L: Loops

Gareth McCaughan 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

Computers are good at repetitive things, so we often want them to do something over and over again (perhaps with slight changes from one time around to the next):

- Add up *all the numbers* in some list
- Move *all the Evil Alien Invaders* one step closer to Earth
- Print out *all the numbers from 1 to 100*
- *Keep asking questions* until you get the right answer

and so on. This is called "looping", for some reason.

Python has two different kinds of loop. This sheet tells you about them.

Two kinds of loop

There's an important difference between the two. One (called a `for` loop) is used when you know in advance how many times you want to do whatever-it-is that the loop does. The other (called a `while` loop) is used when you don't know in advance.

'for' loops: When you know how many times

The `for` loop is called a `for` loop because the first thing you have to type when setting one up is the word `for`. The simplest form looks like this:

```
for x in 1,2,3,4,5,1000:
  print 'Here is a number:'
  print x
```

So, you need to give a "variable name" (`x`), a list of things (`1,2,3,4,5,1000`), and some stuff to do once for each item in the list. The "stuff" will get done once with `x` naming the value 1, once with it naming the value 2, and so on.

The list doesn't have to be written out like that. You can, for instance, say:

```
my_list = [1,2,3,4,5,1000]
for x in my_list:
  print "Here's a number:"
  print x
```

And, of course, `my_list` might actually get its value in some more complicated way: it might be the result of a lengthy calculation instead of being typed in directly.

Ranges

Annoyingly, the commonest sort of loop is rather fiddly to do in Python. Often, you just want to do something 10 times (or 93 times, or whatever). You *could* say `for x in 1,2,3,4,5,6,7,8,9,10:` but you'd quickly get bored of typing all that – and what if you wanted 1000 repetitions? Or if the number of repetitions might vary?

Fortunately, you can say this instead:

```
for x in range(10):
  blah blah blah
```

This will do `blah blah blah` 10 times. It may not do it in quite the way you'd expect, though. The sequence of numbers named by `x` isn't 1,2,3,...,10; it's 0,1,2,...,9. That's still 10 numbers in all.

Incidentally, `range` isn't only for using in `for` loops. You can use it elsewhere too:

```
>>> print range(5)
[0, 1, 2, 3, 4]
```

But all `for` loops have the feature that, when the loop begins, the computer has to know what list it's going through. What if you *don't* know when to stop until after you've started?

'while' loops: When you don't know how many times

For this, Python has another kind of loop. It's called `while` because that's the first word you type when setting up this kind of loop:

```
number = 1
while number < 1000:
  print number
  number = 2*number
```

When the computer sees `while number < 1000:`, what it does is:

- See whether `number < 1000` is true or not.
- If it isn't, abandon the loop: carry on with whatever comes after the end of the loop.
- If it is, do the stuff inside the loop ...
- ... and then go back to the first step, seeing whether `number < 1000` is true or false.

In other words, it does the stuff inside the loop over and over again, but only *while* the "condition" `number < 1000` is true.

You might want to look at Sheet C (*Conditionals*) for more information about conditions, and about Python's ideas of "true" and "false".

Leaving a loop early

Sometimes you want to leave a loop "early". For instance, you might have a `for` loop adding up 100 numbers, but want to stop at once if any of the numbers is 0. (Why? I don't know. It's just an example.)

For this, you need the `break` statement. It means "abandon whatever loop you're in the middle of". So, for instance, to add up all the numbers in a list but stop if you ever hit 0:

```
total = 0
for x in the_list:
  if x == 0:
    break
  total = total+x
```

The `break` statement is particularly useful when you have a loop that's like a `while` loop, but where the condition to be tested doesn't actually come up at the start of the loop. For instance, suppose you want to add up lots of random numbers, and stop if any of the numbers is ever equal to 3. (Yes, this is a pointless example. There are plenty of less pointless examples, but they're all longer and more complicated.) Here's how you could do that.

```
total = 0
while 1:
  r = random_between(1,10)
  if r == 3:
    break
  total = total + r
```

The only really weird thing here is the "`1`" in `while 1:`. As Sheet C tells you, Python considers any non-zero number to be "true". So `while 1` means "Do the following stuff for ever, until you hit a `break`."

More advanced features

Sometimes you want to abandon, not a whole loop, but just a single "iteration" of it: in other words, one trip around the loop. The `continue` statement does that. It's a bit like `break` except that instead of leaping out of the loop it effectively goes back to the start of the loop and begins the next trip around it. If you were already on the last iteration of the loop, `continue` thus does the same as `break`.

The following strange-looking construction is sometimes useful.

```
for n in range(10):
  if a[n] == 'aardvark': break
else:
  print 'No aardvark found!'
```

At first sight, it looks like the "else" here is at the wrong level of indentation. But actually the "else" doesn't go with the "if"; it goes with the "for". What it means is: "Do the following stuff if the loop finished normally, and not by `break` being done".

If you're confused by this, don't worry about it. You aren't likely to need it.