Python Classes and Objects

Last Updated : 14 Jul, 2024

A class is a user-defined blueprint or prototype from which objects are created. 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 to maintain its state. Class instances can also have methods (defined by their class) for modifying their state.

To understand the need for creating a class and object in Python let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Syntax: Class Definition

```
class ClassName:
    # Statement
```

Syntax: Object Definition

```
obj = ClassName()
print(obj.atrr)
```

The 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.

Some points on Python class:

- 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.: My class.Myattribute

Creating a Python Class

Here, the class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

Python3

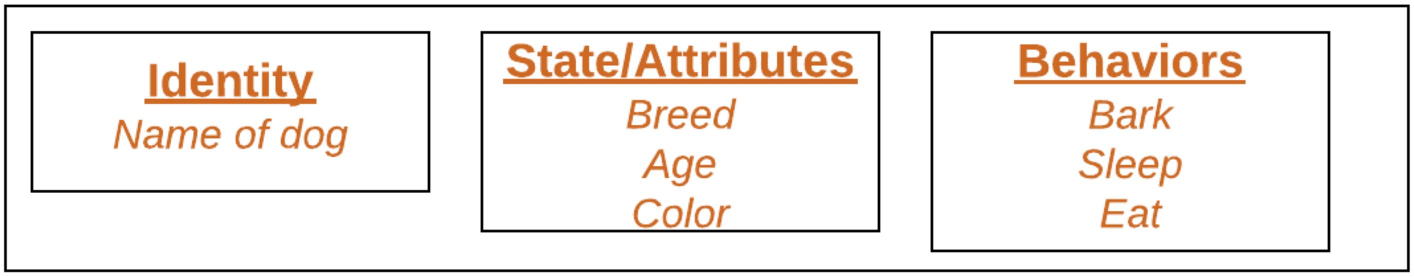
```
class Dog:
    sound = "bark"
```

Object of Python Class

In Python programming 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*. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of:

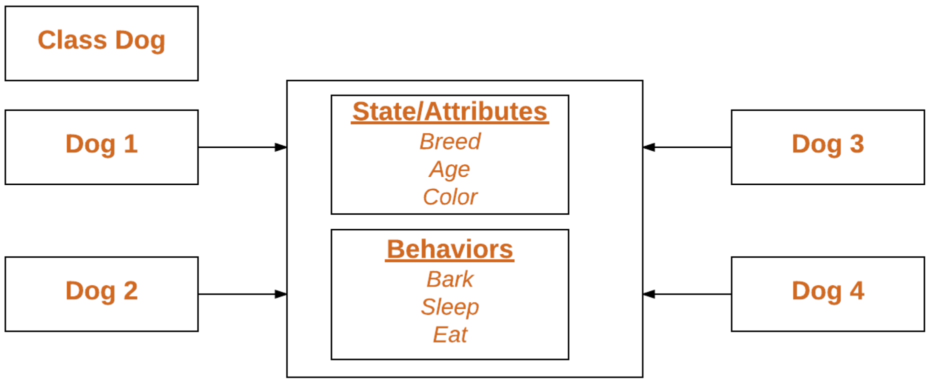
- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one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



Declaring Class 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.

Example:



Example of Python Class and object

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

Python3

```
# Python3 program to
# demonstrate instantiating
# a class
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
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

Explanation :

In this example, we are creating a Dog class and we have created two class variables **attr1 and attr2**. We have created a method named **fun()** which returns the string "I'm a, {attr1}" and I'm a, {attr2}. We have created an object of the Dog class and we are printing at the **attr1** of the object. Finally, we are calling the **fun()** function.

Self Parameter

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) – this is all the special self is about.

Python3

```
class GFG:
    def __init__(self, name, company):
        self.name = name
        self.company = company

    def show(self):
        print("Hello my name is " + self.name+" and I" +
              " work in "+self.company+".")
```

```
obj = GFG("John", "GeeksForGeeks")
obj.show()
```

The Self Parameter does not call it to be Self, You can use any other name instead of it. Here we change the self to the word someone and the output will be the same.

Python3

```
class GFG:
    def __init__(somename, name, company):
        somename.name = name
        somename.company = company

    def show(somename):
        print("Hello my name is " + somename.name +
              " and I work in "+somename.company+".")


obj = GFG("John", "GeeksForGeeks")
obj.show()
```

Output: Output for both of the codes will be the same.

