More Python Programming: Input/output, modules, and error handling

Written by Dave Pawlowski, October 12, 2011

1 Input/output

So far, the only way that your programs are able to get input is by taking information from the keyboard. Similarly, all of your output has been printed to the screen. But suppose you want to read from or print to a file? Python is capable of doing this very easily.

1.1 Reading from a file

In order to read from a file, first you have to tell Python what file to read from, and open the file. This is done using the open() function. Grab data.dat and put it in your local directory. You've worked with this file before in Gnuplot. Now, open it in python:

```
>>> f = open('data.dat','r')
```

open() takes at least one argument, the name of the file to be opened. In this case, I've added a second argument, 'r'. r is the mode; this tells Python that the file should only be used to read. If you don't include the second argument, 'r' is the default. Alternatively, you can specify 'w' for writing, in which case a new file will be created by Python, or if the file exists, it will be erased and then recreated. Also, 'r+' opens the file for both reading and writing. Finially, if you want to append to a file (add data to the end of a file), use 'a'.

Now that you opened a file, let's see what Python has done.

```
>>> print f
```

This should tell you something about the variable f. In fact, when you use open(), Python creates a file object (remember that whole everything in Python is an object stuff?). f is an object, and printing it tells you that you've opened 'data.dat' and that the file is only readable.

Now, in order to read from the file, we need to use the read *method* of f.

```
>>> f.read()
```

That sould give you something pretty ugly. The read method reads the entire file. Those \n characters that you see are newline characters, they show where new lines start in your file.

Reading a file this way isn't particularly useful, as its a pain to parse all the data, and it is not efficient if you open a large file, since it loads it all into memory.

If you try to read the file again:

```
>>> f.read()
```

you will get an empty string. You've reached the end of the file and there is nothing more to read. So, lets start over. First, close the file:

```
>>> f.close()
```

and open it again

```
>>> f=open('data.dat','r')
```

This time, instead of reading the entire file, lets just read it line by line

```
>>> f.readline()
```

You should see that only the first line of the file was read this time. This way of reading a file tends to be easier to deal with, especially if you are going to be populating lists with the data.

In a program, you will want to save the result of the read in a variable. This is done with a simple assignment operation.

```
>>> temp = f.readline()
```

If you try

```
>>> print temp
```

you will see that you read the next line of the file (note that the \n character is interpreted as a newline by Python).

Python offers up a third, and even easier way to read data that is great if you want to fill lists; use a `for` loop!

```
>>> for line in f:
...     print line
...
```

This will loop over all lines in the file, but read them one at a time. Python automatically figures out when you reach the end of the file and exits the `for` loop. You could accomplish the same thing using f.readline and a `while` loop, but this is easier. You should be able to see that you can quickly populate lists using this technique by replacing the print line with an assignment.

We're done reading this file for now, so be sure to close it.

```
>>> f.close()
```

1.2 Writing to a file

Another method of a file object is write. Open up another file:

```
>>> g = open('somedata.txt','w')
```

Now we can use the write method to write to the file:

```
>>> g.write('Hello there!\n')
>>> g.write('This is some text...\n')
```

You need to include the \n character if you want to start a new line. if you want to write something other than a string, you have to convert it to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> g.write(s+'\n')
```

Again, remember to close the file when you are done.

```
>>> g.close()
```

Once the file is closed, you can look at it in a text editor.

1.3 Reading and writing strings

You should have noticed that, used in this way, Python only reads and writes strings. This means that there is a lot of converting from int to string or string to float, etc. when you are using input and output. In general, this is fine. But if you are going to be always using Python to read and write data, then there is an even easier way to handle this: **Pickle**.

2 Modules

Pickle is a standard module that comes with all Python distributions. There are many such modules, **math**, **cmath**, **random**, and **sys** are a few other examples that you may find useful. If you need something in Python that isn't built in, a good place to start looking for it is http://docs.python.org/library/.

Modules are Python's way of storing definitions, statements, functions, etc. intended for use in Python programs that aren't available when you simply startup Python. The main reasons that modules are used are: (1) they make it easy to organize functions and classes in a logical manner and (2) they make it easy to package up your code so that you can access it from any place in your file system or even share it with the world. All sorts of people have created modules that do various things. A quick search of the web will show you that there are endless numbers of modules available.

In order to use a module, you first have to load it:

