Classes

Introduction

One thing that you will get to know about programming, is that programmers like to be lazy. If something has been done before, why should you do it again?

That is what functions cover in python. You've already had your code do something special. Now you want to do it again. You put that special code into a function, and re-use it for all it is worth. You can refer to a function anywhere in your code, and the computer will always know what you are talking about. Handy, eh?

Of course, functions have their limitations. Functions don't store any information like variables do - every time a function is run, it starts afresh. However, certain functions and variables are related to each other very closely, and need to interact with each other a lot. For example, imagine you have a golf club. It has information about it (i.e. variables) like the length of the shaft, the material of the grip, and the material of the head. It also has functions associated with it, like the function of swinging your golf club, or the function of breaking it in pure frustration. For those functions, you need to know the variables of the shaft length, head material, etc.

That can easily be worked around with normal functions. Parameters affect the effect of a function. But what if a function needs to affect variables? What happens if each time you use your golf club, the shaft gets weaker, the grip on the handle wears away a little, you get that little more frustrated, and a new scratch is formed on the head of the club? A function cannot do that. A function only makes one output, not four or five, or five hundred. What is needed is a way to group functions and variables that are closely related into one place so that they can interact with each other.

Chances are that you also have more than one golf club. Without classes, you need to write a whole heap of code for each different golf club. This is a pain, seeing that all clubs share common features, it is just that some have changed properties - like what the shaft is made of, and it's weight. The ideal situation would be to have a design of your basic golf club. Each time you create a new club, simply specify its attributes - the length of its shaft, its weight, etc.

Or what if you want a golf club, which has added extra features? Maybe you decide to attach a clock to your golf club (why, I don't know - it was *your* idea). Does this mean that we have to create this golf club from scratch? We would have to write code first for our basic golf club, plus all of that again, and the code for the clock, for our new design. Wouldn't it be better if we were to just take our existing golf club, and then tack the code for the clock to it?

These problems that a thing called object-oriented-programming solves. It puts functions and variables together in a way that they can see each other and work together, be replicated, and altered as needed, and not when unneeded. And we use a thing called a 'class' to do this.

Creating a Class

What is a class? Think of a class as a blueprint. It isn't something in itself, it simply describes how to make something. You can create lots of objects from that blueprint - known technically as an *instance*.

So how do you make these so-called 'classes'? very easily, with the `class` operator:

Code Example 1 - defining a class

```
# Defining a class
class class_name:
    [statement 1]
    [statement 2]
    [statement 3]
    [etc]
```

Makes little sense? Thats ok, here is an example, that creates the definition of a `Shape`:

Code Example 2 - Example of a Class

```
#An example of a class
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    description = "This shape has not been described yet"
    author = "Nobody has claimed to make this shape yet"
    def area(self):
        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y
    def describe(self,text):
        self.description = text
    def authorName(self,text):
        self.author = text
    def scaleSize(self,scale):
        self.x = self.x * scale
    self.y = self.y * scale
```

What you have created is a description of a shape (That is, the variables) and what operations you can do with the shape (That is, the fuctions). This is very important - you have not made an actual shape, simply the description of what a shape is. The shape has a width (x), a height (y), and an area and perimeter (area(self) and perimeter(self)). No code is run when you define a class - you are simply making functions and variables.

The function called __init__ is run when we create an *instance* of Shape - that is, when we create an actual shape, as opposed to the 'blueprint' we have here, __init__ is run. You will understand how this works later.

`self` is how we refer to things in the class from within itself. `self` is the first parameter in any function defined inside a class. Any function or variable created on the first level of indentation (that is, lines of code that start one TAB to the right of where we put `class Shape` is automatically put into `self`. To access these functions and variables elsewhere inside the class, their name must be preceeded with `self` and a full-stop (e.g. `self.variable_name`).

Using a class

Its all well and good that we can make a class, but how do we use one? Here is an example, of what we call creating an instance of a class. Assume that the code example 2 has already been run:

Code Example 3 - Creating a class

```
rectangle = Shape(100,45)
```

What has been done? It takes a little explaining...

The __init__ function really comes into play at this time. We create an *instance* of a class by first giving its name (in this case, `Shape`) and then, in brackets, the values to pass to the __init__ function. The init function runs (using the parameters you gave it in brackets) and then spits out an *instance* of that class, which in this case is assigned to the name `rectangle`.

Think of our class instance, `rectangle`, as a self-contained collection of variables and functions. In the same way that we used `self` to access functions and variables of the class instance from within itself, we use the name that we assigned to it now (`rectangle`) to access functions and variables of the class instance from *outside* of itself. Following on from the code we ran above, we would do this:

Code Example 4 - accessing attributes from outside an instance

```
#finding the area of your rectangle:
print rectangle.area()

#finding the perimeter of your rectangle:
print rectangle.perimeter()

#describing the rectangle
rectangle.describe("A wide rectangle, more than twice\
 as wide as it is tall")

#making the rectangle 50% smaller
rectangle.scaleSize(0.5)

#re-printing the new area of the rectangle
print rectangle.area()
```

As you see, where `self` would be used from within the class instance, its assigned name is used when outside the class. We do this to view and change the variables inside the class, and to access the functions that are there.

