Introduction To Python

Week 4: Formatting & File I/O

Dr. Jim Lupo
Asst Dir Computational Enablement
LSU Center for Computation & Technology

Formatting

- Controlling output formatting to make things look *purdy*.
- Remember doing this?

```
>>> x = 5./6.; y = 6./7.
>>> print x, y
0.833333333333 0.857142857143
```

- What if you want this?

```
>>> x = 5./6.; y = 6./7.
>>> print x, y
0.83  8.571e-01
```

String Templates

- String templates define how values are to be displayed.
- Use when printing, or just creating strings in general.
- Syntax here: '%m.nf %m.nf'%(v1, v2)
  - Example of using a tuple!
- 1 or more format specifiers in a string.
- Matching number of values

```
>>> print '%3.2f  %9.3e'%(x,y)
0.83  8.571e-01
```

Deconstruction

- % - start of format specifier. Use %% if you want to print a %.
- m - total width of output field.
- n - digits after decimal point
- 'f' - floating point conversion
- 'e' - engineering or scientific conversion.

Many different types of conversions.

Some Obvious Conversions

Conversion Type	Specifier	Example
String	s	'%s World'%('Hello')
Float	f	'watts: %8.3f'%(power)
Integer	d	'No of dwarves: %d'%(7)
Integer	i	(Same as %d)

Sneaky Behavior of %s

- %s actually means 'print string representation of object - assumes object has a method to generate a string value!
- Previous example was a string.
- What of a boolean?

```
>>> x = True
>>> print '%s'%x
True
>>> print '%.1s'%x
T
>>> x = 10
>>> print '%s'%x
10
```

Special Conversions

Conversion Type	Specifier	Example
Signed Octal	o	'%o'%(8)
Signed Hex	x	'%x'%(pointer)
Signed Hex	X	'%X'%(pointer)
Single character	c	'%c'%65

Some Exercises

- Try these. Any surprises?

```
   x = 90
1. print '%c'%x
2. print '%x'%x
3. print '%f'%x
4. print '%o'%x
5. print '%4o'%x
6. print '%5o'%x
7. print '%8.3E'%x
```

Rascally Alternate Forms

- Hex numbers were missing 0x
- Octal numbers were missing leading 0.
- Left up to user to add, or use special alternate form: %#

```
>>> print '%#X'%x
0X5A
>>> print '%#x'%x
0x5a
>>>
```

Boring Tables!

- Lots of things can be done.
- Boring to try and go through all.
- Know they exist.
- Know where they are documented.
- Learn new ones when needed!

Exercise

- Print the numbers from 99 to 110 (inclusive) in fields 4 spaces wide.
- Repeat, but 0 fill - that is leading zeros instead of spaces.

- What do your program stanzas look like?

A Solution

Part 1:

```
for i in range(99,111) :
    print '%4d'%i
```

Part 2:

```
for i in range(99,111) :
    print '%04d'%i
```

Building a Report Header

- A summary report requires a header that reads:

  "20150618 Report: 14 data points"
- The number of data points should be easy.
- How do we handle the date portion?

  First excursion into realm of ***modules***!

Modules?

- Predefined collections of python code that provide all sorts of functionality.
- Technically they are classes that provide data and methods to manipulate the data.
- Modules are added using the ***import*** command.

Many To Chose From

- Visit:

  https://docs.python.org/2/py-modindex.html

- This lists the ***standard*** modules distributed with Python
- There are many others!
- ***time*** module supports all sorts of operations from displaying time and date in different formats to time arithmetic.

Basic Usage

- Here is a straight-forward way of getting a date string:

```
>>> import time
>>> date = time.strftime('%Y%m%d')
>>> print date
20150618
>>>
```

- There are some 26 different format directives (%x) controlling types of data to display - calendar dates, times-of-day, etc.

Simple Header Generation

```
print '%s Report: %d data points' % \
           ( time.strftime('%Y%m%d'), np )
```

Or

```
header = '%s Report: %d data points'
print header % ( time.strftime('%Y%m%d'), np )
```

Standard I/O Revisited

- raw_input() - read strings from STDIN
- print - write strings to STDOUT
- These are files, but are limited to the keyboard, terminal, and the magic of redirection.
- Need something more general purpose.

Simple 3-Line File (3-lines.txt)

```
Hi There!
This is Line 2.
Last line.
```

Type this in, or get session-4.zip from the web site.

Simple File Reader (simpleio1.py)

```
f = open('3-lines.txt')      # Create file object; Needs file name or path
x = f.readline()             # Read next line method
print x
print x.strip()
y = f.readline()             # Read next line method
print y
z = f.readline()             # Read next line method
print z
w = f.readline()             # Read next line method
print w
print len(w)
f.close()                    # Discard file object
```

simpleio1.py Output

```
Hi There!

Hi There!
This is Line 2.

Last line.



0
```

Included end-of-line!

