Object Oriented Programming Explained Simply for Data Scientists

By Rahul Agarwal

Object-Oriented Programming or OOP can be a tough concept to understand for beginners. And that's mainly because it is not really explained in the right way in a lot of places. Normally a lot of books start by explaining OOP by talking about the three big terms — **Encapsulation, Inheritance and Polymorphism**. But the time the book can explain these topics, anyone who is just starting would already feel lost.

So, I thought of making the concept a little easier for fellow programmers, Data Scientists and Pythonistas. The way I intend to do is by removing all the Jargon and going through some examples. I would start by explaining classes and objects. Then I would explain why classes are important in various situations and how they solve some fundamental problems. In this way, the reader would also be able to understand the three big terms by the end of the post.

In this series of posts named **Python Shorts **, I will explain some simple but very useful constructs provided by Python, some essential tips, and some use cases I come up with regularly in my Data Science work.

This post is about explaining OOP the laymen way.

What are Objects and Classes?

Put simply, everything in Python is an object and classes are a blueprint of objects. So when we write:

```
a = 2
b = "Hello!"
```

We are creating an object a of class `int` holding the value 2 And and object b of class `str` holding the value "Hello!". In a way, these two particular classes are provided to us by default when we use numbers or strings.

Apart from these a lot of us end up working with classes and objects without even realizing it. For example, you are actually using a class when you use any scikit Learn Model.

```
clf = RandomForestClassifier()
clf.fit(X,y)
```

Here your classifier `clf` is an object and fit is a method defined in the class `RandomForestClassifier`

But Why Classes?

So, we use them a lot when we are working with Python. But why really. What is it with classes? I could do the same with functions?

Yes, you can. But classes really provide you with a lot of power compared to functions. To quote an example, the `str` class has a lot of functions defined for the object which we can access just by pressing tab. One could also write all these functions, but that way, they would not be available to use just by pressing the tab button.

This property of classes is called **encapsulation.**

From Wikipedia — ***encapsulation*** *refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.*

So here the `str` class bundles the data("Hello!") with all the methods that would operate on our data. I would explain the second part of that statement by the end of the post. In the same way, the `RandomForestClassifier` class bundles all the classifier methods(`fit` , `predict` etc.)

Apart from this, Class usage can also help us to make the code much more modular and easy to maintain. So say we were to create a library like Scikit-Learn. We need to create many models, and each model will have a `fit` and `predict` method. If we don't use classes, we will end up with a lot of functions for each of our different models like:

```
RFCFit
RFCPredict
SVCFit
SVCPredict
LRFit
LRPredict

and so on.
```

This sort of a code structure is just a nightmare to work with, and hence Scikit-Learn defines each of the models as a class having the `fit` and `predict` methods.

Creating a Class

So, now we understand why to use classes and how they are so important, how do we really go about using them? So, creating a class is pretty simple. Below is a boilerplate code for any class you will end up writing:

```
class myClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def somefunc(self, arg1, arg2):
        #SOME CODE HERE
```

We see a lot of new keywords here. The main ones are `class` , `__init__` and `self` . So what are these? Again, it is easily explained by some example.

Suppose you are working at a bank that has many accounts. We can create a class named `Account` that would be used to work with any account. For example, below I create an elementary toy class `Account` which stores data for a user — namely `account_name` and `balance` . It also provides us with two methods to `deposit` / `withdraw` money to/from the bank account. Do read through it. It follows the same structure as the code above.

```
class Account:
    def __init__(self, account_name, balance=0):
        self.account_name = account_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self,amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Cannot Withdraw amounts as no funds!!!")
```

We can create an account with a name Rahul and having an amount of 100 using:

```
myAccount = Account("Rahul",100)
```

We can access the data for this account using:

```
myAccount.balance
```

```
100
```

```
myAccount.account_name
```

```
'Rahul'
```

But, how are these attributes balance and account_name already set to 100, and "Rahul" respectively? We never did call the `__init__` method, so why did the object gets these attribute? The answer here is that `__init__` is a **magic method**(There are a lot of other magic methods which I would expand on in my next post on Magic Methods), which gets run whenever we create the object. So when we create myAccount , it automatically also runs the function `__init__`

So now we understand `__init__` , let us try to deposit some money into our account. We can do this by:

```
myAccount.deposit(100)
```

```
myAccount.balance
```

```
200
```

And our `balance` rose to 200. But did you notice that our function deposit needed two arguments namely `self` and `amount` , yet we only provided one, and still, it works.

