Chapter 4 - Conditional Statements

Every computer language I have ever used has had at least one conditional statement. Most of the time that statement is the **if/elif/else** structure. This is what Python has. Other languages also include the **case/switch** statement which I personally enjoy, however Python does not include it. You can make your own if you really want to, but this book is focused on learning Python fundamentals, so we're going to be only focusing on what's included with Python in this chapter.

The conditional statement checks to see if a statement is True or False. That's really all it does. However we will also be looking at the following Boolean operations: **and**, **or**, and **not**. These operations can change the behaviour of the conditional in simple and complex ways, depending on your project.

The if statement

Python's if statement is pretty easy to use. Let's spend a few minutes looking at some examples to better acquaint ourselves with this construct.

```
>>> if 2 > 1:
        print("This is a True statement!")
    This is a True Statement!
```

This conditional tests the "truthfulness" of the following statement: 2 > 1. Since this statement evaluates to True, it will cause the last line in the example to print to the screen or **standard out** (stdout).

Python Cares About Space

The Python language cares a lot about space. You will notice that in our conditional statement above, we indented the code inside the **if** statement four spaces. This is very important! If you do not indent your blocks of code properly, the code will not execute properly. It may not even run at all.

Also, do **not** mix tabs and spaces. IDLE will complain that there is an issue with your file and you will have trouble figuring out what the issue is. The recommended number of spaces to indent a block of code is four. You can actually indent your code any number of spaces as long as you are consistent. However, the 4-space rule is one that is recommended by the Python Style Guide and is the rule that is followed by the Python code developers.

Let's look at another example:

```
>>> var1 = 1
>>> var2 = 3
>>> if var1 > var2:
        print("This is also True")
```

In this one, we compare two variables that translate to the question: Is 1 > 3? Obviously one is not greater than three, so it doesn't print anything. But what is we wanted it to print something? That's where the **else** statement comes in. Let's modify the conditional to add that piece:

```
if var1 > var2:
    print("This is also True")
else:
    print("That was False!")
```

If you run this code, it will print the string that follows the **else** statement. Let's change gears here and get some information from the user to make this more interesting. In Python 2.x, you can get information using a built-in called **raw_input**. If you are using Python 3.x, then you'll find that raw_input no longer exists. It was renamed to just **input**. They function in the same way though. To confuse matters, Python 2.x actually has a built-in called **input** as well; however it tries to execute what is entered as a Python expression whereas raw_input returns a string. Anyway, we'll be using Python 2.x's **raw_input** for this example to get the user's age.

```
# Python 2.x code
value = raw_input("How much is that doggy in the window? ")
value = int(value)

if value < 10:
    print("That's a great deal!")
elif 10 <= value <= 20:
    print("I'd still pay that...")
else:
    print("Wow! That's too much!")
```

Let's break this down a bit. The first line asks the user for an amount. In the next line, it converts the user's input into an integer. So if you happen to type a floating point number like **1.23**, it will get truncated to just **1**. If you happen to type something other than a number, then you'll receive an exception. We'll be looking at how to handle exceptions in a later chapter, so for now, just enter an integer.

In the next few lines, you can see how we check for 3 different cases: less than 10, greater than or equal to 10 but less than or equal to 20 or something else. For each of these cases, a different string is printed out. Try putting this code into IDLE and save it. Then run it a few times with different inputs to see how it works.

You can add multiple **elif** statements to your entire conditional. The **else** is optional, but makes a good default.

Boolean Operations

Now we're ready to learn about Boolean operations (and, or, not). According to the Python documentation, their order of priority is first **or**, then **and**, then **not**. Here's how they work:

- **or** means that if any conditional that is "ored" together is True, then the following statement runs
- **and** means that all statements must be True for the following statement to run
- **not** means that if the conditional evaluates to False, it is True. This is the most confusing, in my opinion.

Let's take a look at some examples of each of these. We will start with **or**.

```
x = 10
y = 20

if x < 10 or y > 15:
    print("This statement was True!")
```

Here we create a couple of variables and test if one is less than ten or if the other is greater than 15. Because the latter is greater than 15, the print statement executes. As you can see, if

one or both of the statements are True, it will execute the statement. Let's take a look at how **and** works:

```
x = 10
y = 10
if x == 10 and y == 15:
    print("This statement was True")
else:
    print("The statement was False!")
```

If you run the code above, you will see that first statement does not get run. Instead, the statement under the **else** is executed. Why is that? Well, it is because what we are testing is both x **and** y are 10 and 15 respectively. In this case, they are not, so we drop to the else. Thus, when you **and** two statements together, **both** statements have to evaluate to True for it to execute the following code. Also note that to test equality in Python, you have to use a double equal sign. A single equals sign is known as the **assignment operator** and is only for assigning a value to a variable. If you had tried to run the code above with one of those statement only having one equals sign, you would have received a message about invalid syntax.

Note that you can also **or** and **and** more than two statements together. However, I do not recommend that as the more statements that you do that too, the harder it can be to understand and debug.

