Python Lists and Loops

You've made it to Week 3, well done!

Most programs need to keep track of a list (or collection) of things (e.g. names) at one time or another, and this week we'll show you how. We'll also talk about the Python `for` loop, which is designed specifically for working with lists, and show you a few more tricks with strings and integers.

1 Data structures

Imagine a Python program to store a list of authors' names. Given what we've taught you so far, you would need a separately named variable to hold each author's name:

```
>>> name1 = "William Shakespeare"
>>> name2 = "Jane Austen"
>>> name3 = "J.K. Rowling"
```

Every time you wanted to do anything you'd need to mention them individually, e.g. printing a list of authors:

```
>>> print name1
William Shakespeare
>>> print name2
Jane Austen
```

This would just get crazy as the number of authors grows! Luckily programming languages provide ways of efficiently storing and accessing groups of values together: called *data structures*.

2 Lists

The simplest data structure in Python, a *list*, is used to store a list of values (not surprisingly!) of any type:

```
>>> odds = [1, 3, 5, 7, 9]
>>> colours = ['red', 'blue', 'green', 'yellow']
```

The first example is a list of odd integers and the second a list of colour names as strings. Lists are created using a comma separated list of values surrounded by square brackets.

Lists hold a *sequence* of values (like strings hold a sequence of characters). This means items in the list can be accessed using *subscripting* (a.k.a. *indexing*) just like strings:

```
>>> odds[0]
1
>>> colours[-1]
'yellow'
```

Slicing also works like on strings to create a new sublist:

```
>>> colors[:2]
['red', 'blue']
```


An *empty* list is created using just square brackets:

```
>>> values = []
```

We can find out the length of a list just like a string:

```
>>> len(colours)
4
```

Basically, almost every way of accessing one or more characters in a string also works in a similar way to access elements from a list. But lists allow you to store any type of values, not just characters.

3 Authors' names with lists

Lists really simplify our author names problem because we can use a single list (in a single variable) to hold all the authors' names:

```
>>> authors = ['William Shakespeare', 'Jane Austen', 'J.K. Rowling']
```

and we can access each author using an integer index:

```
>>> authors[1]
'Jane Austen'
```

and even better, we can print them out using a `while` loop:

```
>>> i = 0
>>> while i < len(authors):
...   print authors[i]
...   i += 1
William Shakespeare
Jane Austen
J.K. Rowling
```

or for that matter check if we have a particular author:

```
>>> 'J.K. Rowling' in authors
True
>>> 'Dan Brown' in authors
False
```

If we know an element is in the list, we can then find its position:

```
>>> authors.index('William Shakespeare')
0
```

Notice that these snippets work no matter how many authors we have and where they appear in the list.

4 Lists are mutable

Although you can't modify characters in an existing string, the elements in a list *can* be modified by assigning *to* subscripts and slices (the Python documentation calls strings *mutable*):

```
>>> values = [1, 2, 3, 4]
>>> values[-1] = 5
>>> values
[1, 2, 3, 5]
```


5 for loops

The while loop takes 4 lines of code to print the names, because we need to initialise the loop counter, check it is less than the length of the list, and remember to increment it at the end.

Looping over lists is so common that there's an easier way:

```
>>> colours = ['red', 'blue', 'green']
>>> for col in colours:
...   print col
red
blue
green
```

A for loop begins with the keyword for and ends with a colon (because it's a control structure). The col following for is a variable. Each element from the list colours will be assigned to the variable col in turn, and each time the indented body of the loop will then be run.

From the output above we can see that col is assigned the values 'red', 'blue', and 'green' in order because the statement print col is printing out those values.

6 range function

Calling range(n) returns a list of integer values, starting from zero and going up to n - 1:

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

This is often used with for to loop over a range of integers:

```
>>> for i in range(3):
...   print i
0
1
2
```

range can also take a second value:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
```

in which case range returns a list starting at the first value, and going up to but not including the last value. Finally, if you give range a third value, it steps through the values by that amount:

```
>>> range(2, 10, 2)
[2, 4, 6, 8]
```

We can use this trick with negative numbers as well:

```
for i in range(10, 0, -1):
  print i,
print
```

Running this example produces the numbers 10 down to 1 on a single line:

```
10 9 8 7 6 5 4 3 2 1
```


Notice in this example, the `print i` is followed by a comma. This causes `print` not to add a newline to the end. So to put a newline after all of the numbers we put a separate `print` outside of the loop.

The way `range` works should remind you of something else: it works in exactly the same way as slices do for strings and lists.

7 Lists of characters and words

Lists can be constructed from strings using the `list` builtin function.

