Activity 1

Sept. 30, 2010

Introduction to Python

Expressions

We're going to start off by typing in some *expressions*. Expressions are inputs that Python evaluates in order to produce a value, which it then outputs. Exampes of expressions are `6` , `1 + 3` , `( 46 + 14 ) / 3` , and `x + 1`.

1. Let's try giving Python a very simple expression to evaluate. Right now you should see the input prompt on the Python shell. The input prompt looks like this:

   ```
   >>>
   ```

2. Type `3 + 2` next to the input prompt and press Enter. This tells Python to evaluate the expression `3 + 2` and print the answer on the next line. You should now see the following in the shell:

   ```
   >>> 3 + 2
   5
   >>>
   ```

   Although Python does not care whether you put spaces in between the characters in an expression, it is a good habit to put a space between characters and mathematical operators (except parentheses). Python evaluates `3+2` and `3 + 2` the same way, but it is easier to read if you type `3 + 2`.

3. Python uses the standard set of mathematical operators +, - , *, and /, as well as parentheses for orders of operations. Practice evaluating a few expressions using each of the operators.

4. Try having Python evaluate `1 / 3`. What happens? While it may not seem like 3 and 3.0 are any different, there IS a difference in

Python. Numbers without a decimal point, such as 6 and -12, are called *integers* in Python; those that have a decimal, such as 4.5 and 3.0, are called *floating point numbers* or *floats*. The arithmetic for the two kinds of numbers is mostly identical, but there are a few differences. When evaluating expressions, Python rounds to integers unless the user (that's you) tells it to do otherwise. For example, Python evaluates `1 / 3` as 0, but evaluates `1.0 / 3.0` , `1 / 3.0` , and `1.0 / 3` as .33.

5. Evaluate `3.0 * (1.0 / 3.0)`. Since 1.0 and 3.0 are floats, the answer Python gives you will be the float 1.0. In addition, since floats are not rounded, we know that the answer is exact. If the answer had been 1, then we couldn't have known whether the answer had been rounded or was exact.

Assignment Statements

Besides expressions, the other type of input we can give Python are *statements*. Unlike expressions, statements are not evaluted to a specific value by Python, and therefore there is no output after a statement. An important kind of statement is an *assignment statement*. Assignment statements (also simply called *assignments*), instead of telling Python to evaluate an expression and output a value, tell Python to assign a value to a variable. Here is a simple example of an assignment:

```
>>> x = 10
```

The variable to which you are assigning a value always goes on the left side of the equals sign. In the `x = 10` statement above, x is the variable to be assigned a value and 10 is the value which it is being assigned.

1. Hit Enter after typing in x = 10. You should see the following:

   ```
   >>> x = 10
   >>>
   ```

   Python has now assigned the value 10 to the variable `x`. What happens now if you type in the expression `x` and hit Enter? Try evaluating the expressions `x + 5, x * 9,` and `100 / x`. Now assign the value 4 to the variable y and tell Python to evaluate x + y.

2. Python allows variables to be reassigned to a new value. Assign `x = 15`, then ask Python to evaluate x. Be careful when reassigning values, because once a variable is reassigned there is no way to retrieve the variable's original value (in this case 10).

3. Variables are extremely useful for storing values; they are the programming equivalent of cells in Excel. In addition, since variables can be more than just one letter, they allow users to give values readable and reasonable names. For example:

   ```
   pointTotal = 385.0
   numberOfTests = 4
   averageScore = pointTotal / numberOfTests
   ```

   This is an extremely clear and readable program that computes the average test score for a student using the variables `pointTotal`, `numberOfTests`, and `averageScore`.

4. Evaluate the expression `xy`. What's the value? (Recall that x is still 15 and y is still 4). The only way to perform multiplication is with `*`. xy is just a variable name that hasn't been assigned a value.

Lists

A *list* is a collection of items.

