INTRODUCTION TO PROGRAMMING
WEEK 6 - BASIC OBJECT ORIENTED PROGRAMMING

OBJECT ORIENTED PROGRAMMING (OOP)

Object oriented programming is a framework that teaches us to encapsulate all execution (or as much as possible) into objects.

(Very) Loosely speaking, we have already been working with objects - strings, integers, lists, dictionaries. All of these objects have certain **properties** (a string's length) and certain **methods** (iteritems() for dictionaries, append() for lists). Each time we create an object, we have to specify what kind of object it is (string, list, dictionary), or in other words, the object's **class**. If I wrote `employees = {}` and `data = {}`, I would expect `employees` and `data` to behave the same since they are both objects of the same type (class), dictionary.

In this chapter, we will look at how to create custom objects and classes (Employee objects, Dog objects, Person objects, etc.)

OOP is quite difficult to teach because its two major parts - Objects and Classes are defined by each other. Classes define how objects are created. Objects are created using classes. To make this easier, I will start with objects.

OBJECTS

The foundational concept behind OOP is the idea of an object. An object is a self-contained entity that can have properties and methods. For instance, can you guess what this code is doing?

```
my_dog = Dog('Scooter', 'Labrador')

print my_dog.name # 'Scooter'
print my_dog.breed # 'Labrador'
print my_dog.Bark() # 'woof, woof!'

my_dog.Rename('Fido')
print my_dog.name # 'Fido'
```

In the code above, I am creating an Dog object, `my_dog`. This Dog, `my_dog`, has certain attributes (properties), including his name(Scooter) and can perform certain actions (methods), including `Bark()` and `Rename()`.

Don't worry if you don't understand all the code above. It will all make sense in a few sections.

Properties

All objects can have properties, or **instance variables**, which are certain attributes particular to that object alone. In our example, `my_dog`, has 2 properties that we know of: `name` and `breed`. To reference an object's property, just use the dot operator and the property name:

```
my_dog.name # 'Scooter'
my_dog.breed # 'Labrador'
```

Methods

All objects can also have methods, or actions that they perform. **Methods are just functions that belong to objects.** Again, using the dot operator, we can reference the object and the action we want to perform.

```
my_dog.Bark() # 'woof, woof!'
my_dog.Whine() # 'wooo...wooo...'
my_dog.Greet() # 'Hi, my name is Scooter and I love you!'
```

Methods can also perform actions on the properties of the objects that call them. For instance:

```
my_dog.Rename('Fido')
print my_dog.name # 'Fido'
```

In the above example, the `Rename()` method will change the value of the property `name` of `my_dog` such that it is now 'Fido' instead of 'Scooter'.

Creating an Object - Instantiation

The process of creating an object is known as **instantiation**. When we created the `my_dog` object, we did this:

```
my_dog = Dog('Scooter', 'Terrier')
```

This line creates a Dog object with the properties 'Scooter' for name and 'Terrier' for breed and assigns it to the variable, `my_dog`.

CLASSES

Classes are templates for creating objects. They specify the properties and methods that all objects of that class can have, and also dictate HOW an object of that class is created.

In our example, we need to create a Dog class that defines how Dog objects are created and what they can actually do.

```
class Dog(object):
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed
    print self.name + ' the Dog has been created.'

  def Bark(self):
    print 'woof, woof!'
  def Greet(self):
    print 'Hi, I am ' + self.name + ' and I LOVE you!'

  def Rename(self, new_name):
    self.name = new_name
    return self.name
```

Let's look at the different parts of the class.

Constructors

A constructor is a method for creating an object of a certain class. It dictates how an object of a certain class needs to be created. In our Dog class, the constructor method is this:

```
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed
    print self.name + ' the Dog has been created.'
```

Notice the syntax here. You can tell it's a function because of the `def`, but the reserved function name, `__init__` indicates that it's a constructor.

In Python, the first parameter of all method definitions is always `self`. This is true even though the method itself is not called with `self` as the first parameter. Why this is so remains a mystery. All the answers I've gotten on the issue have been "Just because...". It's best to treat it as a syntactical formality.

From the constructor's function signature, we can see that it requires 2 parameters, a name, and a breed. (Again, ignore `self` in the function parameters...). Thus, when the function is called, it needs to be called with 2 parameters.

When we instantiate objects of a certain class, we do so by calling the constructor of that class.

In our example, when we instantiate objects of class Dog, we are actually calling the Dog class's constructor.

```
my_dog = Dog('Scooter', 'Terrier')

# Scooter the Dog has been created.
```

Now, let's look inside a constructor to get a better sense of what is actually happening.

```
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed
    print self.name + ' the Dog has been created.'
```

Once the constructor is called, it takes the variables that were passed in as parameters and assigns them to properties of the object it creates.

In a constructor, the `self` keyword always refers to the object that the constructor is creating when it is called.

In our example, `self` is referring to the Dog object assigned to `my_dog`.

All together then, when we do this:

```
my_dog = Dog('Scooter', 'Labrador')
print my_dog.name # 'Scooter'
print my_dog.breed # 'Labrador'
```

We are creating a Dog object by calling the Dog class's constructor method (`__init__`) with the parameters 'Scooter' and 'Labrador'. Inside the constructor method, we are taking those parameters, 'Scooter' and 'Labrador', and assigning them to properties of a new Dog object. Once created, this Dog object is assigned to the variable `my_dog`. Finally, we print out the property values of name and breed in `my_dog` to prove that it worked.

What happens when we try to run code like this?

```
my_dog = Dog()
```

Or like this?

```
my_dog = Dog('Snoopy')
```

Neither of these examples will run because the constructor method of the Dog class MUST be called with 2 parameters, the dog's name and the dog's breed.