```
>>> import pickle
```

will load the pickle module. Now you have access to all the functions and other things that are defined in the pickle module. So, for example if you wanted to write the contents of a tuple to a file, without first turning it into a string, you could do so:

```
>>> atuple = ('the answer', 42)
>>> g = open('newfile.txt','w')
>>> pickle.dump(atuple,g)
>>> g.close()
```

and that's it. Conversely can read the tuple from the file into Python memory, **as a tuple**, using:

```
>>> g = open('newfile.txt','r')
>>> x = pickle.load(g)
>>> type(x)
```

In fact, you can add any Python variable type using pickle.dump, close the file, and load it again using pickle.load. Try closing the file, adding several lines, and then reading from the file again:

```
>>> g.close()
>>> g = open('newfile.txt','a')
>>> pickle.dump([10,20,30],g)
>>> pickle.dump('Hello!',g)
>>> pickle.dump(50.28355,g)
>>> g.close()
```

Open the file and read it in Python using the load function and you should see that each time you do a load, you get the next variable that you wrote, as expected.

If you use emacs to open 'newfile.txt' you will see that Python adds a few characters that the Pickle module uses to interpret the data. What this means is that these files are only useful to Python. It would be difficult to use Pickle to produce a file that is to be read by other software, like Gnuplot. In order to do that, you have to stick with the built in read and write methods of file objects.

Back to Modules. When we loaded pickle, you were able to use the functions by first specifying the module that they are contained in, then the function name. This can become annoying at times, to constantly write things like pickle.dump and pickle.load... You know you want to use the dump function, so why can't you just say dump?!

You can. One way is to reassign the function to a new variable:

```
>>> dump = pickle.dump
```

This allows you to use dump just as you would pickle.dump. Another more convienent way is when you import the module to begin with. Lets use that math module as an example:

```
>>> from math import *
```

This tells Python to import everything in the math module. Doing things this way means you don't need to type math.sqrt to use the square root funtion of the math module. You just type

```
>>> sqrt(16)
```

Using from, it is possible to only import certain things from modules:

```
>>> from os import path
```

path is a module inside the **os** module, and has its own functions that allow you to manipulate filenames. So after executing the above command, you have access to several functions, such as

```
>>> path.abspath('data.dat')
```

(assuming data.dat is in your current directory), which will return the absolute path to the file. Similarly, if you only wanted the **relpath** function from the **os.path** module, you could use

```
>>> from os.path import relpath
```

The reason that there are several ways to import modules and their functions, is that there are so many functions, that it is inevitable that some have the same name. By using the module name when calling a function, you can have several functions that have the same name, but are different because they are from different modules.

Again, if you need to find a module that does something that isn't automatically loaded by Python, start with http://docs.python.org/library/.

Most of the typical math functions that you need to do scientific programming can be found in the **math** module. This includes trig functions, rounding functions, log functions, and constants like pi and e.

3 Error Handling

If you do something wrong in Python, you get an ugly error that gives you a somewhat, but not overly meaningful, error message. It is up to you to think about what errors your code may come across given normal usage. A common source of error is when you ask the user for input. Usually, the input is used in a type specific way. For example, you may want to add together two numbers that are taken as input from the user, using the raw_input function.

One way to ensure that you get a value that can be converted to an integer is to make use of Python's ability to handle errors (or exceptions, as they may be called in Python).

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "No good! That was not a valid number. Try again..."
...
```

The try except statement in Python is useful for error handling. There are several types of errors in Python, so Python allows you to specify different actions based on the type of error that you encounter. In this example, the error that is tested for is ValueError.

The best way to figure out which exceptions to test for is to think about the possible errors that may occur and run them in Python. If you try:

```
>>> int('test')
```

you will see that Python throws a ValueError exception. There are many types of exceptions in Python. See http://docs.python.org/library/exceptions.html for more.