

Introduction

In the previous tutorial, we examined some fundamental concepts regarding writing Python scripts. We explored PythonWin as a useful integrated development environment (IDE) for writing Python scripts, and we reviewed some basic data types (numbers, strings, lists, tuples, and dictionaries) and a few standard Python functions available for these data types.

In this tutorial, we build on these concepts, examining how to craft Python scripts to get things done. We look at methods for controlling the flow of scripts (looping and conditional statements) as well as basic means for handling script input and output. And we also look at the basic tools for handling errors within scripts.

We'll introduce ourselves to these new concepts by reviewing a set of simple examples. You are encouraged to play around with the syntax and logic of these scripting tools. And in our next set of tutorials we will pull this all into context as we assemble a script that implements all these techniques in order to execute a task.

Learning objectives:

- [Iteration through a collection of objects using **for** loops](#)
- [Iterating until a condition is met using **while** loops](#)
- [The **range\(\)** function](#)
- [Controlling execution with **if** statements](#)
- [Exiting loops using **break** and **continue**](#)
- [Reading files with file objects](#)
- [Writing files with file objects](#)
- [Simple input using the **raw_input\(\)** function](#)
- [Handling script errors with **try** and **except**](#)

Resources:


- Python documentation: <http://docs.python.org/index.html>
- Python WIKI: <http://wiki.python.org/moin/>
- "Learn Python the Hard Way" (e-book) <http://learnpythonthehardway.org/book/>
- Lynda.com course on Python (Duke only):
<http://www.lynda.com/Python-3-tutorials/essential-training/62226-2.html>

1. Iterating through items in a collection using a *for* loop

Scripts are excellent tools for running repetitive tasks. We can invest a bit of time to "train" the machine what to do, and what to do it on, and then we just hit "run" and go play while the work gets done.

Repetitive tasks are usually accomplished using loops, of which there are two flavors: a *for* loop and a *while* loop. First we'll look at a *for* loop, which in plain language is telling the script to repeat a task or set of tasks for each item in a collection (i.e. a list or tuple). Let's look at an example of how a *for* loop is used in Python.

Exercise 1: For loops

1. Open up the script *ForLoopExample.py* in PythonWin. This script creates a list of fruit and then uses a *for* loop to iterate through each item in the fruit list and prints a short message to the screen. Examine and run the script.
2. Next, run the script *one line at a time* using the debugger. You may need to re-enable your debugger menu bar. From the menu bar use the  icon to run one line at a time. While the script is in the middle of the loop (i.e. after one of the days has been printed), type the following at the interactive prompt

```
>> print fruit
```

This enables you to see the value of the "fruit" variable in the middle of the script's execution. *Knowing what the value of a given variable at a given point in the execution of a script can be quite useful in writing (and debugging) more complex scripts.*

-End Exercise 2-

Not much to it, but there are a few concepts worth reviewing.

- First is that in the variable specified in the *for* statement, "fruit" in our example, is updated at every pass of the loop, and we can use this variable within the loop.
- And second is the significance of **indentation** in Python: the group of lines that are indented under the *for* statement are what gets run within the loop. And when the loop is complete (meaning it has run through each item in the specified collection), the script moves to the next "dedented" line.
→ Indentation is how Python groups statements.

2. Iterating until a condition is met using a *while* loop

For loops can be quite useful, but what if you don't have a nice discrete list of objects to loop through? For example, what if you wanted to repeat a calculation until the result met a certain criteria? To do that, we use a *while* loop, which in plain language is telling the script to continue executing a set of commands until some condition is met. Let's dive in with an example.

Exercise 2: While loops

3. Open up the script *WhileLoopExample.py* in PythonWin. This script calculates the area of a circle with radius 'r'. Within the loop, the radius is increase by 1 and the loop stops when the area of the circle is greater than 1000.
4. Run the script and see if you can follow the logic. What is the largest radius of a circle with an area less than 1000 units?
5. Play with the script and run it. Change initial value of r to something higher than 20 (line 9). Change the where clause from `area < 1000` to `area < 5000` in line 21. How do these affect execution?
6. Indent the last line in the script (line 14) and run it. What role does indentation play in Python?

