Object-oriented Python

Introduction to Data Science with Python

What are Objects?

- To a Python developer, objects are a representation of a "real world" thing, such as a car or database table
- Objects have two primary characteristics: data and methods that act on that data
  - Data are referred to as attributes
  - Methods offer ways to view and manipulate the data so the developer doesn't have to do so directly
- Think of the class as the blueprint for building its corresponding object
- Similar to a function, we have to define a class (function) before we can create (invoke) it

Advantages of Object-Oriented Code

- Anything you can do in OO code, you can also do in procedural code, so why use objects?
  - Objects are easier to understand
    - Humans think more in terms of objects so they are more relatable
  - Objects hide complexity
    - Most of today's sophisticated applications would not be possible without OOP
  - Objects are easier to reuse
    - Fewer bugs
    - Encapsulation of business rules and specialized methods
  - Objects are more secure
    - Key values and processes can be protected from misuse

Objects vs. Classes

- Some newbies get this confused but it is pretty easy once you think about it
  - Classes are the blueprint for building an object
  - Objects are the result on an "instantiation" of a particular class
  - Objects are "real", that is, they can be acted on and passed as parameters, classes are virtual so they can't be

Defining Classes in Python

- We use the class statement to define a class
- We instantiate a class by calling the class
- We then say *rover is an object of class Dog*

```
# Simple class declaration
class Dog:
  pass

# Instantiate the class
rover = Dog()
```

Class Attributes

- We use attributes to "describe" our object
- How might you describe a "Dog"
  - Color
  - Breed
  - Hungry
- Set the attributes in the class definition

```
# Class declaration with attributes
class Dog:
  def __init__(self, color, breed, hungry):
       self.color = color
       self.breed = breed
       self.hungry = hungry    # Either true or false


# Instantiate the class
rover = Dog("brown", "dachshund", True)
```

More on the __init__() method and self

- The *self* variable points to the object instance of its class
  - Objects need a reference to themselves in order to be able to modify their various attributes or call their internal methods

- The __init__ method is a *class initializer*
  - In other languages we call it a *constructor*
  - It is called automatically when the object is created and used to initialize attributes or perform special operations

```
class Dog:
  def __init__(self, color, breed, hungry):
        self.color = color
        self.breed = breed
        self.hungry = hungry    # Either true or false

# Instantiate the class
rover = Dog("brown", "dachshund", True)
```

Test Your Knowledge - Ex 7.1

- Now it's your turn to apply what you have learned.

1. Create a class definition for a "WordGame"
2. Give the WordGame class 1 attribute: guesses_allowed
3. Create a WordGame object in code

Inheritance

- Suppose you have a class (e.g. animal) and want to add some specificity
  - You could just create a new class, but it might be nice to be able to reuse the functionality of the existing class
  - Having a base class also allows you to have a single source for most of the implementation
- Inheritance allows one class to take on the attributes and methods of another class (e.g. dog from animal)
  - Inherited classes are easier to implement because much of the functionality is already in place
- In Python 3, by default, all objects are derived from the *object* class

Inheritance

- The easiest way to implement inheritance is to simply call the base class init method
- Supply any other initialization as needed

```
class Animal:
  def __init__(self, name, hungry):
      self.name = name
      self.hungry = hungry    # Either true or false

# Child class inherits from Animal
class Dog(Animal):
  def __init__(self, name, hungry, breed):
      Animal.__init__(self, name, hungry)
      self.breed = breed
```

Test Your Knowledge – Ex 7.2

- Create a new class called TLGame (for Three-Letter Word Game) and inherit from the base class WordGame
  - Have a default number of guesses set to 3
  - Create another parameter for this class constructor called wordlist
    - It will be a list of words

instancemethod vs. classmethod vs. staticmethod

- There are three kinds of methods you can have in a class
  - Instance method
  - Class method
  - Static method

- Each has specific properties you can use for different purposes

Instance Methods

- Use instance methods to get the contents of an instance or perform operations on the attributes of the instance
- The instance method is the traditional method
  - Note this method will always have *self* as the first parameter
  - Instance methods can modify object state

```
class Dog:
  def __init__(self, color, breed, hungry):
      self.color = color
      self.breed = breed
      self.hungry = hungry    # Either true or false

   # Instance method
   def Bark(self):
       print("Yip!")
```

Class Method

- The class method has the @classmethod decorator
- The first parameter to this method is the *cls* parameter which is a pointer to an instance of the class (as distinct from *self* which is a pointer to an object)
- As such, it does not have the ability to alter an object state, but it can modify state that applies to all instances of a class, or class state

```
@classmethod
def HatesBath(cls):
    print("Grr!")
```

Static Method

- Static Method don't require any parameters and can't modify object or class state
  - Note the @staticmethod decorator
- They are useful to combine related methods into a namespace
- You don't have to instantiate the class to call the method

```
class CustomMath():
    @staticmethod
    def sum(a, b):
        return a + b

    @staticmethod
    def square(a):
        return a * a

CustomMath.sum(2, 3)
CustomMath.square(2)
```

Properties

- We could use getters and setters for properties, but it isn't really considered the "Pythonic" way
- Instead we can apply the decorator: @property

```
@property
def IsHungry(self):
    if self.hungry:
        print("Yes")
    else:
        print("No")
```

But wait…there's more

- Object-oriented programming is a deeper subject than what we covered here
  - Attribute encapsulation
  - Getters/Setters
  - Multiple Inheritance
  - Metaclasses
  - Abstract classes

- We won't be using these techniques here but you might want to check them out

Lab 7

- You had a couple of simple exercises, now to the hard part.
- Extend your class to encapsulate the game you built in Lab 6
  - Add properties for each field/variable in the class
  - Convert your functions to static, instance or class methods as appropriate
  - Try your new game