Chapter 2 - All About Strings

There are several data types in Python. The main data types that you'll probably see the most are string, integer, float, list, dict and tuple. In this chapter, we'll cover the string data type. You'll be surprised how many things you can do with strings in Python right out of the box. There's also a string module that you can import to access even more functionality, but we won't be looking at that in this chapter. Instead, we will be covering the following topics:

- How to create strings
- String concatenation
- String methods
- String slicing
- String substitution

How to Create a String

Strings are usually created in one of three ways. You can use single, double or triple quotes. Let's take a look!

```
>>> my_string = "Welcome to Python!"
>>> another_string = 'The bright red fox jumped the fence.'
>>> a_long_string = '''This is a
multi-line string. It covers more than
one line'''
```

The triple quoted line can be done with three single quotes or three double quotes. Either way, they allow the programmer to write strings over multiple lines. If you print it out, you will notice that the output retains the line breaks. If you need to use single quotes in your string, then wrap it in double quotes. See the following example.

```
>>> my_string = "I'm a Python programmer!"
>>> otherString = 'The word "python" usually refers to a snake'
>>> tripleString = """Here's another way to embed "quotes" in a string"""
```

The code above demonstrates how you could put single quotes or double quotes into a string. There's actually one other way to create a string and that is by using the **str** method. Here's how it works:

```
>>> my_number = 123
>>> my_string = str(my_number)
```

If you type the code above into your interpreter, you'll find that you have transformed the integer value into a string and assigned the string to the variable *my_string*. This is known as **casting**. You can cast some data types into other data types, like numbers into strings. But you'll also find that you can't always do the reverse, such as casting a string like 'ABC' into an integer. If you do that, you'll end up with an error like the one in the following example:

```
>>> int('ABC')
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
ValueError: invalid literal for int() with base 10: 'ABC'
```

We will look at exception handling in a later chapter, but as you may have guessed from the message, this means that you cannot convert a literal into an integer. However, if you had done

```
>>> x = int("123")
```

then that would have worked fine.

It should be noted that a string is one of Python immutable types. What this means is that you cannot change a string's content after creation. Let's try to change one to see what happens:

```
>>> my_string = "abc"
>>> my_string[0] = "d"
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'str' object does not support item assignment
```

Here we try to change the first character from an "a" to a "d"; however this raises a TypeError that stops us from doing so. Now you may think that by assigning a new string to the same variable that you've changed the string. Let's see if that's true:

```
>>> my_string = "abc"
>>> id(my_string)
19397208
>>> my_string = "def"
>>> id(my_string)
25558288
>>> my_string = my_string + "ghi"
>>> id(my_string)
31345312
```

By checking the id of the object, we can determine that any time we assign a new value to the variable, its identity changes.

Note that in Python 2.x, strings can only contain **ASCII** characters. If you require **unicode** in Python 2.x, then you will need to precede your string with a **u**. Here's an example:

```
my_unicode_string = u"This is unicode!"
```

The example above doesn't actually contain any unicode, but it should give you the general idea. In Python 3.x, all strings are unicode.

String Concatenation

Concatenation is a big word that means to combine or add two things together. In this case, we want to know how to add two strings together. As you might suspect, this operation is very easy in Python:

```
>>> string_one = "My dog ate "
>>> string_two = "my homework!"
>>> string_three = string_one + string_two
```

The '+' operator concatenates the two strings into one.

String Methods

A string is an object in Python. In fact, everything in Python is an object. However, you're not really ready for that. If you want to know more about how Python is an object oriented programming language, then you'll need to skip to that chapter. In the meantime, it's enough to know that strings have their very own methods built into them. For example, let's say you have the following string:

```
>>> my_string = "This is a string!"
```

Now you want to cause this string to be entirely in uppercase. To do that, all you need to do is call its **upper()** method, like this:

```
>>> my_string.upper()
```

If you have your interpreter open, you can also do the same thing like this:

```
>>> "This is a string!".upper()
```

There are many other string methods. For example, if you wanted everything to be lowercase, you would use the **lower()** method. If you wanted to remove all the leading and trailing white space, you would use **strip()**. To get a list of all the string methods, type the following command into your interpreter:

```
>>> dir(my_string)
```

