O: Objects and classes

Gareth McCaughan and Paul Wright Revision 1.8, May 14, 2001

Credits

For the LaTeX source of this sheet, and for more information on LiveWires and on this course, see the LiveWires web site at http://www.livewires.org.uk/python/

Introduction

In these sheets, I've mostly used the word "object" to mean "any thing Python can work with". So a number is an object; so is a string, or a list, or a dictionary.

But there's another meaning of the word "object". You may have heard the words "object-oriented programming". This is a way of thinking about writing programs that's particularly effective for large and complicated programs. Python lets you do object-oriented programming, and as it happens its "objects" are useful even in smaller programs.

This sheet is all about that kind of object in Python. For the Beginners' course, it's very optional. For the Games course, it's essential to know how these objects work.

Objects and classes

In Python (and some other programming languages too) a "kind of object" is called a *class*, and, as you've probably worked out by now, a thing which belongs to one of those kinds is called an *object*.

When you make a class, you can tell Python what the objects in that class can do. The class is like a mold for stamping out objects which do particular things.

The things that an object in that class can do are called *methods*. Methods are like functions which live inside an object (if you don't know what a function is, now would be a good time to look at Sheet 2 or Sheet F).

Type the following into the Python window where you can see the >>> signs (don't type the >>> or ... parts, they're just there to show you how Python will respond to what you type).

```
>>> class Printer:
...   def print_something (self, what):
...     print "This object says", what
...
```

Just press Enter *here*

You've just made a class called `Printer`. Objects belonging to this class only know how to do one thing at the moment, as we've made one *method* called `print_something`. Let's make an object which belongs to the `Printer` class, like this:

```
>>> printobj = Printer ()
```

That was easy. Now `printobj` is an object from the `Printer` class. Things in the `Printer` class know how to do `print_something`. Try this:

```
>>> printobj.print_something ("Hello there")
```

The full stop is used to get at things which live inside a particular object, so `printobj.print_something` uses the `print_something` method inside the `printobj` object. The `printobj` object has the `print_something` method because we made it from the `Printer` class.

Try getting `printobj` to say some other things. Try making some more objects from the `Printer` class with different names, and get them to say things too.

Variables in methods

Let's try something else:

```
>>> class BetterPrinter:                    define a class called BetterPrinter
...   def set_print (self, what):
...     self.thing_we_print = what
...   def print_it (self):
...     print self.thing_we_print
...                                        Just press [Enter] here
```

We've made another class called `BetterPrinter`. It can do a few interesting things.

Tell Python to make an object which belongs to the `BetterPrinter` class. Let's call it `bprinter`, say. Hint: it works the same way as the thing we typed to make an object belonging to `Printer`. Ask a neighbour or a team member if you get stuck.

`bprinter` belongs to `BetterPrinter`, so it knows how to do `set_print`, and also `print_it`, as these are the two methods we made for `BetterPrinter` objects. Try this:

```
>>> bprinter.set_print ("Hi there")
```

What do you think will happen when we tell `bprinter` to `print_it`? Try it and see:

```
>>> bprinter.print_it ()
```

Try changing what `bprinter` prints using the `set_print` method. You can tell `bprinter` to `print_it` more than once without changing what you've told it to print.

Try making some other objects belonging to the `BetterPrinter` class. If you change what one object prints, does it change what another one prints? Try it.

When Python runs a method, it sets `self` to the object which the method belongs to, so we can easily get at this object and the methods and variables which live inside it. Anything else we put in brackets when we run the method gets put into the variables we listed after `self` when we defined the method. So when we say:

```
>>> bprinter.set_print ("Hi there")
```

`set_print` runs with `self` set to the `bprinter` object and `what` set to "Hi there". (Take a look at where we defined the `BetterPrinter` class above: you can see the names there).

Each object belonging to the `BetterPrinter` class puts what you tell it to print in a variable called `thing_we_print`.

Because I wanted each object to have its own `thing_we_print`, I had tell the object to use its own `thing_we_print` when we use the `set_print` and `print_it` methods. This is what `self.thing_we_print` means: all the objects we made from the `BetterPrinter` class have a `thing_we_print`, so we have to say which one we want. When we ran `bprinter`'s `set_print` method, `self` was set to `bprinter`, so

```
self.thing_we_print = what                    This is a line from the set_print method.
```

means

```
bprinter.thing_we_print = "Hi there"
```

You can check this yourself, as Python doesn't stop you having a look at each object's `thing_we_print` directly. Try this:

```
>>> print bprinter.thing_we_print
```

and you'll see you've managed to get at `bprinter`'s `thing_we_print` without using the `print_it` method. Try setting `thing_we_print` without using the `set_it` method, and then using `print_it` to confirm you really have changed what `bprinter` prints.

So, we can access and even change our objects' variables directly rather than through the `set_print` and `print_it` methods. Often it's a bad idea to do this from outside the class, though. Imagine you are writing a class for other programmers to use: you want them to be able to use your class even if you change how it works on the inside. The way you can do this is to fix the method names, the parameters they take (those things you put in brackets when you call a function or method are called *parameters*, if you'd forgotten that), and what the method returns. These things, which the class presents for other people to use, are often called an *interface* (in ordinary English, an interface is just the place where two different things join, so this makes some kind of sense).

