**Lecture 4 (week 3.1)**

Tuesday 20th 2016

Input/Output (a.k.a. I/O)

At some point, a program will read data and/or output data. There are different type of data I/O available in python. Today we will be concerned with text based I/O, i.e. strings.

There are other ways for a program to exchange data with a user some of them we will covered later in this course (e.g. binary and structured data sets).

Start with sections 14.1 to 14.3 and then 14.5 in <http://greenteapress.com/thinkpython2/html/thinkpython2015.html>

Formatting in Python

When the user reads/writes to a text file, the information is a string with a certain format. Python’s formatter has an *old* and *new* syntax. Both are working in Python 3.

The old format uses **%** as the formatter operator. The syntax is a specific output format template you want (between quotes ‘ ‘), followed by % then followed by what the variable(s) you want to write, between parenthesis ( ):

**In [#]: ‘myformattemplate’ % (myparameters)**

Imagine you want to write the two strings “one” and “two” as one string, with 4 spaces between them

**In [10]: '%s %s' % ('one', 'two')**

**Out[10]: 'one two'**

The **%s** in the format operator can be any type of output taken from a predefined list of options (here **%s** means string type format, below are examples of other possible formats, e.g. integer, signed or unsigned, float, etc…).

The new format uses a method of the string **str** class, namely the format method **str.format()**. To do the same thing as above with the new format method, the syntax is:

**In [18]: '{} {}'.format('one', 'two')**

**Out[18]: 'one two’**

The format operator can contain extra text, which you might want to do for some situations. For instance, with the old **%** operator:

**In [25]: import math**

**In [26]: '%i is less then %f but more than %-d' % (1,math.pi,-1)**

**Out[26]: '1 is less then 3.141593 but more than -1'**

now the same thing with the **format** string method:

**In [30]: '{} is less then {} but more than {}'.format(1,math.pi,-1)**

**Out[30]: '1 is less then 3.141592653589793 but more than -1'**

In order to appreciate the infinite possible variations to format your output, there is nothing better than practice as much as you can. The links below will give you specific examples to help you learn how the string formatting works in Python. with a nice comparison between the old and new way:

Basic formatting: <https://pyformat.info/#simple>

Padding and aligning strings: <https://pyformat.info/#string_pad_align>

Numbers: <https://pyformat.info/#number>

Truncating long strings: <https://pyformat.info/#string_truncating>

Padding numbers: <https://pyformat.info/#number_padding>

Signed numbers: <https://pyformat.info/#number_sign>

This is a technical description of the formatting techniques for both the told and new format methods:

<http://www.python-course.eu/python3_formatted_output.php>

You should read up to “Using dictionaries in “format””, where it becomes too advanced for this course (for now).

File I/O

With Python, you can open an existing file, create a new one, and then write, read and append to it. Python has very specific commands to do these things and if you try to do something not allowed (e.g. write to a file where you don’t have permission), you will get an error.

File Input refers to the read mode: you open the file and read its content.

File Output refers to writing in the file. In the append mode, you add things at the end of an existing file. In the write mode, opening the file immediately destroys any existing file content!

For efficiency, Python doesn't write data to a file immediately. It puts the data into a buffer, and writes when the buffer is full. When you are finished writing to a file, you must close the file to "flush" the last buffer into the file. If you forget to close the file when done writing and your script ends or you close the python session, your file is not written!

Writing to a file

To open a file, you use the **open** Python built-in function (remember <https://docs.python.org/2/library/functions.html>). Imagine you open a new file **mynewfile.txt** and want to write to it, type:

**In [39]: f=open(‘mynewfile.txt','w')**

(note: simple or double quotes are both valid syntax). Now you have just created a new object, **f**, which is a “file object” for writing to file **mynewfile.txt**. As for any Python object, you can apply *methods* to it. You can use the write method:

**In [1]: f=open('mynewfile.txt','w')**

**In [2]: f.write("This is some text.")**

**Out[2]: 18**

**In [3]: f.write("More text on the same line.\n")**

**Out[3]: 28**

**In [4]: f.write("And this is some more text.\n not on the same line")**

**Out[4]: 49**

**In [5]: f.close()**

You have just written in **mynewfile.txt**. Note the use of line ending character **\n**.

You can access file object methods with **help(f)**, where **f** is your chosen file object name.

Executing the above commands from a script will not generate an output.

You can open a file with mixed mode (e.g. **‘rw’**).

The **file.write()** file method can only write strings, not numbers (unlike **print()** when writing on the Python prompt). To write numbers in a file use the **str(number)** function which converts number to string:

**In [1]: x = 1**

**In [2]: y = 2.2**

**In [3]: f = open("numbers.txt", "w")**

**In [4]: f.write(str(x) + " " + str(y) + "\n")**

**Out[4]: 6**

**In [5]: f.close()**

Now is when the Python formatting becomes very handy!

**In [6]: x = 1**

**In [7]: y = 2.2**

**In [8]: f2 = open("numbers2.txt", "w")**

**In [9]: f2.write("%3d %5.2f\n" % (x, y) )**

**Out[9]: 10**

**In [10]: f2.write("%3d %5.2f\n" % (15, 33.333) )**

**Out[10]: 10**

**In [11]: f2.close()**

with the new format method:

**In [12]: x = 1**

**In [13]: y = 2.2**

**In [14]: f3 = open("numbers3.txt", "w")**

**In [15]: f3.write("{0:3d} {1:5.2f}\n".format(x, y) )**

**Out[15]: 10**

**In [16]: f3.write("{0:3d} {1:5.2f}\n".format(15, 33.333) )**

**Out[16]: 10**

**In [17]: f3.close()**

Reading from a file

You first open the file in read-mode. The basic file-reading commands give you text-strings, either the whole file, or one line at a time. You have to decode the text-string into numbers or whatever, and store the results somehow, usually into lists.

