Python Loops and String Manipulation

Last week, we showed you some basic Python programming and gave you some intriguing problems to solve. But it is hard to do anything really exciting until you have mastered what we are covering this week. So read on ...

1 Loopy Python

The thing about computers is that they aren't very smart, but they are extremely fast — fast at doing *exactly* what we tell them to do over and over (and over and over...) again.

Last week, we introduced the `if` *control structure* that decides whether an (indented) block of code should run or not. Any control structure that repeats a block of statements is called a *loop*. We call each individual repetition an *iteration* of the loop.

Almost all programming languages provide one (or more) loop control structures. Python provides two different types, the `while` loop and the `for` loop. We'll look at `while` loops now and save `for` loops for next week!

2 `while` loops

Let's start with a simple example:

```
>>> i = 0
>>> while i < 3:
...   print i
...   i = i + 1
...
0
1
2
```

A `while` loop is basically a repetitive version of an `if` statement. In fact, it uses almost exactly the same syntax: it starts with the `while` keyword, followed by a conditional expression `i < 3` and a colon. On the next line, the block is indented to indicate which statement(s) the `while` controls, just like `if`.

It works like this: *while* the conditional expression `i < 3` is `True`, the body of the loop is run. Here, the body consists of two statements, a `print` statement and `i = i + 1`.

The variable `i` holds the number of times the body has been executed. When the program *first* reaches the `while` statement, `i` has the value `0`. The conditional `i < 3` is `True` this time, so `print i` and `i = i + 1` are run.

At first, the `i = i + 1` line might seem confusing. How can `i` have the same value as `i + 1`? The trick is to remember this is not an equality test, but an assignment statement! So, we calculate the right hand side first (take the value of `i` and add `1` to it), and then we assign that back into the variable `i`.

The `i = i + 1` simply adds one to (or *increments*) the value of `i` every time we go through the loop so we know how many iterations we have completed. So after the *first* iteration, `i` holds the value `1` and we go back to the top of the loop. We then recheck the conditional expression to decide whether to run the body again.

When used this way, the variable `i` is called a *loop counter*. The loop keeps executing until the value of `i` has been incremented to `3`. When this happens, the conditional expression `i < 3` now `False` and so we jump to the statement after the loop body (that is, the end of the program in this case).

3 Anatomy of a Loop

There are four things to think about when writing your own loops:

starting Are there any variables we need to setup before we start a loop? This is often called *loop initialisation*.

stopping How do we know when to stop? This is called the *termination condition*.

doing What are the instructions we want to repeat. This will become the *body* of the loop control structure.

changing Do the instructions change in any way between each repetition. For example, we might need to keep track of how many times we have repeated the body of the loop, that is, the *number of iterations*.

You can use these four like a checklist to make sure the loops you write are doing what you want them to.

Let's look at another example:

```
>>> i = 0
>>> while i < 7:
...   print i
...   i += 2
...
0
2
4
6
```

The initialisation involves setting the loop variable to zero in `i = 0`. The loop will terminate when `i` is no longer smaller than seven `i < 7`. There are two statements in the body.

The second one is a little different this time. Firstly, we're using a special abbreviation `+=` for adding to an existing variable. So `i += 2` is actually the same as `i = i + 2`. It can be used whenever you want to add something to an existing variable, and so you'll often see it for incrementing loop variables.

Secondly, we're adding more than one at a time. The result is the loop will now print out all even numbers less than seven.

4 Reading multiple lines from the user

So far we've only seen examples where you know when you want to stop (e.g. before a number gets too large). However, they also work well when you don't know how many iterations you'll need in advance. For example, when you're reading multiple lines of input from the user:

```
>>> command = raw_input('enter a command: ')
>>> while command != 'quit':
...   print 'your command was ' + command
...   command = raw_input('enter a command: ')
...
enter a command: run
your command was run
enter a command: quit
```

Here it depends on how many times it takes for the user to enter quit as the command. Rather than incrementing a loop counter, the thing that's *changing* involves asking the user for a new command inside the loop.

5 Common loop mistakes

Loops (especially `while` loops) can be rather tricky to get right. Here is one mistake that is particularly common.