If the other programmers know that these things will stay the same whatever changes on the inside of the class, you can then do what you like inside the class to make it do what it's supposed to do. You can even change the method if you think of a better way to do something: as long as we keep the *interface* fixed, the other people using your class won't have to change the way their programs work.

But if people assume they can mess around with the insides of your class, if you change those insides, their programs won't work. Imagine we decide to change the name of `thing_we_print` to `thing` because we don't like typing all those "_" characters. A program which used `thing_we_print` instead of going via `set_print` and `print_it` would not work once we'd done this. But a program which used those two methods (and didn't try to use `thing_we_print` directly) would still work, as we've hidden the workings of the class inside those two methods. This sort of thing becomes important when you're writing reasonably big programs or collaborating with other people on a program.

Inheritance

Let's try something else:

```
>>> class EvenBetterPrinter (BetterPrinter):
...   def add_it (self, what):
...     self.thing_we_print = self.thing_we_print + what
...                                   Just press [Enter] here.
```

You've just made a new class called `EvenBetterPrinter`. Tell Python to make an object from that class called `ebprinter`. (It's just like we've done before: ask if you get stuck).

Now, try this:

```
>>> ebprinter.set_print ("greetings earthlings")
>>> ebprinter.print_it ()
```

What happened? You should find that both `set_print` and `print_it` are methods which `ebprinter` has. But we didn't define them when we defined `EvenBetterPrinter` above, so what's going on?

The answer is that when we created the `EvenBetterPrinter` class, we said this:

```
class EvenBetterPrinter (BetterPrinter):
```

That tells Python that `EvenBetterPrinter` is a kind of `BetterPrinter`, and can do everything that a `BetterPrinter` can do. So `ebprinter` knows about `set_print` and `print_it`, because `BetterPrinter` defines them.

But there's more to `EvenBetterPrinter` than what was in `BetterPrinter`. Try this:

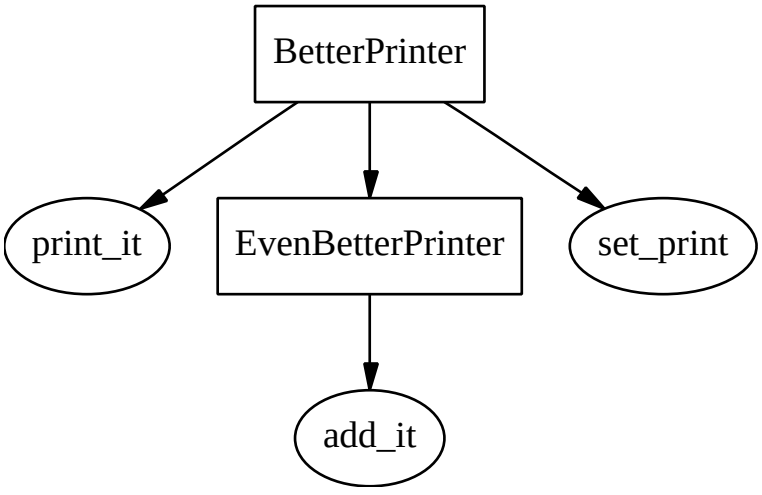
```
>>> ebprinter.add_it (", take me to your leader.")
>>> ebprinter.print_it ()
```

What happened? Have a look back to the definition of the `add_it` method of `EvenBetterPrinter` and see if you can work out what it does. Ask if you get stuck.

Try making other objects from the `EvenBetterPrinter` class and playing with the three methods these objects know about.

What about `BetterPrinter`? Do objects in that class now know about `add_it`? Try making another object from `BetterPrinter` and see whether it knows about `add_it`.

What we have done looks a bit like this (using rectangles for classes and ovals for methods):



`EvenBetterPrinter` is a *subclass* of `BetterPrinter` (on the picture above, it's underneath it: think of a submarine underneath the water, say). Or, alternatively, `BetterPrinter` is a *superclass* of `EvenBetterPrinter` (I suppose you could think of Superman, flying above it! Or maybe not).

We've seen that objects from `EvenBetterPrinter` have the methods that `BetterPrinter` has, as well as the extra `add_it` method we gave `EvenBetterPrinter`. The fancy name for this is *inheritance*, because the `EvenBetterPrinter` class is like a child of the `BetterPrinter` class and inherits these methods from it.

Try making other subclasses of `BetterPrinter` and making objects belonging to the classes you've made. What happens if you make a class which inherits from `BetterPrinter` but has its own `print_it` method which does something different from `BetterPrinter`'s? (In fact, there's a way to get at `BetterPrinter`'s `print_it` even if you have overridden it in your subclass, but we'll not worry about that now).

What's the point of all this? Well, as we said earlier, inheritance is a way to say that A is a kind of B. Maybe we're writing a computer game where all the objects on the screen have some things in common: they all have a speed and a position, they respond to being hit by other objects, and so on. But they also have differences: the player's ship can shoot bullets, for example, but the asteroids the player is shooting at cannot shoot back.

