14.6: OUR FIRST PYTHON OBJECT

Contributed by Chuck Severance
Clinical Associate Professor (School of Information) at University of Michigan

At its simplest, an object is some code plus data structures that is smaller than a whole program. Defining a function allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain a number of functions (which we call "methods") as well as data that is used by those functions. We call data items that are part of the object "attributes".

We use the `class` keyword to define the data and code that will make up each of the objects. The class keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

CODE 14.6.1 (PYTHON):

```
%%python3

class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Code: http://www.py4e.com/code3/party2.py
```

run restart

Each method looks like a function, starting with the `def` keyword and consisting of an indented block of code. This example has one attribute (x) and one method (party). The methods have a special first parameter that we name by convention `self` .

Much like the `def` keyword does not cause function code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a template indicating what data and code will be contained in each object of type `PartyAnimal` . The class is like a cookie cutter and the objects created using the class are the cookies[2]. You don't put frosting on the cookie cutter, you put frosting on the cookies - and you can put different frosting on each cookie.

A Class and Two Objects

If you continue through the example code, we see the first executable line of code:

```
an = PartyAnimal()
```

This is where we instruct Python to construct (e.g. create) an *object* or "instance of the class named PartyAnimal". It looks like a function call to the class itself and Python constructs the object with the right data and methods and returns the object which is then assigned to the variable `an`. In a way this is quite similar to the following line which we have been using all along:

```
counts = dict()
```

Here we are instructing Python to construct an object using the `dict` template (already present in Python), return the instance of dictionary and assign it to the variable `counts`.

When the PartyAnimal class is used to construct an object, the variable `an` is used to point to that object. We use `an` to access the code and data for that particular instance of a PartyAnimal object.

Each Partyanimal object/instance contains within it a variable `x` and a method/function named `party`. We call that `party` method in this line:

```
an.party()
```

When the `party` method is called, the first parameter (which we call by convention `self`) points to the particular instance of the PartyAnimal object that `party` is called from within. Within the `party` method, we see the line:

```
self.x = self.x + 1
```

This syntax using the 'dot' operator is saying 'the x within self'. So each time `party()` is called, the internal `x` value is incremented by 1 and the value is printed out.

To help make sense of the difference between a global function and a method within a class/object, the following line is another way to call the `party` method within the `an` object:

```
PartyAnimal.party(an)
```

In this variation, we are accessing the code from within the *class* and explicitly passing the object pointer `an` in as the first parameter (i.e. `self` within the method). You can think of `an.party()` as shorthand for the above line.

When the program executes, it produces the following output:

```
So far 1
So far 2
So far 3
So far 4
```

The object is constructed, and the `party` method is called four times, both incrementing and printing the value for `x` within the `an` object.