This program is going to run for a long time:

```
i = 0
while i < 3:
  print i
```

This program will run forever — well until you turn your computer off, producing a continuous stream of `0`’s. In fact, the only way to stop this *infinite loop* is to press Ctrl-C which terminates the program with an `KeyboardInterrupt` exception, which will look something like this:

```
Traceback (most recent call last):
  File "example1.py", line 3, in -toplevel-
    print i
KeyboardInterrupt
```

The cause of the infinite loop is a very common mistake — forgetting to increment the loop counter. Most loop mistakes involve *starting*, *stopping* or *changing*, so use the loop checklist to help debug your loops.

6 Getting pieces of a string

Often we need to access individual characters or groups of characters in a string. Accessing a single character is done using the square bracket *subscripting* or *indexing* operation:

```
>>> x = "hello world"
>>> x[0]
'h'
>>> x[1]
'e'
```

You can also access strings from the other end using a negative index:

```
>>> x = "hello world"
>>> x[-1]
'd'
>>> x[-5]
'w'
```

A longer part of the string (called a *substring*) can be accessed by using an extended square bracket notation with two indices called a *slice*:

```
>>> x = "hello world"
>>> x[0:5]
'hello'
>>> len(x)
11
>>> x[6:11]
'world'
```

A slice starts from the first index and includes characters up to but *not including* the second index. So, the slice **`x[0:5]`** starts from the **`x[0]`** and goes up to **`x[5]`** (the space) but doesn’t include it. There are a couple of shortcuts too:

```
>>> x[:5]
'hello'
>>> x[6:]
'world'
```

When the first index is zero (when taking a prefix of the sequence) the zero can be left out (like `x[:5]`). When the last index is the length of the string, (when taking a suffix of the sequence) the last index can be left out (like `x[6:]`).

Negative indices work in the same way that they do for indexing. Slicing is slightly more general than indexing. The index values must fall within the length of the string, but slice indices do not have to. If the slice indices fall outside the length of the string then the empty string will be returned for all or part of the slice.

Strings are also *immutable*, which means that once the string is constructed, it cannot be modified. For example, you cannot change a character in an existing string by assigning a new character to a subscript:

```
>>> x[0] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

You must instead create a new string by slicing up the string and concatenating the pieces, and then assigning the result back to the original variable:

```
>>> x = 'X' + x[1:]
>>> x
'Xello world'
```

Every operation that looks like it is modifying a string is actually returning a new string.

7 Checking the contents of strings

To check whether a string contains a particular character, all you need is the `in` conditional operator, like this:

```
>>> x = 'hello world'
>>> 'h' in x
True
>>> 'z' in x
False
```

The `not in` operator does the opposite. It returns `True` when the string *does not* contain the particular character:

```
>>> 'z' not in x
True
```

These operators also work for substrings:

```
>>> 'hello' in x
True
>>> 'howdy' not in x
True
```

These conditional expressions can then be used with Python `if` and `while` loops:

```
name = raw_input('Enter your name? ')
if 'X' in name:
  print 'Your name contains an X'
```

8 Calling methods on strings

Not only can you create new strings using operators like slicing (`s[1:3]`), addition (`"hello" + "world"`) and multiplication (`"abc"*3`), but you can also call special functions on strings which create new strings with different properties. For instance, we can convert a string to lowercase or uppercase:

```
>>> s = "I know my ABC"
>>> s.lower()
'i know my abc'
>>> s.upper()
'I KNOW MY ABC'
```

We can also replace a particular substring with another substring:

```
>>> s = "hello world"
>>> s.replace('l', 'X')
'heXXo worXd'
>>> s.replace('hello', 'goodbye')
'goodbye world'
```

Notice that the convention for calling these string manipulation functions is different – the string that we are manipulating comes **first**, followed by a period (or full stop), then the function name appears with any other info required in parentheses. These special functions that apply to a particular type of value are called *methods*.

So for example, the `s.replace('l', 'X')` is calling the `replace` method on the string stored in variable `s`. `replace` needs two additional pieces of information: the bit of the string to replace, in this case `'l'` and what to replace it with, in this case `'X'`.