1. Let's start by creating a list of integers. Type the following into a shell:

   ```
   >>> mylist = [1, 2, 3]
   ```

   The assignment above works just like the other assignment statements we've used. The variable is `mylist` and the value it is assigned is the list `[1, 2, 3]`. Python knows that this is a list because of the square brackets, and distinguishes between the different items in the list by the commas. This list has three items - the integers 1, 2, and 3.

2. Tell Python to evaluate the simple expression mylist. This outputs the entire list. How do we access a single item in the list? Indexing! Lists in Python know about their contents, so we can write an expression

asking mylist about the value at the appropriate spot in the list. This is called *indexing*. Type in the expression `mylist[0]` and hit enter. Python gives you the first item in the list. List indices represent the positions of the list and start at 0, instead of 1. Ask Python to evaluate `mylist[1]` and `mylist[2]`. Now ask Python to evaluate `mylist[3]`. What happens?

3. We can modify lists by reassigning new values to the different indices in a list. Enter the following assignment:

   ```
   >>> mylist[2] = 4
   ```

   This reassigns the spot at `mylist[2]` to hold the value `4`. Tell Python to evaluate `mylist` and then `mylist[2]`.

4. We can also write an expression asking Python to output all the values within a certain range of indices in a list. This is done using the colon operator. Assign the list `[1,2,3,4]` to the variable `list1`. Now have Python evaluate the following expression:

   ```
   >>> list1[1:3]
   ```

   The colon operator tells Python to evaluate all the values between the indices on either side of the colon, including the value at the index on the left side of the colon, but *not including* the value at the index on the right side of the colon.

5. Now tell Python to evaluate `list1[:3]` and `list1[1:]`. If you don't specify a starting index Python starts at the beginning of the list. Likewise, if you don't specify an ending index, Python ends at the end of the list.

6. Lists can also be combined with other lists. Assign the variable `list2` to the list `[5, 6, 7]`, and then enter the following assignment:

   ```
   >>> list3 = list1 + list2
   ```

   Now ask Python to evaluate the expression `list3` to see what it does.

7. Try assigning the variable `list4` to the value `list3 + 8`. Python gives you an error message because you can only combine a list with another list. To add an 8 to the end of `list3`, we have to turn the 8 into a list. To do this we put square brackets around the 8. Enter the following assignment statement:

```
>>> list4 = list3 + [8]
```

Now tell Python to evaluate `list4`.

8. We can also put brackets around variables to add them to a list. For example:

```
>>> a = 5
>>> b = [1, 2, 3, 4]
>>> c = b + [a]
```

This adds the integer `a` onto the end of list `b` by making `a` into a list with only one item in it. Tell Python to evaluate the expression `c`.

9. Assign the following variables:

```
>>> list1 = [1, 2, 3]
>>> list2 = [4, 5, 6]
```

What happens if you assign `list[1:1]` to equal the value `list2`? Reassign `list1` back to `[1, 2, 3]`. Now assign `list1[1:2]` to equal `list2`. Take a second to make sure you understand what Python is doing. Finally, reassign `list1` back to `[1, 2, 3]`. Assign `list1[1]` to equal `list2`. How is this different from above?

10. Ask Python to evaluate the expression `len(list2)`. What does this represent? Do you remember what the highest index of `list2` is? Why are these numbers different?

Strings

In Python and in computer languages in general, a *string* is a sequence of characters. Examples of strings are ``Python'' or ``The Count counts everything.''. Strings are delimited by quotation marks. In Python a string can be assigned to a variable as the variable's value.

1. Enter the following assignments:

```
>>> string1 = "The Count"
>>> string2 = "counts everything."
```

Now ask Python to evaluate the expression string1.

2. Write an assignment combining (concatenating) `string1` and `string2` into the variable `string3`. Hint: how did we combine lists?

3. We can combine more than one string at a time by using multiple `+` operators. In addition, Python recognizes a blank space as a valid string character. Concatenate `string1` and `string2` again, making sure there is a space between the words `Count` and `counts`.

4. We can index into strings the same way we can with lists. Ask Python to evaluate the expressions `string1[5]` and `string1[6:10]`.