Now we're ready to take a look at the **not** operation.

```
my_list = [1, 2, 3, 4]
x = 10
if x not in my_list:
    print("'x' is not in the list, so this is True!")
```

In this example, we create a list that contains four integers. Then we write a test that asks if "x" is not in that list. Because "x" equals 10, the statement evaluates to True and the message is printed to the screen. Another way to test for **not** is by using the exclamation point, like this:

```
x = 10
if x != 11:
    print("x is not equal to 11!")
```

If you want to, you can combine the **not** operation with the other two to create more complex conditional statements. Here is a simple example:

```
my_list = [1, 2, 3, 4]
x = 10
z = 11
if x not in my_list and z != 10:
    print("This is True!")
```

Checking for Nothing

Because we are talking about statements that evaluate to True, we probably need to cover what evaluates to False. Python has the keyword **False** which I've mentioned a few times. However an empty string, tuple or list also evaluates to False. There is also another keyword that basically evaluates to False which is called **None**. The None value is used to represent the absence of value. It's kind of analogous to Null, which you find in databases. Let's take a look at some code to help us better understand how this all works:

```
empty_list = []
empty_tuple = ()
empty_string = ""
nothing = None

if empty_list == []:
    print("It's an empty list!")

if empty_tuple:
    print("It's not an empty tuple!")

if not empty_string:
    print("This is an empty string!")

if not nothing:
    print("Then it's nothing!")
```

The first four lines set up four variables. Next we create four conditionals to test them. The first one checks to see if the **empty_list** is really empty. The second conditional checks to see if the **empty_tuple** has something in it. Yes, you read that right, The second conditional only evaluates to True if the tuple is **not** empty! The last two conditionals are doing the opposite of the second. The third is checking if the string **is** empty and the fourth is checking if the **nothing** variable is really None.

The **not** operator means that we are checking for the opposite meaning. In other words, we are checking if the value is NOT True. So in the third example, we check if the empty string is REALLY empty. Here's another way to write the same thing:

```
if empty_string == "":
    print("This is an empty string!")
```

To really nail this down, let's set the **empty_string** variable to actually contain something:

```
>>> empty_string = "something"
>>> if empty_string == "":
        print("This is an empty string!")
```

If you run this, you will find that nothing is printed as we will only print something if the variable is an empty string.

Please note that none of these variables equals the other. They just evaluate the same way. To prove this, we'll take a look at a couple of quick examples:

```
>>> empty_list == empty_string
False
>>> empty_string == nothing
False
```

As you can see, they do not equal each other. You will find yourself checking your data structures for data a lot in the real world. Some programmers actually like to just wrap their structures in an exception handler and if they happen to be empty, they'll catch the exception. Others like to use the strategy mentioned above where you actually test the data structure to see if it has something in it. Both strategies are valid.

Personally, I find the **not** operator a little confusing and don't use it that much. But you will find it useful from time to time.

Special Characters

Strings can contain special characters, like tabs or new lines. We need to be aware of those as they can sometimes crop up and cause problems. For example, the new line character is defined as "n", while the tab character is defined as "t". Let's see a couple of examples so you will better understand what these do:

```
>>> print("I have a \n new line in the middle")
I have a
 new line in the middle
>>> print("This sentence is \ttabbed!")
This sentence is    tabbed!
```

Was the output as you expected? In the first example, we have a "n" in the middle of the sentence, which forces it to print out a new line. Because we have a space after the new line character, the second line is indented by a space. The second example shows what happens when we have a tab character inside of a sentence.

Sometimes you will want to use escape characters in a string, such as a backslash. To use escape characters, you have to actually use a backslash, so in the case of a backslash, you

would actually type two backslashes. Let's take a look:

```
>>> print("This is a backslash \")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
EOL while scanning string literal: <string>, line 1, pos 30
>>> print("This is a backslash \\")
This is a backslash \
```

You will notice that the first example didn't work so well. Python thought we were escaping the double-quote, so it couldn't tell where the end of the line (EOL) was and it threw an error. The second example has the backslash appropriately escaped.

if __name__ == "__main__"

You will see a very common conditional statement used in many Python examples. This is what it looks like:

```
if __name__ == "__main__":
    # do something!
```

You will see this at the end of a file. This tells Python that you only want to run the following code if this program is executed as a standalone file. I use this construct a lot to test that my code works in the way I expect it to. We will be discussing this later in the book, but whenever you create a Python script, you create a Python module. If you write it well, you might want to import it into another module. When you do import a module, it will **not** run the code that's under the conditional because **__name__** will no longer equal **"__main__"**. We will look at this again in **Chapter 11** when we talk about **classes**.

Wrapping Up

We've covered a fair bit of ground in this chapter. You have learned how to use conditional statements in several different ways. We have also spent some time getting acquainted with Boolean operators. As you have probably guessed, each of these chapters will get slightly more complex as we'll be using each building block that we learn to build more complicated pieces of code. In the next chapter, we will continue that tradition by learning about Python's support of loops!