More Python Programming

Written by Dave Pawlowski, October 4, 2011

Statmements

Updating Variables

One of the most common tasks performed in programming is updating a variable, in which the new value of the variable depends on the old:

```
>>> x = x + 1
```

This is read: get the current value of x, add one, an then update x with the new value. If you try to update a variable that doesn't exist, you get an error, because Python evaluates the expression on the right side of the assignment operator before assigning the resulting value to the name on the left:

```
>>> x = x + 1
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'x' is not defined
```

Before you update a variable, you have to initialize it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Boolean values and expressions

When while statements are executed, the condition is tested to be True or False. These values are called **boolean values** (In Python, the capitalization is important). A **boolean expression** is an expression that evaluates to a boolean value. The operator == compares two values and produces a boolean value:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

In the first statement, the two operands are equal, so the expression evaluates to True; in the second statement, 5 is not equal to 6, so we get False.

The == operator is one of the comparison operators; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

Logical operators

There are three logical operators that you can use in Python: `and`, `or`, and `not`. The meaning of these operators is simliar to their meaning in English. For example, x $>$ 0 and x $<$ 10 is true only if x is greater than 0 and less than 10.

n % 2 == 0 or n % 3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

Finally, the not operator negates a boolean expression, so not(x $>$ y) is true if (x $>$ y) is false, that is, if x is less than or equal to y.

Conditional statments

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the `if` statement

If statements

We have already been introduced to the syntax of the `if` statement, but I'll mention it again here as a reminder. The simplest form looks like:

```
if x > 0:
   print "x is positive"
```

The boolean expression after the if statement is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens.

Alternatively, the if statement can have alternative execution, in which there are two possibilities and the condition determines which one gets executed:

```
if x % 2 == 0:
   print x, "is even"
else:
   print x, "is odd"
```

Since the condition must be true or false, exactly one of the alternatives will always be executed. Sometimes there are more than two possibilities, and thus more than two branches are required:

```
if x < y:
    print x, "is less than", y
elif x > y:
    print x, "is greater than", y
else:
    print x, "and", y, "are equal"
```

`elif` is an abbreviation of else if. Again, only one branch will be executed. There is no limit to the number of `elif` statements, but only a single (and optional) `else` statement is allowed, and it must be the last branch in the statement.

It is also possible to nest if-elif-else statements inside one another.

While statements

By now, you should realize that computers were made to do repetitive tasks. Repeated execution of a set of statements is called iteration. Because iteration is so common, there are several ways to do this in most languages, as you have seen. Python is no exception. The first iterative feature we'll look at is the `while` statement:

count.py

```
#!/usr/bin/python
n = 0
while n < 10:
    print n
    n = n+1
print "The End"
```

You can read the while statement as if it were in English: while n is less than 10, print the value of n and then add 1 to it. When you get to the end, print "The End". Note the indentation: everything that is executed during the while loop is indented. The print statement only occurs after the while loop has ended.

In python, the flow of this statement goes as such:

1. Evaluate the condition, yielding `True` or `False`
2. If the condition is false, exit the while statement and continue execution at the next state-

ment.

3. If the condition is true, execute each of the statements in the body and then go back to step 1.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite** loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, Lather, rinse, repeat, are an infinite loop.

In count.py, it is clear the the loop will eventually end since n starts at zero and gets larger each iteration; eventually we have to get to 10. In other cases, it isn't so obvious that the loop will terminate:

sequence.py

```
#!/usr/bin/python
n = 30
while n != 1:
    print n,
    if n % 2 == 0:      # n is even
        n = n / 2
    else:                   # n is odd
        n = n * 3 + 1
```

The condition for this loop is `n != 1`, or n is not equal to 1, so the loop will continue until n is 1, in which case the condition becomes false. In this example, the modulus operator, %, is used. The modulus works on integers and yields the remainder when the first operand is divided by the second.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches 1.

Abbreviated assignment

Above, we incremented a variable by one as part of a while loop. This is done rather frequently in programming, so Python has a few shortcuts for doing so:

```
>>> x = 0
>>> x += 1
>>> print x
```

should show you that x is now 1. The += 1 increments the variable by one. The increment variable does not have to be 1:

```
>>> x = 5
```

```
>>> x += 4
>>> print x
```

will show x being equal to 9. There are other abbreviations that may be handy:

```
>>> x -= 2
>>> x *= 5
>>> x /= 3
```

all do similar increment and assign operations.

Tables

One of the handy uses for loops is for creating tables:

table.py

```
#!/usr/bin/python
x = 1
while x < 13:
    print x, '\t', 2**x
    x += 1
```

This program prints two columns of data in an organized manner. '\t' represents a tab character. The backslash is used to 'escape' characters that are invisible, like newlines: '\n'.

Type conversion

Each type of Python variable comes with a built-in command that attempts to convert values of another type into that type. For example:

```
>>> int("32")
```

will try to convert the string "32" into the integer, 32.

There are similar functions for floats, strings and other types of variables:

```
>>> float(32)
>>> str(32)
>>> bool(1)
>>> bool(0)
>>> bool("Hi!")
>>> hex(124)
```

Are a few examples. Note the `bool` function: For numerical types like integers and floating-points, zero values are false and non-zero values are true. For strings, empty strings are false

and non-empty strings are true.