Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

There are many, many tutorials and lessons covering OOP so feel free to Google search other lessons, and I have also put some links to other useful tutorials online at the bottom of this Notebook.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- *Objects*
- *Using the class keyword*
- *Creating class attributes*
- *Creating methods in a class*
- *Learning about Inheritance*
- *Learning about Polymorphism*
- *Learning about Special Methods for classes*

Lets start the lesson by remembering about the Basic Python Objects. For example:

In [13]:
```
lst = [1,2,3]
```

Remember how we could call methods on a list?

In [14]:
```
lst.count(2)
```

Out[14]:
```
1
```

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

Objects

In Python, everything is an object. Remember from previous lectures we can use type() to check the type of object something is:

In [15]:
```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where

the `class` *keyword comes in.*

class

User defined objects are created using the class keyword. The `class` *is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object* `lst` *which was an instance of a list object.*

Let see how we can use `class` *:*

In [16]:
```
# Create a new object type called Sample
class Sample:
    pass

# Instance of Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new ***instance*** *of a Sample class. In other words, we instantiate the Sample class.*

Inside of the class we currently just have pass. But we can define class attributes and methods.

An ***attribute*** *is a characteristic of an object. A* ***method*** *is an operation we can perform with the object.*

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

Attributes

The syntax for creating an attribute is:

`self.attribute = something`

There is a special method called:

`__init__()`

This method is used to initialize the attributes of an object. For example:

In [17]:
```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Lets break down what we have above.The special method

`__init__()`

is called automatically right after the object has been created:

`def __init__(self, breed):`

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

`self.breed = breed`

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

In [18]:
```
sam.breed
```

Out[18]:
```
'Lab'
```

In [19]:
```
frank.breed
```

Out[19]:
```
'Huskie'
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also class object attributes. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute species for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

In [20]:
```
class Dog:

    # Class Object Attribute
    species = 'mammal'

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

In [21]:
```
sam = Dog('Lab','Sam')
```

In [22]:
```
sam.name
```

Out[22]:
```
'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

In [23]:
```
sam.species
```

Out[23]:
```
'mammal'
```

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its self argument.

Let's go through an example of creating a Circle class:

In [24]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

_In the *init* method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object *does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead.*

In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi.

Now let's change the radius and see how that affects our Circle object:

In [25]:
```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Let's see an example by incorporating our previous work on the Dog class:

In [26]:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

In [27]:
```
d = Dog()
```

```
Animal created
Dog created
```

In [28]:
```
d.whoAmI()
```

```
Dog
```

In [29]:
```
d.eat()
```

```
Eating
```

In [30]:
```
d.bark()
```

```
Woof!
```

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

- *It is shown by the eat() method.*

The derived class modifies existing behavior of the base class.

- *shown by the whoAmI() method.*

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, polymorphism refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in. The best way to explain this is by example:

In [31]:
```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

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

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a Dog class and a Cat class, and each has a `.speak()` *method. When called, each object's* `.speak()` *method returns a result unique to the object.*

There a few different ways to demonstrate polymorphism. First, with a for loop:

In [32]:
```
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Another is with functions:

In [33]:
```
def pet_speak(pet):
```

```
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

In [34]:
```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention on
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Real life examples of polymorphism include:

- *opening different file types - different tools are needed to display Word, pdf and Excel files*
- *adding different objects - the + operator performs arithmetic and concatenation*

Special Methods

Finally let's go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example let's create a Book class:

In [35]:
```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
```

```
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author,

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

In [36]:
```
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

*The **init**(), **str**(), **len**() and **del**() methods*

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

Great! After this lecture you should have a basic understanding of how to create your own objects with class in Python. You will be utilizing this heavily in your next milestone project!

Object Oriented Programming

*In the regular section on Object Oriented Programming (**OOP**) we covered:*

- *Using the class keyword to define object classes*
- *Creating class attributes*
- *Creating class methods*
- *Inheritance - where derived classes can inherit attributes and methods from a base class*
- *Polymorphism - where different object classes that share the same method can be called from the same place*
- *Special Methods for classes like `__init__`, `__str__`, `__len__` and `__del__`*

In this section we'll dive deeper into

- *Multiple Inheritance*
- *The `self` keyword*
- *Method Resolution Order (MRO)*

- *Python's built-in `super()` function*

Inheritance Revisited

Recall that with Inheritance, one or more derived classes can inherit attributes and methods from a base class. This reduces duplication, and means that any changes made to the base class will automatically translate to derived classes. As a review:

In [1]:
```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention on
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):
    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):
    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

In this example, the derived classes did not need their own `__init__` methods because the base class `__init__` gets called automatically. However, if you do define an `__init__` in the derived class, this will override the base:

In [2]:
```
class Animal:
    def __init__(self,name,legs):
        self.name = name
        self.legs = legs

class Bear(Animal):
    def __init__(self,name,legs=4,hibernate='yes'):
        self.name = name
        self.legs = legs
        self.hibernate = hibernate
```

This is inefficient - why inherit from Animal if we can't use its constructor? The answer is to call the Animal `__init__` inside our own `__init__`.

