More Python Programming

Written by Dave Pawlowski, October 9, 2011

1 Advanced data structures

So far, you have met the basic types of variables in Python, floats, integers and strings. There are a few other types of variables that make Python a powerful programming language.

1.1 Lists

One of these variable types are called lists. A list is simply an array of values, where each value is identified by an index. The values that make up a list are called its elements. In a way, lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. In fact, most of the things that you can do with lists, you can do with individual strings. For this reason, both strings and lists fall under a class of Python variables referred to as sequences.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets:

```
>>> a = [1,2,3,4]
>>> print a
>>> b = [“red”,”green”,”blue”,”orange”]
>>> print b
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list dont have to be the same type. The following list contains a string, a float, an integer, and another list:

```
>>> c = [”hello”, 2.0, 5, [10, 20]]
```

A list within a list is said to be **nested**.

Finally, there is a special list that contains no elements. It is called the empty list, and is denoted [].

Like numeric 0 values and the empty string, the empty list is false in a boolean expression.

1.2 Accessing elements of a list

In order to get access to an element of a list, you use the bracket operator. Note that the first element indexed as 0.

```
>>> print a[0]
>>> print b[2]
```

You can access to individual elements of string the same way:

```
>>> str = 'Snow Patrol'
>>> print str[5]
```

will print 'P'.

If you use negative numbers, Python counts backwards from the end of the list:

```
>>> print c[-1]
```

will print the last element of the list, c, which is the nested list itself.

Python interprets slices of list elements a little differently than other languages:

```
>>> print c[1:2]
```

will print only the 2nd element of c. The operator [n:m] returns the part of the list from the n-th character to the m-th character, including the first but excluding the last. This behavior is counter intuitive; it makes more sense if you imagine the indices pointing between the elements.

If you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence. For example:

```
>>> fruit = "banana"
>>> print fruit[:3]
>>> print fruit[3:]
```

1.2.1 Common list functions

Python has several built-in functions that operate on strings. These can be used in Python as such:

```
>>> print func(listname)
```

where 'func' is one of the functions listed below, and 'listname' is a variable that contains a list. Most of these can be used on strings as well.

`len`: returns the length of a list
`in`: tests membership in a sequence (`''hello'' in c` returns true if the string "hello" is in the list `c`.
`range`: creates a list that contains consecutive integers:

```
>>> range(5)
```

creates the list, [0, 1, 2, 3]. Again, the indexing is slightly counter intuitive. `range` can take other, optional, arguments that specify the number to start the list with, and the number to increment the list by.

Finally, you can use operators on lists:

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = a+b
[1, 2, 3, 4, 5, 6]
```

The operator repeats a list a given number of times:

```
>>> a*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

There is much more that you can do with lists in Python, but this should get you well started.

1.3 Tuples

Tuples are similar to `lists`:

```
>>> tup = (2,4,8,10)
>>> nottup = [1,2,3,4,5]
```

Note that tuples are defined using parentheses instead of brackets. The difference is that tuples cannot be modified:

```
>>> tup[1] = 5
```

will return an error whereas

```
>>> nottup[1] = 5
>>> print nottup
[1, 5, 3, 4, 5]
```

is allowed. The behavior of tuples is the same as that of lists and strings, and operators and functions that work on the other variable types will also work on tuples.

Typically, a good reason to use a `tuple` would be for storing a list of data that should never change while a program is run.

1.4 Dictionaries

The other data type that you may find useful is a `dictionary`. These Dictionaries are a different kind of compound type. They are Pythons built-in mapping type. They map keys, which can be any immutable type, to values, which can be any type.

One way to create a dictionary is to start with the empty dictionary and add key-value pairs. The empty dictionary is denoted :

```
>>> fruit2col = {}
>>> fruit2col['apple'] = 'red'
>>> fruit2col['pear'] = 'green'
>>> fruit2col['watermelon'] = 'orange'
```

The first assignment creates a dictionary named fruit2col; the other assignments add new key-value pairs to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print fruit2col
```

The key-value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

Similarly, you can look up a value associated with a key:

```
>>> print fruit2col['pear']
```