Another, more clear way of calling constructors is to also pass in the parameter name when creating an object. This syntax is valid in Python:

```
my_dog = Dog(name='Fido', breed='Labrador')
```

This is flexible for two reasons:

1) It allows us to change the order of the values passed in:

```
my_dog = Dog(name='Fido', breed='Labrador')
my_dog = Dog(breed='Labrador', name='Fido) # these two are the same
```

2) It also allows us to clearly list the values on separate lines like this:

```
my_dog = Dog(name='Fido',
             breed='Labrador')
```

Listing the parameter values on separate lines when calling a class's constructor is often a cleaner way of doing it. It makes no difference to the Python interpreter, but it's easier and more intuitive to read.

Methods

A class can also have other methods other than the constructor. Let’s take a look again at the code for the Dog class and the methods that it has. The first method is the bark() method which looks like this:

```
  def Bark(self):
    print ‘woof, woof!’
```

This is a pretty straight forward method, it just prints a single string, ‘woof, woof!’. Since `my_dog` is a Dog object, it can call this method.

```
my_dog.Bark() # ‘woof, woof!’
```

The second method is a little bit more complicated:

```
  def Greet(self):
    print ‘Hi, I am ’ + self.name + ‘ and I LOVE you!’
```

In this method, it again prints out a string, but the string it prints also references a property of the object that is calling it (self). Here again, the self key word refers to the object that is calling the method.

```
my_dog.Greet() # ‘Hi, I am Scooter and I LOVE you!’
```

The final method, `Rename()`, is the most complicated, it actually changes the property value of an object.

```
  def Rename(self, new_name):
    self.name = new_name
    return self.name
```

Here, we see that the method takes in 1 parameter, new_name, and assigns that parameter to the name property of the object that is calling the method.

```
print my_dog.name # ‘Scooter’
my_dog.Rename(‘Fido’)
print my_dog.name # ‘Fido’
```

Finally, on an interesting side note, Python class methods can also add properties to an object that were not created in the constructor. For instance, in our Dog class, I could add:

```
def AddOwner(self, owner_id):
  self.owner = owner_id
```

This is a perfectly valid assignment even though the constructor did not create the object with an owner property. **This will come useful later in inheritance when subclasses need to add properties that parent classes do not provide.** Python has a certain flexibility about the way properties are assigned to objects. Some consider it a strength, others a weakness.

INHERITANCE

One of the main advantages of object oriented programming is the principle of inheritance. With inheritance, you can create classes that are sub-classes of other classes and thus inherit all their properties and methods. For instance, I can create a class called Person and then create a another class called Asian. When writing the code for the Asian class, I could copy all of the code in Person, but that wouldn't make any sense because an Asian is a kind of person. The Asian class is a subclass of the class Person and should inherit all the Person class's methods and properties.

To make a class a subclass of another, you just enclose the parent class's name in parenthesis after the class declaration for the sub-class.

```
class Person(object):

class Asian(Person):
```

Let's take a look at a more concrete and potentially less offensive example:

```
class Dog(object):
  def __init__(self, name, size):
    self.name = name
    self.size = size

  def Defecate(self):
    print ‘bloop bloop!’

  def Bark(self):
    print ‘woof woof!’

class GermanShepard(Dog):
  def Bark(self):
    print ‘WOOF! WOOF! WOOF!...GRRR’

myDog = Dog(‘Scooter’, ‘Small’)
myDog.Defecate() # ‘bloop bloop!’
myDog.Bark() # ‘woof woof!’

print '\nGerman Shepard Time!\n'

myGerman = GermanShepard(‘Attacker’, ‘Large’)
myGerman.Defecate() # ‘bloop bloop’
myGerman.Bark() # ‘WOOF! WOOF!...GRRR’
```

In this example, we see that `Dog` is the **superclass** or **parent class** of class `GermanShepard`. This is because `GermanShepard` is a type of `Dog`. We can also say that `GermanShepard` is a **subclass** of `Dog`, or that `GermanShepard` **extends** `Dog`. All this language is really inter-changable.

With inheritance, we see that when we call:

```
myGerman.Defecate()
```

This is basically the same as calling the `Defecate()` method of `Dog`. We can reason that all kinds of dogs defecate in the same way, so `GermanShepard` **inherited** the method from `Dog`.

Overriding methods

You might have noticed that the `GermanShepard` object in the example barked differently than the `Dog` object did. Why is this? It is because we **overrode** the `Bark()` method. We need to do this because German Shepards are big dogs and will bark loudly. This is a major advantage to OOP design because it allows full customization of subclasses.

To override a parent class's inherited method, a subclass definition simply needs to declare its own method with the same method name. This will successfully override the method of the parent class.

IN-CLASS EXERCISE

Study the features of the class and sample output below before writing your exercise. The sample output will show you how to create the `Cat` class.

Create a `Cat` class with these features:

`Cat` class

- Properties:
    - name
    - fur color
    - temper (integer)
- Methods:
    - Pur
    - Get Temperament
        - temper >= 5 - 'Angry Cat'
        - 2 =< temper < 5 - 'Normal Cat'
        - temper < 2 - 'Mellow Cat'
- Mellow Out
    - Takes one parameter, an integer
    - Reduces Cat's temper by that integer amount

Then, instantiate a `Cat` object, `my_cat`, such that:

```
my_cat = Cat(name='Garfield',
             fur_color='Orange',
             temper=10)

my_cat.Pur()

print my_cat.GetTemperament()

my_cat.MellowOut(6)

print my_cat.GetTemperament()
```

Will print:

```
Garfield the cat has been created and his fur color is Orange.
purrr...purrr...my name is Garfield
Garfield is a Angry Cat.
Mellowing Garfield out by 6 temper points
Garfield is a Normal Cat.
```

(Hint: Re-use Code from `Dog` class above.)