-End Exercise 1-

Let's pause and review a few important concepts demonstrated in the above example.

First is the necessity that the value being evaluated in a *while* loop ("area" in our case) needs to exist before the *while* loop starts (line 12). If 'area' was not given a value in line 10, Python would not know what 'area' is and the loop would fail. Always declare the variable being evaluated in a while loop prior to the while statement.

And second, note again the significance of **indentation** in Python. Only the lines of code that are indented below the while loop are run within the loop. When the loop completes, Python moves to the first dedented line (line 14).

Another important point about *while* loops is the possibility of **infinite loop**. If you forget to add lines 14 and 15 in your loop, the *area* value will never change and the while loop will continue to run until some drastic action is taken. If this occurs (go ahead and try it by commenting out line 14), your best option is to right click the Python icon in your system tray (lower right corner of your screen), and select "Break into running code". If that fails, try closing PythonWin or use the Windows Task Manager to end the Python.exe process.

3. The *range()* function and its use in *for* loops

The `range()` function is useful if you want to iterate over a sequence of numbers - or just create a variable that includes a sequence of numbers. Let's examine how it's implemented...

Exercise 3: Looping with the range() function

1. First, use the Python help statement in the interactive window to view a summary of the `range()` function:

```
>>> help(range)
```

2. Next, give the `range()` function a try, again in the interactive window:

```
>>> print range(10)
```

What is the first value in the list? The last?

```
>>> print range(1,10)
```

What did this change?

```
>>> print range(1,10,2)
```

What did this change?

How would you use the range function to create a list from 0 to 20 including only even numbers?

3. To see how the range function can be used in a for loop, open the *RangeFunctionExample.py* script in PythonWin.

-End Exercise 3-

The `range()` function is probably more useful than it is exciting. But there you have it.

4. Controlling execution with *if* statements

Many scripts are not just linear sequences of commands; rather, they evaluate a condition - whether a variable has a certain value, whether a file exists, etc. - and execute one set of commands if that's true

and perhaps another if it's false. In Python, controlling the execution of scripts in this manner is done with `if` statements.

An `if` statement can be used alone - to run a set of code only if the condition being evaluated is true. Or it can be used with `else` to offer an alternative course of action to run if the statement is false. In more complex situations the `elif` statement, short for "else if", can be used to evaluate multiple conditions in a sequence. Let's look at a few examples.

Exercise 4: If statements

1. Open the *IfExample.py* script in PythonWin. Before you run it, what do you expect to be printed to the interactive window? Run the script. Were you right?
2. Add the following commands below line 14. Be sure the first line you add is indented to the same position as the `if` statement in line 13.

```
else:
    print "I don't like " + fruit
```

3. Now add the following between the `if` statement and the `else` statement:

```
elif fruit == "grapes":
    print "I don't mind" + fruit
```

-End Exercise 4-

The logic of the `if` statement is pretty straightforward. You can have as many `elif` statements that you need. If nothing meets the condition of the `if` statement or any `elif` statements, then Python will run whatever you might have indented under the `else` statement. Or if you omit the `else` statement, Python will simply continue to the next statement aligned to the same indentation as the original if statement.

Another important note here is the difference between '=' and '=='. A single equals sign is used in assigning a variable value. A double equals sign is used in evaluating logical equivalence.

5. Controlling loops using *break* and *continue*

Now that we understand conditional execution with if statements, let's return to looping and examine two useful loop control statements: `break` and `continue`.

Exercise 5: Using *break* and *continue* within loops

1. Open the *BreakExample.py* script in PythonWin and run it. This script loops through a list of 10 numbers and calculates the square root of the number. However, if the number is less than zero, the loop ends, since we can't calculate the square root of a negative number.
2. Change `break` in line 14 to `continue`. Run the script. What is the difference between `break` and `continue`? Running the script one line at a time may help you discover the key difference.

-End Exercise 5-

The `break` and `continue` statements allow us to control execution within a loop. If `break` is found within a loop, the loop is discontinued. All other commands that within (indented) the loop are skipped and Python goes to the next line at the same indentation level as the loop. The `continue` statement

simply skips any further commands at the same indentation level as where it occurs, and the loop goes to the next item. These can be quite handy in the right circumstances.