You should end up seeing something like the following:

> ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

You can safely ignore the methods that begin and end with double-underscores, such as **__add__**. They are not used in every day Python coding. Focus on the other ones instead. If you'd like to know what one of them does, just ask for **help**. For example, say you want to learn what **capitalize** is for. To find out, you would type

```
>>> help(my_string.capitalize)
```

This would return the following information:

> Help on built-in function capitalize:
>
> **capitalize(...)**
> S.capitalize() -> string
>
> Return a copy of the string S with only its first character capitalized.

You have just learned a little bit about a topic called **introspection**. Python allows easy

introspection of all its objects, which makes it very easy to use. Basically, introspection allows you to ask Python about itself. In an earlier section, you learned about casting. You may have wondered how to tell what type the variable was (i.e. an int or a string). You can ask Python to tell you that!

```
>>> type(my_string)
    <type 'str'>
```

As you can see, the my_string variable is of type str!

String Slicing

One subject that you'll find yourself doing a lot of in the real world is string slicing. I have been surprised how often I have needed to know how to do this in my day-to-day job. Let's take a look at how slicing works with the following string:

```
>>> my_string = "I like Python!"
```

Each character in a string can be accessed using slicing. For example, if I want to grab just the first character, I could do this:

```
>>> my_string[0:1]
```

This grabs the first character in the string up to, but **not** including, the 2nd character. Yes, Python is zero-based. It's a little easier to understand if we map out each character's position in a table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
I		l	i	k	e		P	y	t	h	o	n	!

Thus we have a string that is 14 characters long, starting at zero and going through thirteen. Let's do a few more examples to get these concepts into our heads better.

```
>>> my_string[:1]
'I'
>>> my_string[0:12]
'I like Pytho'
>>> my_string[0:13]
'I like Python'
>>> my_string[0:14]
'I like Python!'
>>> my_string[0:-5]
'I like Py'
>>> my_string[:]
'I like Python!'
>>> my_string[2:]
'like Python!'
```

As you can see from these examples, we can do a slice by just specifying the beginning of the slice (i.e. my_string[2:]), the ending of the slice (i.e. my_string[:1]) or both (i.e. my_string[0:13]). We can even use negative values that start at the end of the string. So the example where we did my_string[0:-5] starts at zero, but ends 5 characters before the end of the string.

You may be wondering where you would use this. I find myself using it for parsing fixed width records in files or occasionally for parsing complicated file names that follow a very specific naming convention. I have also used it in parsing out values from binary-type files. Any job where you need to do text file processing will be made easier if you understand slicing and how to use it effectively.

You can also access individual characters in a string via indexing. Here is an example:

```
>>> print(my_string[0])
```

The code above will print out the first character in the string.

String Formatting

String formatting (AKA substitution) is the topic of substituting values into a base string. Most of the time, you will be inserting strings within strings; however you will also find yourself inserting integers and floats into strings quite often as well. There are two different ways to accomplish this task. We'll start with the old way of doing things and then move on to the new.

Ye Olde Way of Substituting Strings

The easiest way to learn how to do this is to see a few examples. So here we go:

```
>>> my_string = "I like %s" % "Python"
>>> my_string
'I like Python'
>>> var = "cookies"
>>> newString = "I like %s" % var
>>> newString
'I like cookies'
>>> another_string = "I like %s and %s" % ("Python", var)
>>> another_string
'I like Python and cookies'
```

As you've probably guessed, the **%s** is the important piece in the code above. It tells Python that you may be inserting text soon. If you follow the string with a percent sign and another string or variable, then Python will attempt to insert it into the string. You can insert multiple strings by putting multiple instances of **%s** inside your string. You'll see that in the last example. Just note that when you insert more than one string, you have to enclose the strings that you're going to insert with parentheses.

