static method doesn’t know its class or instance***