Another File Method (simpleio2.py)

```
f = open('3-lines.txt')
x = f.readlines()
print x
print len(x)
for i in range(len(x)) :
    print 'Line %d: "%s"'%(i,x[i].strip())
f.close()
```

Convert file to list of lines.

Loop over lines to process.

simpleio2.py Output

```
['Hi There!\n', 'This is Line 2.\n', 'Last line.\n']
3
Line 0: "Hi There!"
Line 1: "This is Line 2."
Line 2: "Last line."
```

Write a File (simpleio3.py)

```
f = open('3-lines.txt')
x = f.readlines()
f.close()
f = open('foo.txt','w')
for i in range(len(x)) :
    f.write('Line %d: "%s"'%(i,x[i].strip().upper()))
f.close()
```

Default: Read the file

Write the file

Line 0: "HI THERE!"Line 1: "THIS IS LINE 2."Line 2: "LAST LINE."

Explicit NewLine or End-of-Line

```
f.write('Line %d: "%s"\n'%(i,x[i].strip().upper()))
```

↑

String escape character.
\n means newline, or end-of-line.

Other String Escape Characters

Sequence	Meaning
\<newline>	Ignore new line.
\\	\
\'	Single quote
\"	Double quote
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Verticle tab
\DDD	Character matching octal value
\xDD	Character matching hex value
\other	Other character

More on **open()**

- Formal syntax:
  open(name [, mode [, buffering]])
- name .. path name of file.
- mode .. how to access the file.
- buffering .. how to handle data from file.

Open Modes

Mode String	Meaning
r	Read - error if file does not exist.
w	Write - error if file exists.
a	Append - file may or may not exist.
r+	Open existing file for read and write - error if file does not exist.
w+	Open for read and write, but existing file is truncated, or empty file created.
a+	Open for read and write at end of existing file, or empty file created.
<*above*>b	Treat as binary - newlines ignored.

Buffering

- Controls how data is read in from device and presented to program.
- System reads from device and places in buffer. Program really reads from buffer. When writing, data goes out to device when buffer is filled.

Value	Meaning
< 0	Use operating system default. Terminals may be different than files. 8192 bytes per access common.
0	Unbuffered - access device on each read or write - may be *sloooow*.
= 1	Buffered one line at a time.
> 0	Define buffer size, but operating system may round the value.

Data Analysis Problem

- Data file ***output.dat*** contains *state* variable values:

```
TimeStep Time(fs) Temperature(k) Pressure(Pa) Energy(erg)
10 12.372 300.0 1.5 5.0e+5
20 24.201 301.0 1.9 6.0e+5
30 39.005 305.0 2.5 7.0e+5
40 38.897 305.1 2.6 7.1e+5
50 35.221 305.1 2.7 7.2e+5
60 46.876 305.1 2.7 7.2e+5
70 57.001 305.1 2.8 7.3e+5
80 68.222 304.9 2.6 7.1e+5
90 78.235 305.0 2.7 7.0e+5
100 88.321 305.0 2.7 7.0e+5
```

Program Specs

- Report the label, minimum, maximum, and average value of state variable.
- Compute the average time per timestep.
- Expected results:

```
Average Timestep: 0.88321
Temperature(k) : 300.0 305.1 304.13
Pressure(Pa) : 1.5 2.8 2.47
Energy(erg) : 500000.0 730000.0 679000.0
```

analyze.py

```
# Open the input file and ingest all the data.

f = open('output.dat')
x = f.readlines()
f.close()

# Grab column labels from first line.

labels = x[0].split()

# Define indices for min, max, and summation in the
value lists.

min = 0
max = 1
sum = 2

# Define indices for the state variables in the values
list.

temperature = 0
pressure = 1
energy = 2

# Define how many indices to skip to get data from
the input lines.

offset = 2

# Initialize the values lists.

values = [[1.e+37,-1.e+37, 0.0],\
         [1.e+37,-1.e+37, 0.0],\
         [1.e+37,-1.e+37, 0.0]]
```

```
# Now read all the data from the remaining lines.

for i in range(1,len(x)) :
   n = x[i].split()
   steps = float(n[0])
   time = float(n[1])
   values[temperature][sum] += float(n[offset + temperature])
   values[pressure][sum] += float(n[offset + pressure])
   values[energy][sum] += float(n[offset + energy])
   for j in range(3) :
      if values[j][min] > float(n[offset + j]) :
         values[j][min] = float(n[offset + j])
      if values[j][max] < float(n[offset + j]) :
         values[j][max] = float(n[offset + j])

# Display results on terminal.

print 'Average Timestep:', time / steps
for j in range(3) :
    print labels[offset + j], ':', values[j][min], \
          values[j][max], values[j][sum] / float(len(x)-1.0)

# Now generate output file.

f = open('foo.txt','w')
f.write( 'Average Timestep: %e\n'%(time/steps) )
for j in range(3) :
    f.write( '%s : %e  %e  %e\n' % (labels[offset + j], \
            values[j][min], values[j][max], \
            values[j][sum] / float(len(x)-1.0) ) )
f.close()
```