So, what is this `self` ? The way I like to explain `self` is by calling the same function in an albeit different way. Below, I call the same function deposit belonging to the class account and provide it with the myAccount object and the amount. And now the function takes two arguments as it should.

```
Account.deposit(myAccount,100)
```

```
myAccount.balance
```

```
300
```

And our `myAccount` balance increases by 100 as expected. So it is the same function we have called. Now, that could only happen if `self` and `myAccount` are exactly the same object. When I call `myAccount.deposit(100)` Python provides the same object myAccount to the function call as the argument self. And that is why `self.balance` in the function definition really refers to `myAccount.balance`.

But, still, some problems remain

We know how to create classes, but still, there is another important problem that I haven't touched upon yet.

So, suppose you are working with Apple iPhone Division, and you have to create a different Class for each iPhone model. For this simple example, let us say that our iPhone's first version currently does a single thing only — Makes a call and has some memory. We can write the class as:

```
class iPhone:
    def __init__(self, memory, user_id):
         self.memory = memory
         self.mobile_id = user_id
    def call(self, contactNum):
         # Some Implementation Here
```

Now, Apple plans to launch iPhone1 and this iPhone Model introduces a new functionality — The ability to take a pic. One way to do this is to copy-paste the above code and create a new class `iPhone1` like:

```
class iPhone1:
    def __init__(self, memory, user_id):
         self.memory = memory
         self.mobile_id = user_id
         self.pics = []

    def call(self, contactNum):
         # Some Implementation Here

    def click_pic(self):
         # Some Implementation Here
         pic_taken = ...
         self.pics.append(pic_taken)
```

But as you can see that is a lot of unnecessary duplication of code and Python has a solution for removing that code duplication. One good way to write our `iPhone1` class is:

```
Class iPhone1(**iPhone**):
    def __init__(self,**memory,user_id**):
         **super().__init__(memory,user_id)**
         self.pics = []
    def click_pic(self):
         # Some Implementation Here
         pic_taken = ...
         self.pics.append(pic_taken)
```

And that is the concept of inheritance. As per Wikipedia : **Inheritance** is the mechanism of basing an object or class upon another object or class retaining similar implementation. Simply put, `iPhone1` has access to all the variables and methods defined in class `iPhone` now.

In this case, we don't have to do any code duplication as we have inherited(taken) all the methods from our parent class `iPhone` . Thus we don't have to define the call function again. Also, we don't set the mobile_id and memory in the `__init__` function using `super` .

But what is this `super().__init__(memory,user_id)` ?

In real life, your `__init__` functions won't be these nice two-line functions. You would need to define a lot of variables/attributes in your class and copying pasting them for the child class (here `iPhone1`) becomes cumbersome. Thus there exists `super()` . Here `super().__init__()` actually calls the `__init__` method of the parent `iPhone` Class here. So here when the `__init__` function of class `iPhone1` runs it automatically sets the `memory` and `user_id` of the class using the `__init__` function of the parent class.

Where do we see this in ML/DS/DL? Below is how we create a PyTorch model. This model inherits everything from the `nn.Module` class and calls the `__init__` function of that class using the super call.

```
class myNeuralNet(nn.Module):

    def __init__(self):
        **super().__init__()**
        # Define all your Layers Here
        self.lin1 = nn.Linear(784, 30)
        self.lin2 = nn.Linear(30, 10)

    def forward(self, x):
        # Connect the layer Outputs here to define the forward pass
        x = self.lin1(x)
        x = self.lin2(x)
        return x
```

But what is Polymorphism? We are getting better at understanding how classes work so I guess I would try to explain Polymorphism now. Look at the below class.

```
import math

class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        pass

    def getName(self):
        return self.name

class Rectangle(Shape):
    def __init__(self, name, length, breadth):
        super().__init__(name)
        self.length = length
        self.breadth = breadth

    def area(self):
        return self.length*self.breadth

class Square(Rectangle):
    def __init__(self, name, side):
        super().__init__(name,side,side)

class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def area(self):
        return pi*self.radius**2
```

Here we have our base class `Shape` and the other derived classes — `Rectangle` and `Circle`. Also, see how we use multiple levels of inheritance in the `Square` class which is derived from `Rectangle` which in turn is derived from `Shape`. Each of these classes has a function called `area` which is defined as per the shape. So the concept that a function with the same name can do multiple things is made possible through **Polymorphism** in Python. ***In fact, that is the literal meaning of Polymorphism: "Something that takes many forms".*** So here our function area takes multiple forms.

