LESSON A-15

NESTED IF STATEMENTS &
STRING/INTEGER CONVERSION

In this lesson you'll continue to learn how to work with if statements by adding increasingly complex criteria. You will then move on to learn how to specify strings versus integers in Python.

Nested If Statements

In Lesson A-14, you learned to structure simple if statements in Python. But what happens if you need to have multiple criteria? For example, if you only wanted to print a receipt if the sale value was $1 and the item was red? For this, you would need to use nested if statements.

A nested if statement, describes an if statement that's inside of, and dependent on, another if statement. This can be used to add extra layers of decisions that you might want to make happen in your program.

Formatting: Importance of Indentation

When formatting nested if statements, indentation is very important. This tells Python when statements are dependent upon each other.

For example, this program simulates a code that might open a lock. The lock can only open with the code 345.

```
if x == 3:
    print('First digit accepted')
    if y == 4:
        print('Second digit accepted')
        if z == 5:
            print('Third digit Accepted')
            print('Lock open')
```

What will happen when the digits 3, 4, 5 are entered?

The code digits are initially programmed in using the variables x, y, and z. The first if condition x == 3 will be evaluated, and if True, **First digit accepted** will be printed, and the next condition will be evaluated. If y == 4 evaluates as True then **Second digit accepted** will be printed, and the third condition z == 5 will be evaluated. If this evaluates as True, then **Third Digit accepted** and **Lock open** will both be printed.

The only way **Lock Open** can ever be printed is if all three conditions preceding it evaluate as True. If any of the digits do not equal the expected digit, then the rest of the code will be skipped, and the lock will not open.

But what happens if the statements aren't properly indented? Python no longer sees them as dependent upon one another and you can't obtain the conditional buildup seen in the last example.

For example, consider the following code:

```
if x == 3:
    print('First digit accepted')
if y == 4:
    print('Second digit accepted')
if z == 5:
    print('Third digit Accepted')
    print('Lock open')
```

What happens if the user enters 555?

Since the first digit is wrong, the first condition will evaluate as False, so no message would be printed. Due to the improper indentation, the second condition will also be evaluated, even though the first digit was incorrect. Without indentation, Python doesn't realize it's not supposed to proceed to the next line of code! In this case, the second digit is also incorrect, so no message will be printed.

The problem really shows up when the third condition is tested. This digit is correct, so the condition will evaluate as True, and both **Third digit accepted** and **Lock open** will be printed. The lock was allowed to open, even though two of the three digits were incorrect. Without indentation, Python treated all three conditions as independent of one another.

If your intent is to have multiple if statements trigger an action, then make sure your indentation is allowing the conditions to be evaluated as expected. Also, test to make sure your program is operating as designed, and that incorrect entries don't generate unexpected behavior.

ALWAYS USE FOUR SPACES TO INDENT CODE. It may be tempting to use the tab key out of convenience, but tabs can be interpreted differently by different programs. Spaces are treated the same way in every program, so they are the safest way to learn to indent your code. If you need more levels of indentation just add four more spaces for each level:

```
first_line              # this line has no spaces
    second_line         # this line has 4 spaces
        third_line      # this line has 8 spaces
    fourth_line         # this line has 4 spaces
        fifth_line      # this line has 8 spaces
```

Strings vs. Integers

As you learned in the previous lesson, strings are one or more characters made up of letters, numbers, or symbols. You can ask Python if string '6' equals string '6' and it will evaluate as True. This happens because those two characters are the same, much like Python would say that the strings 'a' and 'a' are equal. You can also add string 'a' to string 'a' and you will get an output string of 'aa'. This worked great in previous lessons to add “My name is “ and “name” together to get a full sentence.

The problem occurs when you try to do mathematical calculations with strings of numbers. Consider the following program:

```
a = '4'

b = '2'

c = a + b

print(c)
```

What number do you think will be printed to the console as c? Here's a hint: it's not 6.

42 will be printed because the character '4' added to the character '2' is '42'. Python doesn't understand that these characters represent numeric values. For that you will need to use integers.

Converting a Value to an Integer

Integers are only allowed to be numbers, so trying to store the value 'apple' as an integer will result in an error. Here are some examples:

Statement	Action	Use for Numeric Functions?
x = '6'	This will store the value 6 as a string in a variable called x	No
x = 6	This will store the value 6 as an integer in a variable called x	Yes
x = input('Type a number: ')	Typing 6 at this prompt will cause the program to store the value of 6 as a string in a variable called x	No

The last one is a bit of a problem. How do you use an input of 6 for a mathematical equation when Python is storing it as a string? Fortunately, Python has a great way to convert back and forth between strings and integers. The command for converting a numeric string to an integer is:

```
int()
```

Just insert a variable value between the parentheses, and Python will convert the value to an integer, so you can use that value for math purposes. Remember, this will only work on numeric values. Attempting to convert the string value 'apple' into an integer will result in a Python error.

Here is an example of converting a string to an integer:

`x = '42'`	This will initially store the value of 42 as a string in variable x
`y = int(x)`	This will take the current value of x, convert it to an integer, and store that new value in the variable y

This integer command will be very useful in Activity #3, where you will convert the numeric user input from a string into an integer, so you can compare it to other numbers later in the program.

Converting a Value to a String

But what if you have the opposite problem? You have an integer you need to convert to a string so you can combine it with another string? Python cannot merge integers and strings to be displayed together so you must convert any integers to strings.

The command to convert a value to a string is:

`str()`

It can be used just like the **`int()`** command above. Here is an example of converting an integer to a string:

`x = 42` `y = str(x)`	This will initially store the value of 42 as an integer in variable x
	This will take the current value of x, convert it to a string, and store that new value in the variable y

This can be helpful when trying to print integers and strings together. As noted above, Python cannot concatenate (merge) strings and integers together in the same print command.

Bad Code	Good Code

`x = 42`
`print('Your number is ' + x)`
 	
This print statement is trying to combine the string 'Your number is ' and the integer value of x. This will result in an error.
	`x = 42`
	`y = str(x)`
	
	`print('Your number is ' + y)`
	
	Since x is being converted into a string value called y, this print statement can combine 'Your number is ' and the string value of y, with no errors. 'Your number is 42' will be printed to the shell.

This conversion can also happen within a single line. In Lesson A-13 you used this statement to bring in user input as an integer:

```
var = int(input('Enter a number: '))
```

Normally the input command will bring in any value entered as a string, even numeric values. The problem with this is that you can't perform math operations on a string, even when the string is something like '6'. That '6' might as well be 'apple' or 'tree'. Python has no idea how to subtract 2 from 'apple' or from '6'.

The input string must be converted to an integer using the **int()** command. There are a couple of ways to accomplish the same thing:

Option #1:

```
var = input('Enter a number: ')

var = int(var)
```

The input from the user is initially saved to the variable **var**. Then the int(var) will take the integer value **var** and overwrite the string value of **var** with the integer value.

Or you can do this by wrapping the **int()** around the entire input command as seen below.

Option #2:

```
var = int(input('Enter a number: '))
```

In this example, from Lesson 13, the user is asked for input, that input is converted to an integer, and that integer value is saved to the variable called **var**. This way is slightly more complicated looking, but it results in one less line of code.