You should close the file when you are done, but there's not much harm if you forget.

Some tips:

You need to know the structure of a file before you can make code to read it.

You often need to skip over some initial lines.

It's common to have numbers arranged in columns, but you need to know which columns contain what.

If you don't know how many valid data lines there are, your program has to detect when the data ends, without crashing at the end of the file.

Consider the file **ludodata.txt** containing the following:

**N x y xerror yerror**

**1 1.24 3.14 0.012 0.52**

**2 1.26 3.89 0.012 0.52**

**3 1.28 4.04 0.012 0.53**

**4 1.30 4.33 0.012 0.53**

**5 1.33 4.67 0.012 0.54**

**6 1.35 5.08 0.012 0.54**

**Data from 10:15:39 on September 12 2016**

With the following example, you will see how to read a file, line by line and extract numerical values when present:

**In [1]: f=open('ludodata.txt','r')**

**In [2]: data=f.readline()**

**In [3]: data**

**Out[3]: 'N x y xerror yerror\n'**

**In [4]: data=f.readline()**

**In [5]: data**

**Out[5]: '1 1.24 3.14 0.012 0.52\n'**

**In [6]: data.split()**

**Out[6]: ['1', '1.24', '3.14', '0.012', '0.52']**

**In [7]: cols=data.split()**

**In [8]: cols[0]**

**Out[8]: ‘1'**

**In [10]: float(cols[0])**

**Out[10]: 1.0**

**In [11]: float(cols[1])**

**Out[11]: 1.24**

**In [13]: f.close()**

**In [14]: data=f.readline()**

**---------------------------------------------------------------------------**

**ValueError Traceback (most recent call last)**

**<ipython-input-14-479dc886b26e> in <module>()**

**----> 1 data=f.readline()**

**ValueError: I/O operation on closed file.**

The last command shows that once a file is closed, its file object (**f** in that case) is flushed from memory as well.

If you wanted to create an array of numbers with the values inside file **ludodata.txt**, you would have to do this line by line using the string **split** method and the string to float conversion function **float()**. This would be extremely inefficient with long files! We haven’t learned yet how to manipulate lists of number and loops, but this would be the way to make this short and quick and you will learn this very soon in class. For now, know that it is possible to make this much less painful when using the **readline()** method with *lists* and *loop*.

Reading from a file with the numpy **loadtxt()** method

A very useful method of numpy is **loadtxt()** which allows you to read a text file containing an array with one command line and the output is immediately stored in a numpy array. It is very convenient. loadtxt() can read any type of text file, even those with strings, however, how to use loadtxt() for mixed data type is a bit too advanced for now ,but you can start practicing with the example given at the end of this paragraph. So we will focus on how to use loadtxt() to read files that look like this **ludoarray.txt** text file:

**# N x y xerror yerror**

**1 1.24 3.14 0.012 0.52**

**2 1.26 3.89 0.012 0.52**

**3 1.28 4.04 0.012 0.53**

**4 1.30 4.33 0.012 0.53**

**5 1.33 4.67 0.012 0.54**

**6 1.35 5.08 0.012 0.54**

**# Data from 10:15:39 on September 12 2016**

Then, using **loadtxt()**:

**In [1]: import numpy as np**

**In [2]: data=np.loadtxt('ludoarray.txt')**

**In [3]: data**

**Out[3]:**

**array([[ 1. , 1.24 , 3.14 , 0.012, 0.52 ],**

**[ 2. , 1.26 , 3.89 , 0.012, 0.52 ],**

**[ 3. , 1.28 , 4.04 , 0.012, 0.53 ],**

**[ 4. , 1.3 , 4.33 , 0.012, 0.53 ],**

**[ 5. , 1.33 , 4.67 , 0.012, 0.54 ],**

**[ 6. , 1.35 , 5.08 , 0.012, 0.54 ]])**

Note that lines starting with a “#” are considered comments and are ignored by **loadtxt()** by default (but you can change this).

You can also do a selective reading of the array using the usecols argument in loadtxt():

**In [1]: import numpy as np**

**In [2]: data=np.loadtxt('ludoarray.txt',usecols=(1,3))**

**In [3]: data**

**Out[3]:**

**array([[ 1.24 , 0.012],**

**[ 1.26 , 0.012],**

**[ 1.28 , 0.012],**

**[ 1.3 , 0.012],**

**[ 1.33 , 0.012],**

**[ 1.35 , 0.012]])**

If you want a more advanced pick at **loadtxt()** you can look at the numpy **loadtxt()** page:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>

and then a specific example with mixed types:

<http://scipython.com/book/chapter-6-numpy/examples/using-numpys-loadtxt-method/>

Writing to a file with the numpy **savetxt()** method

Similarly, you can write arrays in files in text format using the **savetxt()** numpy method.

**In [1]: import numpy as np**

**In [2]: data=np.loadtxt('ludoarray.txt')**

**In [3]: data**

**Out[3]:**

**array([[ 1. , 1.24 , 3.14 , 0.012, 0.52 ],**

**[ 2. , 1.26 , 3.89 , 0.012, 0.52 ],**

**[ 3. , 1.28 , 4.04 , 0.012, 0.53 ],**

**[ 4. , 1.3 , 4.33 , 0.012, 0.53 ],**

**[ 5. , 1.33 , 4.67 , 0.012, 0.54 ],**

**[ 6. , 1.35 , 5.08 , 0.012, 0.54 ]])**

**In [4]: np.savetxt(‘ludonewarray.txt',data)**

Note that **savetxt()** can be used with different option which handle the format of the saved text:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html>