What is Object Oriented Programming in Python

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects". The object contains both data and code: Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).

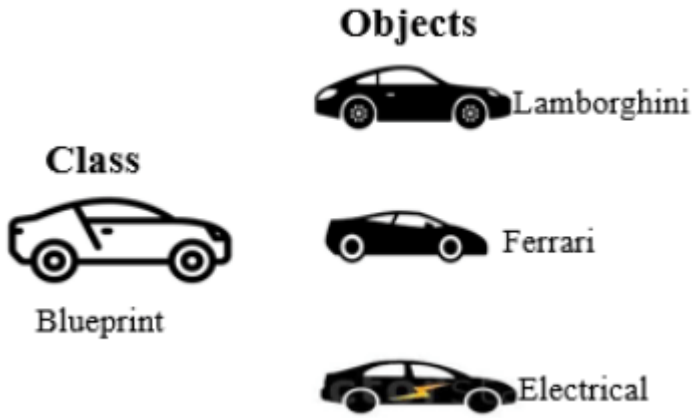
An object-oriented paradigm is to design the program using classes and objects. Python programming language supports different programming approaches like functional programming, modular programming. One of the popular approaches is object-oriented programming (OOP) to solve a programming problem is by creating objects.

Major principles of object-oriented programming system are given below.

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.



How to create a class in Python?

There are some terms which you need to know while working with classes in Python.

1. The “class” keyword
2. The instance attributes
3. The class attributes
4. The “self” keyword
5. The “__init_” method

The “class” keyword

With the class keyword, we can create a Python class as shown in the example below.

```
class BookStore:
    pass
```

What is self?

Python provides the “self” keyword to represent the instance of a class. It works as a handle for accessing the class members such as attributes from the class methods.

Also, please note that it is implicitly the first argument to the **init** method in every Python class.

What is init (constructor) in Python?

The “**init**()” is a unique method associated with every Python class.

Python calls it automatically for every object created from the class. Its purpose is to initialize the class attributes with user-supplied values.

It is commonly known as Constructor in object-oriented programming.

In [1]:
```
class BookStore:
    def __init__(self):
        print("__init__() constructor gets called...")

B1 = BookStore()
```

```
__init__() constructor gets called...
```

The instance attributes

These are object-specific attributes defined as parameters to the **init** method. Each object can have different values for themselves.

In the below example, the “attrib1” and “attrib2” are the instance attributes.

In [2]:
```
class BookStore:
    def __init__(self, attrib1, attrib2):
        self.attrib1 = attrib1
        self.attrib2 = attrib2
```

The class attributes

Unlike the instance attributes which are visible at object-level, the class attributes remain the same for all objects.

Check out the below example to demonstrate the usage of class-level attributes.

In [3]:
```
class BookStore:
    instances = 0
    def __init__(self, attrib1, attrib2):
        self.attrib1 = attrib1
        self.attrib2 = attrib2
        BookStore.instances += 1

b1 = BookStore("", "")
b2 = BookStore("", "")

print("BookStore.instances:", BookStore.instances)
```

```
BookStore.instances: 2
```

In this example, the “instances” is a class-level attribute. You can access it using the class name. It holds the total no. of instances created.

We’ve created two instances of the class . Hence, executing the example should print “2” as the output.

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

In [4]:
```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores **. For example, __doc** gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

In [5]:
```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

```
10
<function Person.greet at 0x0000015E7B205790>
This is a person class
```

How to Define a Class

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a Dog class:

In [6]:
```
class Dog:
    pass
```

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will

eventually go. It allows you to run this code without Python throwing an error.

Note: Python class names are written in CapitalizedWords notation by convention. For example, a class for a specific breed of dog like the Jack Russell Terrier would be written as JackRussellTerrier.

The Dog class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

The properties that all Dog objects must have are defined in a method called .**init**(). Every time a new Dog object is created, .**init**() sets the initial state of the object by assigning the values of the object's properties. That is, .**init**() initializes each new instance of the class.

You can give .**init**() any number of parameters, but the first parameter will always be a variable called self. When a new class instance is created, the instance is automatically passed to the self parameter in .**init**() so that new attributes can be defined on the object.

Let's update the Dog class with an .**init**() method that creates .name and .age attributes:

In [7]:
```
class Dog:
    def __init__(self,name,age):
        self.name = name
        self.age = age
```

Notice that the .**init**() method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the .**init**() method belongs to the Dog class.

In the body of .**init**(), there are two statements using the self variable:

self.name = name creates an attribute called name and assigns to it the value of the name parameter. self.age = age creates an attribute called age and assigns to it the value of the age parameter. Attributes created in .**init**() are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .**init**().

