# Of Dice & Dragons, Part 2: Inheritance

Due: Up to Discretion of Mr Whaley

**Objective**: To become accustomed in defining classes and creating objects while also learning about class inheritance.

**Background**: In Python, classes may contain variables (which are often referred to as data types to be more general) that are used to describe the state of the object, and they may also contain methods (which are functions defined within the class definition) that are used to describe the behavior of the object. So, it is safe to say that class definitions may contain only variables, only methods, or both! Recall that classes are programmer-defined, thus object-oriented programming languages provides us with a lot of power in describing the real-world.

As Python is object-oriented, a language feature most often associated with OOP languages is inheritance, which is the ability to define a new class that is a modified version of an existing class. How is this useful, and why even bother with it? A big concern among programmers is the need to write programs in a quick and efficient manner. Thus, the utilization of inheritance becomes useful because it allows the programmer to take a class that has already been defined and extend it by including additional lines of code. This saves time by avoiding producing more code, and in fact makes many programs more compact.

In a class definition, although not explicitly stated, there is the concept of class attributes and instance attributes, which all relate to the scope of the attribute. What is meant by scope is the kind of relationship the attribute has towards both the class definition and to the Python program. For example, class attributes are variables defined inside a class BUT outside of any method and are associated with the class. On the other hand, instance attributes are also defined inside a class, BUT are associated with a particular instance. A good rule of thumb to follow is that, if there is a variable within a method definition inside a class definition, that variable is likely to be an instance attribute. As such, any variables that are declared and initialized within the **__init__()** method are class attributes.

This is important to discuss because, when defining a new class that inherits from an existing class, the new class receives everything already defined from the parent class, from its class and instance attributes right down to the methods associated with the existing class. This sounds a bit like genetic inheritance, right? That's why the existing class is often referred to as the parent class, and the new class is often referred to as the child class. So, if three new classes are created from the same existing class, then there are three children classes associated to a single parent class! An important thing to note about this process of inheritance, utilizing genetic inheritance as a good reference point to build upon from, is that the child may often not inherit the behavior of the parent. As such, methods that are defined in the parent class can be overridden by the programmer in the child class. What this means is that it allows the child class to provide a different behavior than the one expected from the parent class.

## Syntax

The proper syntax to use in inheritance is as follows:

```
# This is the parent class.
Class Foo1:

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


# This is the child class. Note how Foo1, the class to be used as the parent, is enclosed in
# parentheses. This now makes Foo2 the child of Foo1.
class Foo2(Foo1):

        # No need to have to define an __init__() method as Foo2 inherits the __init__()
        # method from its parent, Foo1.

        def do_something(self):
                print(“Hello!”)
```

## Problems

Given the following **Pet** class definition:

```
class Pet:
        """Represents an animal companion a person may have."""

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

        def __str__(self):
                return "%s is a %s" % (self.name, self.species)

        def getName(self):
                return self.name

        def getSpecies(self):
                return self.species
```

Create a **Dog** and **Cat** class from the **Pet** class. Both **Dog** and **Cat** classes will have two methods. Both must have **__init__()** methods with a name parameter. For the **Dog** class, include an additional parameter that is of type Boolean called chases_cats, and a **chaseCats()**

method that returns the Boolean value of the chases_cats variable. Likewise, for the **Cat** class, include an additional parameter that is of type Boolean called hates_dogs, and a **hateDogs()** method that returns the Boolean value of the hates_dogs variable.

****HINT****: When defining both **__init__()** methods in the **Dog** and **Cat** classes, it may be beneficial to invoke the **__init__()** method in the **Pet** class.

**Questions & Discussion**

1) From the first problem in using the **Pet** class to create two child classes, does the **Pet** class have any class attributes? What about instance attributes? What about for the **Dog** class? The **Cat** class?
2) Is there a limit to how many child classes a programmer may create from a single parent class?
3) Can a child class be used to define its own child class? In other words, can a child class be a parent class as well?