Manipulating Python Strings

If you have been exposed to another programming language before, you might have learned that you need to *declare* or *type* variables before you can store anything in them. This is not necessary when working with strings in Python. We can create a string simply by putting content wrapped with quotation marks into it with an equal sign (=):

```
message = "Hello World"
```

String Operators: Adding and Multiplying

A string is a type of object, one that consists of a series of characters. Python already knows how to deal with a number of general-purpose and powerful representations, including strings. One way to manipulate strings is by using *string operators*. These operators are represented by symbols that you likely associate with mathematics, such as +, -, *, /, and =. When used with strings, they perform actions that are similar to, but not the same as, their mathematical counterparts.

Concatenate

This term means to join strings together. The process is known as *concatenating* strings and it is done using the plus (+) operator. Note that you must be explicit about where you want blank spaces to occur by placing them between single quotation marks also.

In this example, the string “message1” is given the content “hello world”.

```
message1 = 'hello' + ' ' + 'world'print(message1)-> hello world
```

Multiply

If you want multiple copies of a string, use the multiplication (*) operator. In this example, string *message2a* is given the content “hello” times three; string *message 2b* is given content “world”; then we print both strings.

```
message2a = 'hello ' * 3message2b = 'world'print(message2a +
message2b)-> hello hello hello world
```

Append

What if you want to add material to the end of a string successively? There is a special operator for that (+=).

```
message3 = 'howdy'message3 += ' 'message3 += 'world'print(message3)->
howdy world
```

String Methods: Finding, Changing

In addition to operators, Python comes pre-installed with dozens of string methods that allow you to do things to strings. Used alone or in combination, these methods can do just about anything you can imagine to strings. The good news is that you can reference a list of String Methods on the Python website, including information on how to use each properly. To make

sure that you've got a basic grasp of string methods, what follows is a brief overview of some of the more commonly used ones:

Length

You can determine the number of characters in a string using `len`. Note that the blank space counts as a separate character.

```
message4 = 'hello' + ' ' + 'world'print(len(message4))-> 11
```

Find

You can search a string for a *substring* and your program will return the starting index position of that substring. This is helpful for further processing. Note that indexes are numbered from left to right and that the count starts with position 0, not 1.

```
message5 = "hello world"message5a =
message5.find("worl")print(message5a)-> 6
```

If the substring is not present, the program will return a value of -1.

```
message6 = "Hello World"message6b =
message6.find("squirrel")print(message6b)-> -1
```

Lower Case

Sometimes it is useful to convert a string to lower case. For example, if we standardize case it makes it easier for the computer to recognize that "Sometimes" and "sometimes" are the same word.

```
message7 = "HELLO WORLD"message7a = message7.lower()print(message7a)->
hello world
```

The opposite effect, raising characters to upper case, can be achieved by changing `.lower()` to `.upper()`.

Replace

If you need to replace a substring throughout a string you can do so with the `replace` method.

```
message8 = "HELLO WORLD"message8a = message8.replace("L",
"pizza")print(message8a)-> HEpizzapizzaO WORpizzaD
```

Slice

If you want to `slice` off unwanted parts of a string from the beginning or end you can do so by creating a substring. The same kind of technique also allows you to break a long string into more manageable components.

```
message9 = "Hello World"message9a = message9[1:8]print(message9a)->
ello Wo
```

You can substitute variables for the integers used in this example.

```
startLoc = 2endLoc = 8message9b = message9[startLoc:
endLoc]print(message9b)-> llo Wo
```

This makes it much easier to use this method in conjunction with the `find` method as in the next example, which checks for the letter “d” in the first six characters of “Hello World” and correctly tells us it is not there (-1). This technique is much more useful in longer strings – entire documents for example. Note that the absence of an integer before the colon signifies we want to start at the beginning of the string. We could use the same technique to tell the program to go all the way to the end by putting no integer after the colon. And remember, index positions start counting from 0 rather than 1.

```
message9 = "Hello World"print(message9[:5].find("d"))-> -1
```

There are lots more, but the string methods above are a good start. Note that in this last example, we are using square brackets instead of parentheses. This difference in *syntax* signals an important distinction. In Python, parentheses are usually used to *pass an argument* to a function. So when we see something like

```
print(len(message7))
```

it means pass the string *message7* to the function `len` then send the returned value of that function to the `print` statement to be printed. If a function can be called without an argument, you often have to include a pair of empty parentheses after the function name anyway. We saw an example of that, too:

```
message7 = "HELLO WORLD"message7a = message7.lower()print(message7a)->
hello world
```

This statement tells Python to apply the `lower` function to the string *message7* and store the returned value in the string *message7a*.

The square brackets serve a different purpose. If you think of a string as a sequence of characters, and you want to be able to access the contents of the string by their location within the sequence, then you need some way of giving Python a location within a sequence. That is what the square brackets do: indicate a beginning and ending location within a sequence as we saw when using the `slice` method.

Escape Sequences

What do you do when you need to include quotation marks within a string? You don’t want the Python interpreter to get the wrong idea and end the string when it comes across one of these characters. In Python, you can put a backslash (\) in front of a quotation mark so that it doesn’t terminate the string. These are known as escape sequences.

```
print('\"')-> "
```

```
print('The program printed \"hello world\"')-> The program printed "hello world"
```

Two other escape sequences allow you to print tabs and newlines:

```
print('hello\thello\thello\nworld')->hello hello helloworld
```