For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

In [8]:
```
class Dog:
    # Class attribute
    species = "Canis familiaris"

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

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a Dog class, let's create some dogs!

Instantiate an Object in Python

In [9]:
```
class Dog:
...     pass
```

This creates a new Dog class with no attributes or methods.

Creating a new object from a class is called instantiating an object. You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses:

In [10]:
```
Dog()
```

Out[10]:
```
<__main__.Dog at 0x15e7b1f62e0>
```

You now have a new Dog object at 0x106702d30. This funny-looking string of letters and numbers is a memory address that indicates where the Dog object is stored in your computer's memory. Note that the address you see on your screen will be different.

In [11]:
```
1 Dog()
```

Out[11]:
```
<__main__.Dog at 0x15e7b23a130>
```

The new Dog instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first Dog object that you instantiated.

To see this another way, type the following:

In [12]:
```
1 a = Dog()
2 b = Dog()
3 a==b
```

Out[12]:
```
False
```

In this code, you create two new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

Class and Instance Attributes

Now create a new Dog class with a class attribute called .species and two instance attributes called .name and .age:

In [13]:
```
1  class Dog:
2
3      species = "Canis familiaris"
4
5      def __init__(self,name,age):
6          self.name = name
7          self.age = age
8
9  MILO = Dog("MILO",4)
10 OLLIE = Dog("OLLIE",3)
11
```

This creates two new Dog instances—one for a 4-year-old dog named milo and one for a 3-year-old dog named ollie.

The Dog class's .**init**() method has three parameters, so why are only two arguments passed to it in the example?

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of .**init**(). This essentially removes the self parameter, so you only need to worry about the name and age parameters.

After you create the Dog instances, you can access their instance attributes using dot notation:

In [14]:
```
1 MILO.name
```

Out[14]:
```
'MILO'
```

In [15]:
```
1 MILO.age
```

Out[15]:
```
4
```

In [16]:
```
1 OLLIE.age
```

Out[16]:
```
3
```

In [17]:
```
1 OLLIE.species
```

Out[17]:
```
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values can be changed dynamically:

In [18]:
```
1 MILO.age = 10
2 MILO.age
```

Out[18]:
```
10
```

In [19]:
```
OLLIE.species ="Felis silvestris"
OLLIE.species
```

Out[19]:
```
'Felis silvestris'
```

Instance Methods

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like .**init**(), an instance method's first parameter is always self.

In [20]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

.description() returns a string displaying the name and age of the dog. .speak() has one parameter called sound and returns a string containing the dog's name and the sound the dog makes. Save the modified Dog class to a file called dog.py and press F5 to run the program. Then open the interactive window and type the following to see your instance methods in action:

In [21]:
```
miles = Dog("Miles", 4)

miles.description()
'Miles is 4 years old'

miles.speak("Woof Woof")
'Miles says Woof Woof'

miles.speak("Bow Wow")
'Miles says Bow Wow'
```

Out[21]:
```
'Miles says Bow Wow'
```

In the above Dog class, .description() returns a string containing information about the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class. However, .description() isn't the most Pythonic way of doing this.

In [ ]:
```

```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
harry = Person()
```

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

In [22]:
```
class Person:
    " this is a person class"
    age = 30

    def greet(self):
        print("Hi")

 # create a new object of person class
Deepali = Person()
print(Person.greet)

print(Deepali.greet)

# Calling object's greet() method
Deepali.greet()
```

```
<function Person.greet at 0x0000015E7B226700>
<bound method Person.greet of <__main__.Person object at 0x0000015E7B24F070>>
Hi
```

In [ ]:
```
1
```

Python class demo

Given here is an example where we are building a BookStore class and instantiating its object with different values.

Create a BookStore class in Python

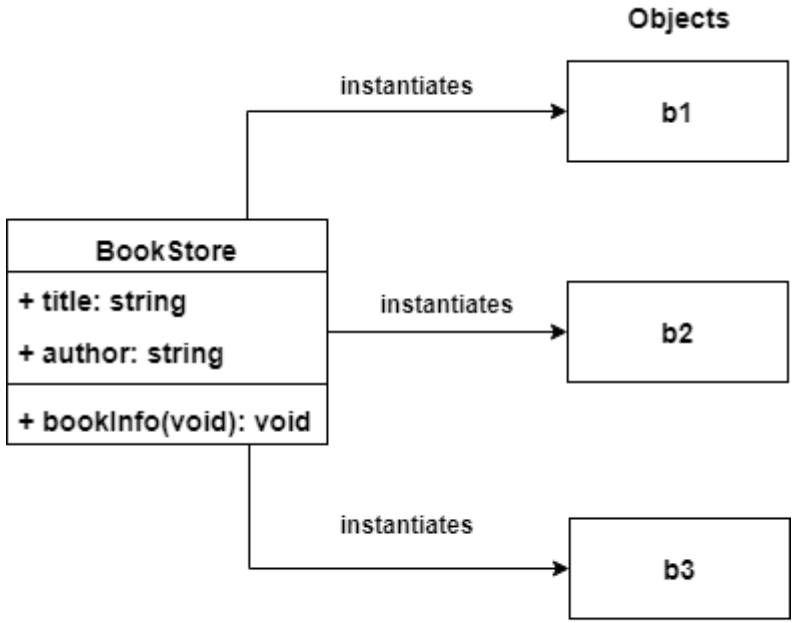
In [23]:
```
class BookStore:
    numOfBooks = 0