What we can do is make a class for an "object on the screen" which handles these common things, to save us having to write out the same methods over and over again. But all objects on the screen are not the same, so we can make subclasses of our "object on the screen" class to handle the things that each sort of object does differently. The player, the bullets and the asteroids are all kinds of "objects on the screen", so they're all subclasses of the "object on the screen" class.

Magic methods

There's one more thing we need to know about how classes work in Python so that we can understand the worksheets which use them (the Games worksheets, for example).

Let's make another subclass of our old friend `BetterPrinter`:

```
>>> class YAPrinter (BetterPrinter):
...   def __init__ (self, what):
...     self.thing_we_print = what
...     print "An object from YAPrinter is born."
...     print "Yippee! I'm alive!"
...                                    Just press Enter here.
```

Try this:

```
>>> yap = YAPrinter ("Bonjour")
```

What happened? This works because the method named `__init__` is special in Python ("init" is short for *initialisation*, which is a fancy name for the things we do to set something up for the first time). If a class defines an `__init__` method, that method is run whenever a new object from that class is created. So when we made `yap` an object from the `YAPrinter` class, the `__init__` method ran and printed out its message.

You probably noticed something else which was different from what we've seen before. When we created `yap`, we put something inside the brackets after the name of the class. When we do this, the `__init__` method gets passed what we type in the variables we listed after `self` when we defined `__init__`. So, inside `__init__`, `what` was set to "Bonjour". Have a look at the definition of `__init__` above. What does it do with the `what` variable? What will happen when you use `yap`'s `print_it` method? Talk about it to your neighbour, and when you think you know, try it and see.

`__init__` is useful because it allows us to set up the object with the things it needs to know to work. For example, if we were writing our space game, we might give the `__init__` method the location on the screen where each object starts the game.

There are other special methods in Python too, but you needn't worry about defining them accidentally as their names always begin with "__". As long as you don't begin any of your method names like that (unless you mean to) you'll be OK.

Copying and Comparing

You don't strictly need to know this next bit to be able to understand the worksheets, but it can be quite confusing.

Let's make ourselves a nice, simple class :

```
>>> class Thing:
...   def __init__ (self, a, b, c):
...     self.a = a
...     self.b = b
...     self.c = c
...
>>> thing1 = Thing(1, 2, 3)
>>> thing2 = thing1
```

Just press Enter *here.*

So far everything is pretty much as you would expect. If you look at what's in `thing2` (type `print thing2.a` for instance), you'll find that everything has the same value as in `thing1`. But now try this:

```
>>> thing1.a = 0
>>> print thing2.a
```

You might expect that Python would say `1` back to you, because that's what it did last time and you haven't fiddled with `thing2` since then. It doesn't, though. Somehow, when we changed `thing1.a`, we also changed `thing2.a`.

What happened is that `thing1` and `thing2` are actually the same object, just with different names. When we typed `thing2 = thing1`, we didn't make a new object, we just in effect said "we want to call *thing1* by the name *thing2* as well." If you want to make an entirely different copy of an object you have to do it "by hand," making a new `Thing` and then copying `thing1.a`, `thing1.b` and `thing1.c` separately.

That make seem a little odd, but it's usually what you want. When you've picked on one of the asteroids in your game, for instance:

```
asteroid = asteroids[3]
```

you don't really want to create a whole new asteroid on the screen, you just want to change the speed or direction or whatever of the asteroid that you've selected.

Now suppose we want to compare a couple of `Thing`s that we've made. There are two different ways of doing this. The first is to use the "==" that we're used to from comparing numbers. What that does is to go through the contents of each object (assuming that they are the same class), comparing each corresponding item and returning a true value (`1`) if they are all equal.

```
>>> thing3 = Thing(1, 2, 3)
>>> print thing1 == thing2
1
>>> print thing1 == thing3
0
>>> thing3.a = 0
>>> print thing1 == thing3
1
```

You can use all the other conditions in exactly the same way, although only "not equals" (<> or `!=`) is of much use.

The other test you can use is Python's `is` comparison. This checks to see whether two objects are in fact just different names for the same object.

```
>>> print thing1 is thing2
1
>>> print thing1 is thing3
0
>>> print thing1 is not thing3
1
```

As you can imagine, it's a bit faster to test for `thing1 is thing2` than for `thing1 == thing2`, particularly if you

know that there's only one copy of the thing that you're testing for about. That said, it doesn't make very much difference, and you can use either test (so long as you know what you mean!) equally well.

Conclusion

What have we learned? You should now know:

- How to create a class and define methods for it.
- How to create an object from that class and use its methods.
- How to set variables which live inside objects.
- How to create a subclass from another class.
- How to use the `__init__` method.
- How to copy and compare objects.

If you're not sure about any of these things, go back to that section and play with creating objects and classes, and ask for help if you need it. Once you've got the hang of it, you can go on to use classes and objects in your own programs.