Another way that Polymorphism works with Python is with the `isinstance` method. So using the above class, if we do:

```
mySquare = Square("Square_1",10)
```

```
isinstance(mySquare,Square)
```

```
True
```

```
isinstance(mySquare,Rectangle)
```

```
True
```

```
isinstance(mySquare,Shape)
```

```
True
```

```
isinstance(mySquare,Circle)
```

```
False
```

Thus, the instance type of the object `mySquare` is `Square`, `Rectangle` and `Shape`. And hence the object is polymorphic. This has a lot of good properties. For example, We can create a function that works with an `Shape` object, and it will totally work with any of the derived classes(`Square`, `Circle`, `Rectangle` etc.) by making use of Polymorphism.

```
def getInfo(obj):
    if not isinstance(obj,Shape):
        raise TypeError("obj must have type Shape")
    print(f"Hi my name is {obj.name} and my area is {obj.area()}")

getInfo(mySquare)
```

```
Hi my name is Square_1 and my area is 100
```

Some More Info:

Why do we see function names or attribute names starting with **Single and Double Underscores?** Sometimes we want to make our attributes and functions in classes private and not allow the user to see them. This is a part of **Encapsulation** where we want to "restrict the direct access to some of an object's components". For instance, let's say, we don't want to allow the user to see the memory(RAM) of our iPhone once it is created. In such cases, we create an attribute using underscores in variable names.

So when we create the `iPhone` Class in the below way, you won't be able to access your phone `memory` or the `privatefunc` using Tab in your ipython notebooks because the attribute is made private now using `_`.

```
class iPhone:
    def __init__(self, memory):
        self._memory = memory
    def _privatefunc():
        pass
    def getMemory(self):
        return self._memory
    def setMemory(self,memory):
        self._memory = memory

myphone = iPhone(64000000)
```

```
myphone.
```

```
getMemory
setMemory
```

But you would still be able to change the variable value using(Though not recommended),

```
print(myphone._memory)
```

```
64000000
```

```
myphone._memory = 1
```

```
myphone.getMemory()
```

```
1
```

You would also be able to use the method `_privatefunc` using `myphone._privatefunc()`. If you want to avoid that you can use double underscores in front of the variable name. For example, below the call to `print(myphone.__memory)` throws an error. Also, you are not able to change the internal data of an object by using `myphone.__memory = 1`.

```
class iPhone:
    def __init__(self, memory):
        self.__memory = memory
    def __privatefunc():
        pass
    def getMemory(self):
        return self.__memory
    def setMemory(self,memory):
        self.__memory = memory

myphone = iPhone(64000000)
```

```
# THE BELOW WILL THROW AN ERROR
print(myphone.__memory)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-118-4a4352b32012> in <module>
      1 # THE BELOW WILL THROW AN ERROR
----> 2 print(myphone.__memory)

AttributeError: 'iPhone' object has no attribute '__memory'
```

```
myphone.__memory = 1
```

```
myphone.__memory
```

```
1
```

```
myphone.getMemory()
```

```
64000000
```

```
myphone.setMemory(100)
```

```
myphone.getMemory()
```

```
100
```

But, as you see you can access and modify these `self.__memory` values in your class definition in the function setMemory for instance.

Conclusion

I hope this has been useful for you to understand classes. There is still so much to classes that remain that I would cover in my next post on magic methods. Stay Tuned. Also, to summarize, in this post, we learned about OOP and creating classes along with the various fundamentals of OOP:

- **Encapsulation**: Object contains all the data for itself.
- **Inheritance**: We can create a class hierarchy where methods from parent classes pass on to child classes

- **Polymorphism**: A function takes many forms, or the object might have multiple types.

To end this post, I would be giving an exercise for you to implement as I think it might clear some concepts for you. ***Create a class that lets you manage 3d objects(sphere and cube) with volumes and surface areas***. The basic boilerplate code is given below:

```
import math

class Shape3d:
    def __init__(self, name):
        self.name = name

    def surfaceArea(self):
        pass

    def volume(self):
        pass

    def getName(self):
        return self.name

class Cuboid():
    pass

class Cube():
    pass

class Sphere():
    pass
```