CSV Files

- CSV stands for ***comma separate variable*** file.
- More generally, a flat, tabular, character delimited data file.
- Our example used white space:

```
10  12.372  300.0  1.5  5.0e+5
```

- Traditional would look like (i.e. EXCEL csv output):

```
10,12.372,300.0,1.5,5.0e+5
```

- Any character could be used:

```
10:12.372:300.0:1.5:5.0e+5
```

- Processing could get messy (i.e. quoting separator)

CSV Module

- If it's a common need, there is likely a module for it, and there is!

```
import csv
```

- Plenty of methods look at.
- Redo ***analyze.py*** with it:

  analyze-csv.py

analyze-csv.py

```
import csv

# Define indices for min, max, and summation in the value lists.

min = 0
max = 1
sum = 2

# Define indices for the state variables in the values list.

temperature = 0
pressure = 1
energy = 2

# Define how many indices to skip to get data from the input lines.

offset = 2

# Initialize the values lists.

values = [[1.e+37,-1.e+37, 0.0],\
          [1.e+37,-1.e+37, 0.0],\
          [1.e+37,-1.e+37, 0.0]]

# Open the input file and prepare CSV reader.

f = open('output.dat','rb')
x = csv.reader(f,delimiter=' ')
lineno = 0

# Iterate through the lines.
```

```
for l in x :

  lineno += 1
  if lineno == 1 :
    # Grab column labels from first line.
    labels = l
  else :
    # Read data from the remaining lines.
    steps = float(l[0])
    time = float(l[1])
    values[temperature][sum] += float(l[offset + temperature])
    values[pressure][sum] += float(l[offset + pressure])
    values[energy][sum] += float(l[offset + energy])
    for j in range(3) :
      if values[j][min] > float(l[offset + j]) :
        values[j][min] = float(l[offset + j])
      if values[j][max] < float(l[offset + j]) :
        values[j][max] = float(l[offset + j])

f.close()

# Display results on terminal.

print 'Average Timestep:', time / steps
for j in range(3) :
   print labels[offset + j], ':', values[j][min], \
       values[j][max], values[j][sum] / float(lineno-1.0)

# Now generate output file.

f = open('foo.txt','w')
f.write( 'Average Timestep: %e\n'%(time/steps) )
for j in range(3) :
   f.write( '%s : %e  %e  %e\n' % (labels[offset + j], \
         values[j][min], values[j][max], \
         values[j][sum] / float(lineno-1.0) ) )
f.close()
```

Structured Data I/O

- Only textual data so far: numbers to strings and back again.
- How to deal with complex data? Consider:

```
x = {'alpha':[1,'a',2], 'beta':[2,'b',3]}
print x
```

```
{'alpha': [1, 'a', 2], 'beta': [2, 'b', 3]}
```

- Output looks like just a string. How to write it out and read a dictionary back in?

The Problem, Illustrated

nonpickled.py

```
x = {'alpha':[1,'a',2], 'beta':[2,'b',3]}
y = {'delta':[3,'c',4], 'gamma':[4,'d',5]}
f = open('nonpickled.dat','w')
f.write('%s\n%s\n'%(x,y))
f.close()
f = open('nonpickled.dat')
z = f.readlines()
f.close()
print z
```

```
["{'alpha': [1, 'a', 2], 'beta': [2, 'b', 3]}\n",
             "{'gamma': [4, 'd', 5], 'delta': [3, 'c', 4]}\n"]
```

???

How do we get x and y back?

Use Pickle

- Pickle encodes objects so they can be written and read as single entities.
- Uses a binary format to preserve all bits in any variable type.
- Requires the ***pickle*** module.

pickled.py

```
import pickle
w = {'alpha':[1,'a',2], 'beta':[2,'b',3]}
x = {'delta':[3,'c',4], 'gamma':[4,'d',5]}
f = open('pickled.dat','wb')
pickle.dump(w,f)
pickle.dump(x,f)
f.close()
f = open('pickled.dat','rb')
y = pickle.load(f)
z = pickle.load(f)
f.close()
print 'w :', w
print 'y :', y
print 'x :', x
print 'z :', z
```

Write in binary mode.

Write 1 object at a time.

Read in binary mode.

Read back in same order.

Sanity check.

pickled.dat?

```
(dp0
S'alpha'
p1
(lp2
I1
aS'a'
p3
aF2.133
asS'beta'
p4
(lp5
I2
aS'b'
p6
aF3.244
as.(dp0
```

```
S'gamma'
p1
(lp2
I4
aS'd'
p3
aF5.466
asS'delta'
p4
(lp5
I3
aS'c'
p6
aF4.355
as.
```