6. Reading text files using Python's *file object*.

Let's move on from controlling the flow of scripts to another important concept: reading data into a script rather than having all variables be assigned values from within the script itself. We'll also explore how to write output to a file. The key concept here is Python's *file object*.

In the exercise below, we will read values in from the same NWIS file we used in the first Databases tutorial, namely NWIS data collected from the EnoNearDurham gage.

Exercise 6. Reading data from a text file using the file object

1. Open the *ReadDataExample.py* script in PythonWin.
2. Ensure that the *EnoNearDurham.txt* file is in the same folder as the script and run the script one line at a time until you get into the while loop. Then run the remainder of the script.

- End Exercise 6-

Now let's review the key concepts in this script.

First let's review line 13 where the file object is created. The Python `open()` function has two parameters. The first is the name of the file to be opened and the second is the mode in which it is opened. The file we are opening is the *EnoNearDurham.txt* file, the value of the `gageDataFile` variable. And here we are opening it in "read-only" mode, which means we cannot alter the contents of the file. There are two other modes, "write" and "append" mode, which we will talk about shortly. These would be accessed by changing the 'r' to a 'w' or 'a', respectively.

Once the file object is created, we can access the file's contents in our script. You can type `help(fileObj)` to view a quick reference on the file object, or `dir(fileObj)` to get a listing of what properties and functions are available for this object. The most likely functions you'd use in accessing the contents of a text file are the `readlines()` and `readline()` function.

The `readlines()` function reads all the lines in a text file, putting the contents into a list object where each line in the text file becomes an item in the list. Feel free to try it:

Exercise 6a: Reading lines into a list with readlines()

1. Open the file again (copy and paste line 13 into the interactive window and then run it)

```
>>> fileObj = open(gageDataFile, 'r')
```

2. Read the contents into a list object named `lineList`:

```
>>> lineList = fileObj.readlines()
```

3. Print the 3rd line to the screen

```
>>> print lineList[3]
```

-End Exercise 6a-

The `readlines()` function is useful because we have instant access to any line in the text file. We could also loop through the lines quite easily using a for loop (`for each lineString in lineList:`).

The potential issue with the `readlines()` function is that, if you have a large text file, you need to store the entire contents in the computer's memory. If you have a large text file (or are unsure how large it might be), you may be better off with the `readline()` function instead.

The `readline()` function, which is used in our script, reads a single line from the text file at a time, and then moves a file pointer, or cursor, to the next line in the file. Line 16 in our script reads the first line in the text file and places the contents in the "lineString" variable. If we executed line 16 again, the second line would be read and placed in the lineString variable. If there was no second line in the text file, the `readline()` function would set the value of the lineString variable to an empty string.

The `readline()` function is often used within a while loop to enable processing of the entire contents, as demonstrated in our example script. (Again, it's important to move to the next line in the file or you'll get an infinite loop!) When the entire contents have been read, the lineString variable is set to an empty string, which successfully ends while loop.

Knowledge check:

See if you can insert an if statement within the while loop of the above script so that only lines not starting with "#" are printed to the screen...

Last but not least, take note of line 28. This closes the file object, which is just the same as closing the file when it's open in Notepad or Word. Closing the file releases Python's handle on the file allowing other applications to access it without any restriction.

7. Writing to text files using Python's file object

So, we now know how to read data from a text file, but how might we *write* data to a text file? This can be quite useful as, so far, all our scripts have been able to do is print ephemeral output to the screen. Writing output to a text file is not that much more complex than reading a file. Take a look at the example script to see how it's done.

Exercise 7: Writing output to a text file.

1. Open the *WriteDataExample.py* script. This script builds off of the previous script where we read lines in from the *EnoNearDurham.txt* file, but instead of printing lines to the screen, we write them to an output file. Furthermore, it adds a filter to what lines get written. The end result is a text file that includes only the lines needed for quick import into Access, like we did in Section 2 of the class.
2. Run script one line at a time until you hit line 21. Be sure you have a grasp on what each line does. The comments should help, but if you don't understand, be sure to ask.
3. Now run line 21, where the new file is created, take a look at the S:\Scripting folder in Windows. You should indeed see that a new file has been created.
4. Run one iteration of lines 25-29. The logic here is that we look at each line in the line list one at a time. If the first 4 characters in the line are "USGS", it's a data line, and we write the line to the output file (line 29). If it's not a data line (i.e. a comment or a header line), we *continue* over line 29 and the line does not get written to the output.