    def __init__(self,booktitle,author):
        self.booktitle = booktitle
        self.author = author
        BookStore.numOfBooks +=1

    def bookinfo(self):
        print("Book Title :",self.booktitle)
        print("Book Author :",self.author,"\n")

#creating virtual book store
b1 = BookStore("Think and Grow Rich","Napoleon Hill")
b2 = BookStore("Man's Search for Meaning","Victor E")
b3 = BookStore("Things Fall Apart"," Chinua Achebe")

# calling memeber function for each book
b1.bookinfo()
b2.bookinfo()
b3.bookinfo()


print("BookStore.noOfBooks:",BookStore.numOfBooks)
```

```
Book Title : Think and Grow Rich
Book Author : Napoleon Hill

Book Title : Man's Search for Meaning
Book Author : Victor E

Book Title : Things Fall Apart
Book Author :  Chinua Achebe

BookStore.noOfBooks: 3
```

In this example, we have created three objects of the BookStore class, i.e., b1, b2, and b3. Each of the objects is an instance of the BookStore class.

UML diagram of BookStore class (Unified Modeling Language)-Visual Paradigm

The UML diagram of the above code is as follows.



Python Class and Object Creation

What does inheritance mean in OOP?

Inheritance is the core feature of object-oriented programming which extends the functionality of an existing class by adding new features. You may compare it with real-life situations when a child inherits the property of his parents in addition to adding his own. He may even derive the surname (the second name) from his parents.

What is the purpose of inheritance?

By using the inheritance feature, we can have a new blueprint with old attributes but without making any changes to the original one. We refer to the new class as the derived or child class whereas the old one becomes the base or parent class.

Python Inheritance & OOP Concepts Python Inheritance & OOP Concepts

How to implement inheritance in Python?

You can introduce inheritance by using the following syntax.

```
class ParentClass:
  Parent class attributes
  Parent class methods
class ChildClass(ParentClass):
  Child class attributes
  Child class methods
```

Dog Park Example

Pretend for a moment that you're at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The Dog class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the Dog class in the editor window by adding a .breed attribute:

In [24]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

The instance methods defined earlier are omitted here because they aren't important for this discussion.

Press F5 to save the file. Now you can model the dog park by instantiating a bunch of different dogs in the interactive window:

In [25]:
```
miles = Dog("Miles", 4, "Jack Russell Terrier")
buddy = Dog("Buddy", 9, "Dachshund")
jack = Dog("Jack", 3, "Bulldog")
jim = Dog("Jim", 5, "Bulldog")
```

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like woof, but dachshunds have a higher-pitched bark that sounds more like yap.

Using just the Dog class, you must supply a string for the sound argument of .speak() every time you call it on a Dog instance:

```
buddy.speak("Yap")
#'Buddy says Yap'

jim.speak("Woof")
#'Jim says Woof'


jack.speak("Woof")
#'Jack says Woof'

```

Passing a string to every call to .speak() is repetitive and inconvenient. Moreover, the string representing the sound that each Dog instance makes should be determined by its .breed attribute, but here you have to manually pass the correct string to .speak() every time it's called.

You can simplify the experience of working with the Dog class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for .speak().

Parent Classes vs Child Classes

Let's create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here's the full definition of the Dog class:

In [26]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. Add the following to the dog.py file to create three new child classes of the Dog class:

In [27]:
```
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

In [28]:
```
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

In [29]:
```
miles.species


buddy.name


print(jack)


jim.speak("Woof")

```

```
Jack is 3 years old
```

Out[29]:
```
'Jim says Woof'
```

To determine which class a given object belongs to, you can use the built-in type():

In [30]:
```
type(miles)
```

Out[30]:
```
__main__.JackRussellTerrier
```

What if you want to determine if miles is also an instance of the Dog class? You can do this with the built-in isinstance():

In [31]:
```
isinstance(miles, Dog)
```

Out[31]:
```
True
```

Notice that isinstance() takes two arguments, an object and a class. In the example above, isinstance() checks if miles is an instance of the Dog class and returns True.