```
>>> list('abcd')
['a', 'b', 'c', 'd']
```

This is useful if you need to modify individual characters in a string.

Often we want to split a string into a list of pieces, for example splitting a sentence into individual words. The **split** method of strings creates a list of words:

```
>>> line = 'the quick brown fox jumped over the lazy dog'
>>> words = line.split()
>>> words
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

This is useful for checking if a *word* is in a string:

```
>>> 'jumped' in words
True
```

which is different to checking if a substring is in a string:

```
>>> 'jump' in line # jump appears in the string
True
>>> 'jump' in words # but not as a separate word
False
```

8 Joining a list of strings

Another useful string method is **join** which joins a list of strings back together using a string as a *separator*:

```
>>> sep = ':'
>>> values = ['a', 'b', 'c', 'd', 'e']
>>> sep.join(values)
'a:b:c:d:e'
```

You'll often see a literal string (like the space `' '`) used:

```
>>> ' '.join(values)
'a b c d e'
```

and another common trick is to join a list with an empty string:

```
>>> ''.join(values)
'abcdef'
```


9 Appending, sorting and reversing

There are various list methods that can be used to modify lists:

```
>>> pets = ['dog', 'mouse', 'fish']
>>> pets.append('cat')
>>> pets
['dog', 'mouse', 'fish', 'cat']
```

The **append** method adds an item to the end of the list.

```
>>> pets.sort()
>>> pets
['cat', 'dog', 'fish', 'mouse']
```

The **sort** method sorts the items in order, e.g. strings in alphabetical order and numbers in ascending order.

```
>>> pets.reverse()
>>> pets
['mouse', 'fish', 'dog', 'cat']
```

The **reverse** method reverses the order of the entire list.

Just like the string *methods* we saw last week, notice that calls to list methods have the list they operate on appear *before* the method name, separated by a dot, e.g. **pets.reverse()**. Any other values the method needs to do its job is provided in the normal way, e.g. **'cat'** is given as an extra argument *inside* the round brackets in **pets.append('cat')**.

Unlike methods applied to strings, notice that these methods *modify* the original lists rather than creating new ones and so they don't return a value. You can tell this because there is no output when you type them into the interpreter, so we need to puts **pets** on a separate line to see what happened to our list.

10 Loops + lists

If we want to print the words across the screen rather than one per line we can add a comma after the **print** statement like this:

```
>>> words = ['James', 'was', 'here']
>>> for w in words:
...   print w,
...
James was here
```

Another example is reading in multiple lines to produce a list of strings:

```
lines = []
line = raw_input()
while line != '':
  lines.append(line)
  line = raw_input()

print lines
```

This program will keep reading lines until a blank line is entered, that is, until you press Enter on a new line without typing anything before it.

```
Hello World
how are you?

['Hello World', 'how are you?']
```

Let's just quickly analyse the last program to confirm you understand loops and lists.

The first line creates an empty list using empty square brackets **`lines = []`**. The next line reads the first line of input from the user by calling **`raw_input`**. This line is also the *initialisation* statement for the loop, setting the variable **`line`** ready for the conditional expression.

Now we have the loop itself. Here we are using a **`while`** loop because we don't know how many lines there might be entered by the user. We will stop when **`raw_input`** returns us no data, that is, an empty string **`''`**. The **`while`** statement needed to continue reading until the empty string condition is **`while line != ''`**.

It turns out we can abbreviate this further because Python treats the empty string (or a list for that matter) as *equivalent to* **`False`**. Any other value (e.g. the string **`'Hello World'`**) is **`True`**. This means we can write **`while line != '':`** as simply **`while line:`**.

11 More String methods

Last week we showed you how to call a method on a string object (like **`upper`**), and introduced a couple of string methods for converting strings to uppercase and lowercase, and replacing bits of a string with other strings, and we saw the **`split`** and **`join`** methods for strings above. Here are a few more string tricks...

Checking if a string starts or ends with a substring:

```
>>> s = "hello world"
>>> s.startswith('hello')
True
>>> s.endswith('rld')
True
```

Removing whitespace from around a string:

```
>>> s = "   abc  "
>>> s.strip()
'abc'
>>> s.lstrip()
'abc  '
>>> s.rstrip()
'   abc'
```

Finding the index of a character in a string:

```
>>> s = "hello world"
>>> s.find('w')
6
>>> s.find('x')
-1
```