```
Hello my name is John and I work in GeeksForGeeks.
```

Explanation:

In this example, we are creating a GFG class and we have created the **name, and company** instance variables in the constructor. We have created a method named **say_hi()** which returns the string "Hello my name is " + {name} +" and I work in "+ {company}+".".We have created a person class object and we passing the name **John and Company** GeeksForGeeks to the instance variable. Finally, we are calling the **show()** of the class.

Pass Statement

The program's execution is unaffected by the **pass** statement's inaction. It merely permits the program to skip past that section of the code without doing anything. It is frequently employed when the syntactic constraints of Python demand a valid statement but no useful code must be executed.

Python3

```
class MyClass:
    pass
```

__init__() method

The __init__ method is similar to constructors in C++ and Java. 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.

Python3

```
# 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
```

Explanation:

In this example, we are creating a Person class and we have created a **name** instance variable in the constructor. We have created a method named as say_hi() which returns the string "Hello, my name is {name}".We have created a person class object and we pass the name Nikhil to the instance variable. Finally, we are calling the say_hi() of the class.

__str__() method

Python has a particular method called **__str__()**. that is used to define how a **class** object should be represented as a string. It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information. When a class object is used to create a string using the built-in functions print() and str(), the **__str__()** function is automatically used. You can alter how objects of a **class** are represented in strings by defining the **__str__()** method.

Python3

```
class GFG:
    def __init__(self, name, company):
        self.name = name
        self.company = company

    def __str__(self):
        return f"My name is {self.name} and I work in {self.company}."


my_obj = GFG("John", "GeeksForGeeks")
print(my_obj)
```

Output:

```
My name is John and I work in GeeksForGeeks.
```

Explanation:

In this example, We are creating a class named GFG.In the class, we are creating two instance variables **name and company**. In the __str__() method we are returning the **name** instance variable and **company** instance variable. Finally, we are creating the object of GFG class and we are calling the __str__() method.

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. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variables using a constructor.

Python3

```
# Python3 program to show that the variables with a value
# assigned in the class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

# Class for Dog


class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed, color):

        # Instance Variable
        self.breed = breed
        self.color = color


# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

Output:

```
Rodger details:
Rodger is a dog
Breed:  Pug
Color:  brown
Buzo details:
Buzo is a dog
Breed:  Bulldog
Color:  black
Accessing class variable using class name
dog
```

Explanation:

A class named Dog is defined with a class variable animal set to the string "dog". Class variables are shared by all objects of a class and can be accessed using the class name. Dog class has two instance variables **breed and color.** Later we are creating two objects of the **Dog** class and we are printing the value of both objects with a class variable named animal.

Defining instance variables using the normal method:

Python3

```
# Python3 program to show that we can create
# instance variables inside methods

# Class for Dog


class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed):

        # Instance Variable
        self.breed = breed

    # Adds an instance variable
    def setColor(self, color):
        self.color = color

    # Retrieves instance variable
    def getColor(self):
        return self.color


# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

Output:

```
brown
```

Explanation:

In this example, We have defined a class named **Dog** and we have created a class variable animal. We have created an instance variable breed in the **constructor**. The class Dog consists of two methods **setColor** and **getColo**r, they are used for creating and initializing an instance variable and retrieving the value of the instance variable. We have made an object of the **Dog** class and we have set the instance variable value to brown and we are printing the value in the terminal.

Conclusion

understanding Python classes and objects is fundamental for anyone looking to master Python programming. By now, you should have a solid grasp of how classes serve as blueprints for creating objects, and how objects are instances that encapsulate both data and functions. Embracing these concepts can significantly streamline your coding tasks and elevate your projects. If you're eager to expand your Python knowledge further, consider enrolling in our Free Python Course. This course is designed to help you build on the fundamentals and explore more advanced Python features in a structured way.

Python Classes and Objects – FAQs

What are classes and objects in Python?

- *Classes in Python are blueprints for creating objects. They define the attributes (data) and methods (functions) that objects of the class will have.*
- *Objects are instances of classes. They are created from the class blueprint and can have their own unique data while sharing common methods defined in the class.*

What is Python class type?

In Python, a class type refers to the type of object that a class creates. It defines the structure and behavior of objects instantiated from that class.

Why use classes in Python?

Classes in Python provide a way to structure and organize code into reusable components. They facilitate code reusability, modularity, and maintainability by encapsulating data (attributes) and functionality (methods) within objects.

How to define a class in Python?

To define a class in Python, use the class keyword followed by the class name and a colon (:). Inside the class block, define attributes and methods.

```
class MyClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2
    def some_method(self):
        # Method definition
        pass
```

What is an object in OOP?

In Object-Oriented Programming (OOP), an object is a tangible entity that represents a particular instance of a class. It combines data (attributes) and behaviors (methods) specified by the class.

Why do we need classes and objects?

Classes and objects provide a way to model real-world entities and abstract concepts in code. They promote code organization, encapsulation (data hiding), inheritance (code reuse), and polymorphism (method overriding), making complex systems easier to manage and extend.