We aren't limited to a single instance of a class - we could have as many instances as we like. I could do this:

Code Example 5 - More than one instance

```
longrectangle = Shape(120,10)
fatrectangle = Shape(130,120)
```

and both `longrectangle` and `fatrectangle` have their own functions and variables contained inside them - they are totally independent of each other. There is no limit to the number of instances I could create.

Lingo

Object-oriented-programming has a set of lingo that is associated with it. Its about time that we have this all cleared up:

- When we first describe a class, we are *defining* it (like with functions)
- The ability to group similar functions and variables together is called *encapsulation*
- The word 'class' can be used when describing the code where the class is defined (like how a function is defined), and it can also refer to an instance of that class - this can get confusing, so make sure you know in which form we are talking about classes
- A variable inside a class is known as an *Attribute*
- A function inside a class is known as a *method*

- A class is in the same category of things as variables, lists, dictionaries, etc. That is, they are *objects*
- A class is known as a 'data structure' - it holds data, and the methods to process that data.

Inheritance

Lets have a look back at the introduction. We know how classes group together variables and functions, known as attributes and methods, so that both the data and the code to process it is in the same spot. We can create any number of *instances* of that class, so that we don't have to write new code for every new object we create. But what about adding extra features to our golf club design? This is where *inheritance* comes into play.

Python makes inheritance really easily. We define a new class, based on another, 'parent' class. Our new class brings everything over from the parent, and we can also add other things to it. If any new attributes or methods have the same name as an attribute or method in our parent class, it is used instead of the parent one. Remember the Shape class?

Code Example 6 - the Shape class

```
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    description = "This shape has not been described yet"
    author = "Nobody has claimed to make this shape yet"
    def area(self):
        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y
    def describe(self,text):
        self.description = text
    def authorName(self,text):
        self.author = text
    def scaleSize(self,scale):
        self.x = self.x * scale
    self.y = self.y * scale
```

If we wanted to define a new class, lets say a square, based on our previous Shape class, we would do this:

Code Example 7 - Using inheritance

```
class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x
```

It is just like normally defining a class, but this time we put in brackets after the name, the parent class that we inherited from. As you see, we described a square *really* quickly because of this. That's because we inherited everything from the shape class, and changed only what needed to be changed. In this case we redefined the __init__ function of Shape so that the X and Y values are the same.

Let's take from what we have learnt, and create another new class, this time inherited from Square. It will be two squares, one immediately left of the other:

Code Example 8 - DoubleSquare class

```
# The shape looks like this:
# _________
#|    |    |
#|    |    |
#|____|____|

class DoubleSquare(Square):
    def __init__(self,y):
        self.x = 2 * y
        self.y = y
    def perimeter(self):
        return 2 * self.x + 3 * self.y
```

This time, we also had to redefine the `perimeter` function, as there is a line going down the middle of the shape. Try creating an instance of this class. As a helpful hint, the idle command line starts where your code ends - so typing a line of code is like adding that line to the end of the program you have written.

Pointers and Dictionaries of Classes

Thinking back, when you say that one variable equals another, e.g. `variable2 = variable1`, the variable on the left-hand side of the equal-sign takes on the value of the variable on the right. With class instances, this happens a little differently - the name on the left *becomes* the class instance on the right. So in `instance2 = instance1`, `instance2` is 'pointing' to `instance1` - there are two names given to the one class instance, and you can access the class instance via either name.

In other languages, you do things like this using *pointers*, however in python this all happens behind the scenes.

The final thing that we will cover is dictionaries of classes. Keeping in mind what we have just learnt about pointers, we can assign an instance of a class to an entry in a list or dictionary. This allows for virtually any amount of class instances to exist when our program is run. Lets have a look at the example below, and see how it describes what I am talking about:

Code Example 9 - Dictionaries of Classes

```
# Again, assume the definitions on Shape,
# Square and DoubleSquare have been run.
# First, create a dictionary:
dictionary = {}

# Then, create some instances of classes in the dictionary:
dictionary["DoubleSquare 1"] = DoubleSquare(5)
dictionary["long rectangle"] = Shape(600,45)

#You can now use them like a normal class:
print dictionary["long rectangle"].area()

dictionary["DoubleSquare 1"].authorName("The Gingerbread Man")
print dictionary["DoubleSquare 1"].author
```

As you see, we simply replaced our boring old name on the left-hand side with an exciting, new, dynamic, dictionary entry. Pretty cool, eh?

Conclusion

And that is the lesson on classes! You won't believe how long it took me to write this in a clear-cut manner, and I am still not completely satisfied! I have already gone through and rewritten half of this lesson once, and if you're still confused, I'll probably go through it again. I've probably confused some of you with my own confusion on this topic, but remember - it is not something's name that is important, but what it does (this doesn't work in a social setting, believe me... ;)).

Thanks to all,

sthurlow.com