5. Have a look at *WriteDataExample2.py*. Here I've added some code so that only records listing discharge > 6.0cfs are written to the output file. With just a few lines, we've run a data query without having to import the records into MS Access!

-End Exercise 7-

8. Simple script input using the *raw_input()* function

When PythonWin is installed, it provides additional functionality to the base installation of Python. One of these functions is a simple popup window that you can raise when your script is executed that asks the user to supply an input. This is done with the `raw_input()` function. In later tutorials we will see more sophisticated and robust means for interacting with users (and other applications), but implementing `raw_input()` can be a useful technique at times.

Exercise 8 - Using the *raw_input()* function

1. Open the *RawInputExample.py* script and run it.
2. Manipulate the code so that the prompts shown when the script is run are different.
3. What happens if you don't enter a number at the second prompt?

-End Exercise 8-

The `raw_input()` function is fairly straightforward. You supply a string as the function's argument, and it becomes the prompt that appears with the string. And whatever the users types before hitting enter becomes the value that is passed from the function.

You should be reminded, however, that this function is tied to PythonWin and will not work on all flavors of Python. Again, we will explore better ways to supply information interactively into a Python script. Still, this function can be quite handy at times.

9. Handling script errors with *try* and *except*

Errors occur in scripts. When a statement is expecting something that either doesn't exist or perhaps does exist but in the wrong format, then Python is unable to continue, often spewing angry red text in the interactive window. The previous exercise demonstrated that very thing: a nasty error occurs when users enter something that's not a number at the second prompt. So what can we do about this?

The answer is to trap and handle errors so that, at the very least, the script exits gracefully and perhaps even deals with the error and moves on. In Python, errors are handled with `try` and `except` statements. Let's dive in and have a look how these work:

Exercise 8 - Handling errors with *try* and *except*

1. Open the *HandlingErrorsExample.py* script. This script calculates the square root of a user supplied number. Run the script with a valid number (e.g. 10) to ensure it runs as expected.
2. Next, run the script again, but supply something other than a number at the prompt, e.g. "A". Try this again, running the script one line at a time to see the sequence of what happens.

3. Finally, run the script one last time, but enter a negative number. An error will since you can't calculate the square of a negative number. How is this error handled?

-End Exercise 9-

In the above script, line 10 is a `try` statement, and it along with the indented lines 11-18 underneath it comprise a "try clause". This try clause allows Python to handle any error that occurs within it. If an error does occur, Python doesn't instantly bail and blast a bunch of error messages at the user. Instead, Python looks for an `except` statement and jumps down there.

Lines 20-22 comprise the "except clause". These statements are run *only* if an error occurs within the `try` clause immediately preceding it. In the format presented in this script, the except statement produces an *exception object* as the variable "e". We don't need to go into much more detail here other than "e" now likely contains information on what went wrong. Our except clause prints a general error message and then any information pertaining to the error that occurred, which often can be instructive.

The net result is still that processing ceases, but it ends gracefully and possibly helpfully.

Generally speaking, a try-except statement can be used to wrap entire programs or just particular portions of code to trap and identify errors. If an error occurs within the try statement, an exception is raised, and the code under the except statement is then executed. Using a simple except statement is the most basic form of error handling.

Handling errors can grow to be quite sophisticated, more sophisticated than we need to know here. Knowing how to handle different types of errors is important in writing elaborate Python scripts, but for most of what you'll likely be doing, simple try-except statements as shown here will be suitable.

Closing remarks

Python is certainly a lot more than what is covered here, but master of these topics will move you a long way towards writing scripts that get things done. It's important that you leave this tutorial with a firm grasp of the logic and format of how each technique is implemented: while loops, for loops, if statements, the range function, break and continue statements, file objects, and error trapping.

In the next tutorial we will use these fundamental techniques to assemble a script that, I hope, puts them into context as to how they might be used.