Once a dictionary is defined and populated, you can remove a key-value pair from it using `del`:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
>>> del inventory['pears']
```

Dictionaries have several built–in methods and can be extremely useful, though maybe not for doing

2 Iteration using for loops

Like all programming languages, Python is capable of traversing a set of values using `for` loops. In order to use a `for` loop, you need to have a `sequence` defined to iterate over:

```
>>> n = range(10)
>>> for i in n:
...         print i
```

will print out the values from 0 to 9. This type of task is easily done using a while loop. However, it is possible to use the `for` loop in a more useful manner, i.e.:

```
>>> names = ['Dave','Lindsay','Ben','Rooster']
>>> for i in names:
...         print i
```

Again, you could do this using a while loop, but you would have to add at least a couple lines of code. Again, a string is a sequence just like a list, so you can do a similar operation on a string:

```
>>> name = 'Dave Pawlowski'
>>> for i in name:
...          print i
```

3 Functions

One of the most fundamental concepts in the context of procedural programming is the idea of the function. A function is a named sequence of statements that performs a desired operation. This operation is specified in a function definition. Functions allow your code to be organized and compartmentalized. They allow you to reuse segments of code in multiple programs. They allow you to test segments of code without having to run an entire program. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code. Creating a new function can make a program smaller by eliminating repetitive code.

In Python, the syntax for a function definition is:

```
def NAME( List of Parameters ):
    Statements
```

You can make up any names you want for the functions you create, except that you cant use a

name that is a Python keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function.

There can be any number of statements inside the function, but they have to be indented from the def. In the examples in this book, we will use the standard indentation of four spaces. Function definitions consist of a header, which begins with a keyword and ends with a colon, and a body, which consists of one or more Python statements, each indented the same amount from the header. In Python, this is considered a compound statement. You've met other compound statements: `if` and `while`.

In a function definition, the keyword in the header is `def`, which is followed by the name of the function and a list of parameters enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters. In either case, the parentheses are required.

Open up a file called functions.py. We will define a few functions in this file, and them use them. First, define a function called hello:

```
def hello():
   print "hello"
```

This function simply prints the word "hello" when called. The empty parentheses indicate that the function itself does not require any parameters.

Defining a new function does not make the function run. To do that we need a function call. Function calls contain the name of the function being executed followed by a list of values, called arguments, which are assigned to the parameters in the function definition. Our first example have an empty parameter list, so the function calls do not take any arguments. Add the following lines to your file. Notice that the parentheses are required in the function call:

```
print 'I am calling the function now: '
hello()
print 'All done!'
```

A few things: (1) You can call a function repeatedly. This means that you can include function calls inside while and for loops. (2) You can have one function call another function.

Lets do another example:

```
def plusone(x):
    return x + 1
```

In this case, the function plusone is defined as requiring a single parameter. The `return` statement tells Python to assign the result of the function to be `x + 1`. This is called using:

```
x = 10
x = plusone(x)
```

It is expected that x is defined before the function call. Otherwise, you will get an error. The exception to this rule is if you assign default values:

```
def multiprint(txt, n=5 ):
    i = 0
    while i < n:
        print txt
        i += 1

sometext = 'Dave'
multiprint(sometext)
```

In this case, the value n is optional. If you do not include it in the function call, the value defaults to 5 in the program. Alternatively, you may set it to an other value and it will override the default. Optional parameters must follow non-optional parameters in the function definition.

Another thing that you may have noticed is that when you call a function, the arguments don't have to have the same name as the parameters in the function definition.

Flow of Execution

When you run a program, execution always begins at the first statement of the program. Statements are executed one at a time, from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Although it is not common, you can define one function inside another. In this case, the inner definition isnt executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note that this means that functions have to be defined before you actually call them.

Scope

When you create a variable inside a function, it only exists inside the function, and you cannot use it outside. For example:

```
def cat(str1, str2):
    string = str1 + str2
    print string

string1 = 'Michigan football '
string2 = 'is Awesome!'
cat(string1,string2)
```

When the function ends, the variable `string` is destroyed. Note that if you add the line:

```
print string
```

you will get an error. `string` is considered to be *local* to the `cat` function. Similarly, parameters are considered to be local to their functions. If you tried to use the variables `str1` and `str2`, you will get an error.