ECOR 1041 Lecture 3
Python demo: variables.

Some Experiments to Learn About Python Variables

We are going to create some experiments to help us learn about Python variables, including the evaluation of expressions that contain variables.

Variables are created by assignment statements. Here is an assignment statement that creates a variable named `degrees_C` that represents a temperature on the Celsius scale, and assigns or binds it to the `float` 21.0.

```
>>> degrees_C = 21.0
```

Notice the equals sign, =. In mathematics, this symbol denotes equality; for example, 2 + 2 = 4. Another example: the equality specified by the equation x + 3 = 7 is true when the value of x is 7.

In Python (and many other programming languages), = denotes assignment. The expression that follows = is evaluated, and that value is assigned to the variable.

Also, notice that nothing was displayed after `degrees_C = 21.0`. In Python, an assignment is not an expression that evaluates to a value, so there is no result to be displayed in the shell.

Aside: Python (unlike C, C++, Java and many other programming languages) does not have variable declaration statements (statements that allocate one or more variables, specifying the name and type of each one).

When an expression contains variables, Python uses the values to which the variables are bound when it evaluates the expression.

Let us evaluate `degrees_C`:

```
>>> degrees_C
21.0
```

Here is another example: we can calculate the equivalent Fahrenheit temperature by multiplying the value bound to `degrees_C` by nine-fifths, then adding 32. One Python expression for performing this calculation is:

```
>>> degrees_C * 9 / 5 + 32
69.8
```

If we rearrange the expression slightly, we get a different result:

```
>>> 9 / 5 * degrees_C + 32
69.80000000000001
```

Aside: Why is the result now 69.80000000000001, and not 69.8? Why do two expressions that are mathematically equivalent yield different results? (Think back to the second lecture.)

Combining these concepts, here is how we convert a temperature from degrees Celsius to degrees Fahrenheit, then assign the result to a new variable named `degrees_F`:

```
>>> degrees_C = 21.0
>>> degrees_F = degrees_C * 9 / 5 + 32
>>> degrees_F
69.8
```

The value that is bound to a variable can be changed as a program executes. (That is why they are called variables.) For example, we can assign a new value (the `float` 0.0) to `degrees_C`:

```
>>> degrees_C = 0.0
>>> degrees_C
0.0
```

Now consider this code. What will Python display when the last expression is evaluated?

```
>>> degrees_C = 21.0
>>> degrees_F = degrees_C * 9 / 5 + 32
>>> degrees_F
69.8

>>> degrees_C = 0.0
>>> degrees_F
```

Answer: `69.8`

Students who have used spreadsheets sometimes think the value bound to `degrees_F` will change to 32.0 when we bind a new value to `degrees_C`. (In a spreadsheet, when we change the value stored in a cell, and other cells contain formulas that use the first cell, then the values of those cells are automatically recalculated. That is not how assignment in a programming language works.)

We see that the value bound to `degrees_F` did not change when we assigned 0.0 to `degrees_C`, even though `degrees_C` is used in the expression that calculates the Fahrenheit temperature. To summarize: assigning a new value to a variable does not change the value bound to any other variable.

A variable can appear on both sides of the = sign; for example,

```
>>> degrees_C = -15
>>> degrees_C
-15
>>> degrees_C = degrees_C + 21
```

As a mathematical equation, this does not make sense: how can the value of `degrees_C` be equal to that value plus 21? But in Python, = denotes assignment, so this statement means: add the value currently bound to `degrees_C` to the integer 21, then assign the result to `degrees_C`:

```
>>> degrees_C = degrees_C + 21
6
```

We have seen that literal values have types, and we have learned how to use Python's built-in `type` function to determine the type of a value. We can call type with variables as arguments; for example:

```
>>> degrees_C = 26.0
>>> type(degrees_C)
<class 'float'>
```

When type is called, Python uses the value bound to `degrees_C` as the function's argument. The value bound to `degrees_C` has type `float`, so the call to `type` returns `<class 'float'>`.

What happens when we assign -15 to `degrees_C`, then repeat the experiment?

```
>>> degrees_C = -15
>>> type(degrees_C)
<class 'int'>
```

The value bound to `degrees_C` has type `int`, so the call to `type` returns `<class 'int'>`.

This is known as *dynamic typing*: a variable can refer to values of different types at different points in a program's execution.