STRINGS

A string is a *sequence* of *characters*. Strings are basically just a bunch of words.

You will be using strings in almost every Python program that you write, so pay attention to the following part.

Single Quote

You can specify strings using single quotes such as 'Quote me on this'.

All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Double Quotes

Strings in double quotes work exactly the same way as strings in single quotes. An example is "What's your name?".

Triple Quotes

You can specify multi-line strings using triple quotes - (""" or '''). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
```

```
This is the second line.

"What's your name?," I asked.

He said "Bond, James Bond."

'''
```

Strings Are Immutable

This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

NOTE	*Note for C/C++ Programmers* There is no separate char data type in Python. There is no real need for it and I am sure you won't miss it.
NOTE	*Note for Perl/PHP Programmers* Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.

The format method

Sometimes we may want to construct strings from other information. This is where the format() method is useful.

Save the following lines as a file str_format.py:

```
age = 20

name = 'Swaroop'


print '{0} was {1} years old when he wrote this book'.format(name, age)

print 'Why is {0} playing with that python?'.format(name)
```

Output:

```
$ python str_format.py

Swaroop was 20 years old when he wrote this book

Why is Swaroop playing with that python?
```

How It Works

A string can use certain specifications and subsequently, the format method can be called to substitute those specifications with corresponding arguments to the format method.

Observe the first usage where we use {0} and this corresponds to the variable name which is the first argument to the format method. Similarly, the second specification is {1} corresponding to age which is the second argument to the format method.

Note that Python starts counting from 0 which means that first position is at index 0, second position is at index 1, and so on.

Notice that we could have achieved the same using string concatenation:

```
name + ' is ' + str(age) + ' years old'
```

but that is much uglier and error-prone. Second, the conversion to string would be done automatically by the format method instead of the explicit conversion to strings needed in this case. Third, when using the format method, we can change the message without having to deal with the variables used and vice-versa.

Also note that the numbers are optional, so you could have also written as:

```
age = 20
name = 'Swaroop'

print '{} was {} years old when he wrote this book'.format(name, age)
print 'Why is {} playing with that python?'.format(name)
```

which will give the same exact output as the previous program.

What Python does in the format method is that it substitutes each argument value into the place of the specification.

There can be more detailed specifications such as:

```
# decimal (.) precision of 3 for float '0.333'
print '{0:.3f}'.format(1.0/3)
# fill with underscores (_) with the text centered
# (^) to 11 width '___hello___'
print '{0:_^11}'.format('hello')
# keyword-based 'Swaroop wrote A Byte of Python'
print '{name} wrote {book}'.format(name='Swaroop',
                                   book='A Byte of Python')
```

Output:

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

Since we are discussing formatting, note that print always ends with an invisible "new line" character (\n) so that repeated calls to print will all print on a separate line each.

To prevent this newline character from being printed, you can end the statement with a comma:

```
print "a",
print "b",
```

Output is:

```
a b
```

Escape Sequences

Suppose, you want to have a string which contains a single quote ('), how will you specify this string? For example, the string is "What’s your name?". You cannot specify 'What’s your name?' because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This can be done with the help of what is called an *escape sequence*. You specify the single quote as \' : notice the backslash. Now, you can specify the string as 'What's your name?'.

Another way of specifying this specific string would be "What’s your name?" i.e. using double quotes. Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence \\.

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown previously or you can use an escape sequence for the newline character - \n to indicate the start of a new line. An example is:

```
'This is the first line\nThis is the second line'
```

Another useful escape sequence to know is the tab: \t. There are many more escape sequences but I have mentioned only the most useful ones here.

One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example:

```
"This is the first sentence. \
This is the second sentence."
```

is equivalent to

```
"This is the first sentence. This is the second sentence."
```

Raw String

If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing r or R to the string.

An example is:

```
r"Newlines are indicated by \n"
```

NOTE

Note for Regular Expression Users

Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as `'\\1'` or `r'\1'`.

Source: http://www.swaroopch.com/notes/python/