In [3]:
```
class Animal:
    def __init__(self,name,legs):
        self.name = name
        self.legs = legs
```

```
class Bear(Animal):
    def __init__(self,name,legs=4,hibernate='yes'):
        Animal.__init__(self,name,legs)
        self.hibernate = hibernate

yogi = Bear('Yogi')
print(yogi.name)
print(yogi.legs)
print(yogi.hibernate)
```

```
Yogi
4
yes
```

Multiple Inheritance

Sometimes it makes sense for a derived class to inherit qualities from two or more base classes. Python allows for this with multiple inheritance.

In [4]:
```
class Car:
    def __init__(self,wheels=4):
        self.wheels = wheels
        # We'll say that all cars, no matter their engine, have four wheels by

class Gasoline(Car):
    def __init__(self,engine='Gasoline',tank_cap=20):
        Car.__init__(self)
        self.engine = engine
        self.tank_cap = tank_cap # represents fuel tank capacity in gallons
        self.tank = 0

    def refuel(self):
        self.tank = self.tank_cap


class Electric(Car):
    def __init__(self,engine='Electric',kWh_cap=60):
        Car.__init__(self)
        self.engine = engine
        self.kWh_cap = kWh_cap # represents battery capacity in kilowatt-hours
        self.kWh = 0

    def recharge(self):
        self.kWh = self.kWh_cap
```

So what happens if we have an object that shares properties of both Gasolines and Electrics? We can create a derived class that inherits from both!

In [5]:
```
class Hybrid(Gasoline, Electric):
    def __init__(self,engine='Hybrid',tank_cap=11,kWh_cap=5):
        Gasoline.__init__(self,engine,tank_cap)
        Electric.__init__(self,engine,kWh_cap)


prius = Hybrid()
print(prius.tank)
print(prius.kWh)
```

```
0
0
```

In [6]:
```
prius.recharge()
print(prius.kWh)
```

```
5
```

Why do we use `self`?

We've seen the word "self" show up in almost every example. What's the deal? The answer is, Python uses `self` to find the right set of attributes and methods to apply to an object. When we say:

prius.recharge()

What really happens is that Python first looks up the class belonging to `prius` (Hybrid), and then passes `prius` to the `Hybrid.recharge()` method.

It's the same as running:

Hybrid.recharge(prius) but shorter and more intuitive!

Method Resolution Order (MRO)

Things get complicated when you have several base classes and levels of inheritance. This is resolved using Method Resolution Order - a formal plan that Python follows when running object methods.

To illustrate, if classes B and C each derive from A, and class D derives from both B and C, which class is "first in line" when a method is called on D?

Consider the following:

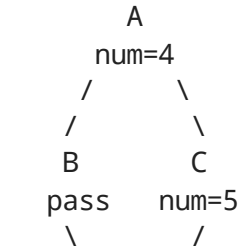
In [7]:
```
class A:
    num = 4

class B(A):
    pass

class C(A):
    num = 5

class D(B,C):
    pass
```

Schematically, the relationship looks like this:



```
      \     /
        D
       pass
```

Here `num` is a class attribute belonging to all four classes. So what happens if we call `D.num`?

In [8]:
```
D.num
```

Out[8]:
```
5
```

*You would think that `D.num` would follow `B` up to `A` and return **4**. Instead, Python obeys the first method in the chain that defines num. The order followed is `[D, B, C, A, object]` where object is Python's base object class.*

In our example, the first class to define and/or override a previously defined `num` is `C`.

`super()`

Python's built-in `super()` function provides a shortcut for calling base classes, because it automatically follows Method Resolution Order.

In its simplest form with single inheritance, `super()` can be used in place of the base class name :

In [9]:
```
class MyBaseClass:
    def __init__(self,x,y):
        self.x = x
        self.y = y

class MyDerivedClass(MyBaseClass):
    def __init__(self,x,y,z):
        super().__init__(x,y)
        self.z = z
```

Note that we don't pass `self` to `super().__init__()` as `super()` handles this automatically.

In a more dynamic form, with multiple inheritance like the "diamond diagram" shown above, `super()` can be used to properly manage method definitions:

In [10]:
```
class A:
    def truth(self):
        return 'All numbers are even'

class B(A):
    pass

class C(A):
    def truth(self):
        return 'Some numbers are even'
```

In [11]:
```
class D(B,C):
    def truth(self,num):
        if num%2 == 0:
            return A.truth(self)
        else:
            return super().truth()

d = D()
d.truth(6)
```

Out[11]:
```
'All numbers are even'
```

In [12]:
```
d.truth(5)
```

Out[12]:
```
'Some numbers are even'
```

In the above example, if we pass an even number to `d.truth()`, we'll believe the `A` version of `.truth()` and run with it. Otherwise, follow the MRO and return the more general case.

Great! Now you should have a much deeper understanding of Object Oriented Programming!

Thank You

CONNECT WITH ME :

LinkedIn GitHub kaggle Medium

PRASADMJADHAV2