Now let's see what happens if we don't insert enough strings:

```
>>> another_string = "I like %s and %s" % "Python"
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: not enough arguments for format string
```

Oops! We didn't pass enough arguments to format the string! If you look carefully at the example above, you'll notice it has two instances of %s, so to insert strings into it, you have to pass it the same number of strings! Now we're ready to learn about inserting integers and floats. Let's take a look!

```
>>> my_string = "%i + %i = %i" % (1,2,3)
>>> my_string
'1 + 2 = 3'
>>> float_string = "%f" % (1.23)
>>> float_string
'1.230000'
>>> float_string2 = "%.2f" % (1.23)
>>> float_string2
'1.23'
>>> float_string3 = "%.2f" % (1.237)
>>> float_string3
'1.24'
```

The first example above is pretty obvious. We create a string that accept three arguments and we pass them in. Just in case you hadn't figured it out yet, no, Python isn't actually doing any addition in that first example. For the second example, we pass in a float. Note that the output includes a lot of extra zeroes. We don't want that, so we tell Python to limit it to two decimal places in the 3rd example ("%.2f"). The last example shows you that Python will do some rounding for you if you pass it a float that's greater than two decimal places.

Now let's see what happens if we pass it bad data:

```
>>> int_float_err = "%i + %f" % ("1", "2.00")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: %d format: a number is required, not str
```

In this example, we pass it two strings instead of an integer and a float. This raises a TypeError and tells us that Python was expecting a number. This refers to not passing an integer, so let's fix that and see if that fixes the issue:

```
>>> int_float_err = "%i + %f" % (1, "2.00")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: float argument required, not str
```

Nope. We get the same error, but a different message that tells us we should have passed a float. As you can see, Python gives us pretty good information about what went wrong and how to fix it. If you fix the inputs appropriately, then you should be able to get this example to run.

Let's move on to the new method of string formatting!

Templates and the New String Formatting Methodology

This new method was actually added back in Python 2.4 as string templates, but was added as a regular string method via the **format** method in Python 2.6. So it's not really a new method, just newer. Anyway, let's start with templates!

```
>>> print("%(lang)s is fun!" % {"lang":"Python"})
Python is fun!
```

This probably looks pretty weird, but basically we just changed our **%s** into **%(lang)s**, which is basically the %s with a variable inside it. The second part is actually called a Python dictionary that we will be studying in the next section. Basically it's a key:value pair, so when Python sees the key "lang" in the string AND in the key of the dictionary that is passed in, it replaces that key with its value. Let's look at some more samples:

```
>>> print("%(value)s %(value)s %(value)s !" % {"value":"SPAM"})
SPAM SPAM SPAM !
>>> print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2})
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
KeyError: 'z'
>>> print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2, "z":3})
1 + 2 = 3
```

In the first example, you'll notice that we only passed in one value, but it was inserted 3 times! This is one of the advantages of using templates. The second example has an issue in that we forgot to pass in a key, namely the "z" key. The third example rectifies this issue and shows the result. Now let's look at how we can do something similar with the string's format method!

```
>>> "Python is as simple as {0}, {1}, {2}".format("a", "b", "c")
'Python is as simple as a, b, c'
>>> "Python is as simple as {1}, {0}, {2}".format("a", "b", "c")
'Python is as simple as b, a, c'
>>> xy = {"x":0, "y":10}
>>> print("Graph a point at where x={x} and y={y}".format(**xy))
Graph a point at where x=0 and y=10
```

In the first two examples, you can see how we can pass items positionally. If we rearrange the order, we get a slightly different output. The last example uses a dictionary like we were using in the templates above. However, we have to extract the dictionary using the double asterisk to get it to work correctly here.

There are lots of other things you can do with strings, such as specifying a width, aligning the text, converting to different bases and much more. Be sure to take a look at some of the references below for more information.

- Python's official documentation on the str type
- String Formatting
- More on String Formatting
- Python 2.x documentation on unicode

Wrapping Up

We have covered a lot in this chapter. Let's review:

First we learned how to create strings themselves, then we moved on to the topic of string concatenation. After that we looked at some of the methods that the string object gives us. Next we looked at string slicing and we finished up by learning about string substitution.

In the next chapter, we will look at three more of Python's built-in data types: lists, tuples and dictionaries. Let's get to it!