Introduction To Python

Week 5: Finish up I/O
On To Functions

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

XSEDE
Extreme Science and Engineering Discovery Environment

I/O - Continued

- Revisit the programs ***analyze.py*** and ***analyze-csv.py***.
- Then a more Pythonesque approach.

XSEDE
Extreme Science and Engineering Discovery Environment

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()
```

XSEDE
Extreme Science and Engineering
Discovery Environment

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()
```

analyze-dict.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()

names = ['temperature', 'pressure', 'energy']
values = {
   'temperature':{'idx':2,'min':1.e+37,'max':-1.e+37,'sum':0.0},
   'pressure':{'idx':3,'min':1.e+37,'max':-1.e+37,'sum':0.0},
   'energy':{'idx':4,'min':1.e+37,'max':-1.e+37,'sum':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[values['temperature']['idx']])
   values['pressure']['sum'] \
     += float(n[values['pressure']['idx']])
   values['energy']['sum'] \
     += float(n[values['energy']['idx']])
   for l in names :
       if values[l]['min'] > float(n[values[l]['idx']]) :
           values[l]['min'] = float(n[values[l]['idx']])
       if values[l]['max'] < float(n[values[l]['idx']]) :
           values[l]['max'] = float(n[values[l]['idx']])

# Display results on terminal.

print 'Average Timestep:', time / steps
for j in names :
    print labels[values[j]['idx']], ':', 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 names :
    f.write( '%s : %e  %e  %e\n' % \
            (labels[values[j]['idx']], \
             values[j]['min'], values[j]['max'], \
             values[j]['sum'] / float(len(x)-1.0) ) )
f.close()
```

XSEDE
Extreme Science and Engineering Discovery Environment

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 out and read back a dictionary?

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?

XSEDE
Extreme Science and Engineering Discovery Environment

The Hard Way

- Read the lines.
- Play Python interpreter.
- Hope you get it right!

Use Pickle (A Module)

- Provides methods to encodes objects so they can be written as single entities.
- Provides methods to read and decode single entities, recovering the original object.

pickled.py

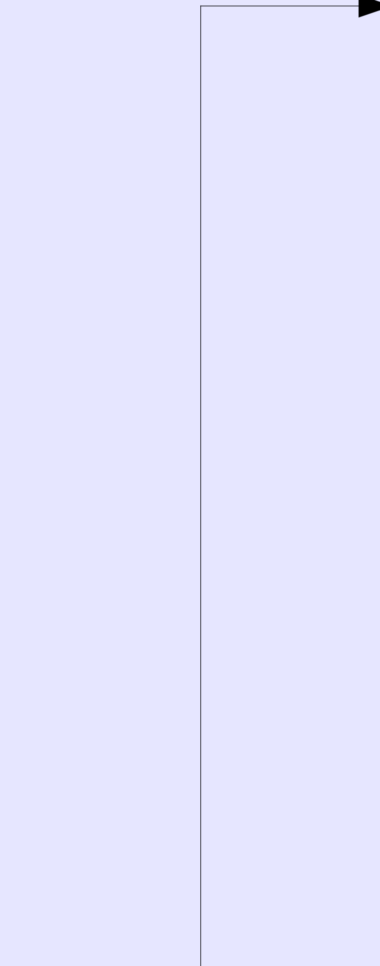
```
import pickle
w = {'alpha':[1,'a',2], 'beta':[2,'b',3]}
x = {'delta':[3,'c',4], 'gamma':[4,'d',5]}
f = open('pickled.dat','wb')
pickle.dump(w,f)
pickle.dump(x,f)
f.close()
f = open('pickled.dat','rb')
y = pickle.load(f)
z = pickle.load(f)
f.close()
print 'w :', w
print 'y :', y
print 'x :', x
print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

- Write in binary mode.
- Write 1 object at a time.
- Read in binary mode.
- Read back in same order.
- Sanity check.

XSEDE
Extreme Science and Engineering Discovery Environment

pickled.dat Contents


```
(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.
```

Aside on Use of ***import***

- Style: import pickle
  - Use method load: pickle.load()
- Style: import pickle as foofoo
  - Use method load: foofoo.load()
- Style: from pickle import load
  - Use method load: load()
- Style: from pickle import load as foo
  - Use method load: foo()
- Style: from pickle import *
  - Use any method by name.

XSEDE
Extreme Science and Engineering Discovery Environment

Math Functions

```
>>> from math import *
>>> print 'There are', degrees(1.0), 'degrees in 1 radian.'
There are 57.2957795131 degrees in 1 radian.
>>> print 'There are', radians(45.0), 'radians in 45.0 degrees.'
There are 0.785398163397 radians in 45 degrees
>>> print pi, e
3.14159265359 2.71828182846
>>> print log(pi), log(pi,e), log(pi,10)
1.14472988585 1.14472988585 0.497149872694
```

XSEDE
Extreme Science and Engineering Discovery Environment

Function Lingo

x = log(pi,e)

- The function, **log**, is called with 2 arguments, **pi** and **e** (the argument list).
- The function **result** is returned as the value of the expression and assigned to the variable **x**.
- Here, the arguments are called the **positional** arguments - they are order dependent.

Write A Celsius to Fahrenheit Function?

$^{o}F = ^{o}C * 9 / 5 + 32.$

Define a Function

- The syntax to create a function is:

```
def fname( arglist ) :
    statement block
```

- A 'Hello' function example:

```
def hello( a ) :
    return 'Hello, %s!'%(a)
```

```
>>> x = hello('Bob')
>>> print x
Hello, Bob!
```

XSEDE
Extreme Science and Engineering Discovery Environment

Function Deconstruction

```
def hello( a ) :
    return 'Hello, %s!'%(a)
```

- **hello** - the function name.
- **a** - one argument in the argument list (dummy argument). Gets treated like a variable inside the function.
  - Need one actual argument for each dummy argument
- **return** - sets value returned by function.
- Statement block here is only 1 line long. Can be as long as needed.

Define C2F Function

```
def c2f( C ):
    F = C * 9. / 5. + 32.0
    return F
```

Could return the temperature in Rankin and Kelvin as well:

```
def c2fr( C ):
    F = C * 9. / 5. + 32.0
    return F, F+459.67, C+273.15
```

More Complex Functions

- Basic statistics of a list of numbers might include:
  - mean - the numerical average value.
  - median - mid-point value of the list.
  - range - minimum and maximum values.
- Put several functions in one file.

Example - **functions.py**

```
def average( list ) :
    """ Find average value of a list of numbers."""
    sum = 0.0
    for x in list :
        sum += x
    return sum / len(list)

def minmax( list ) :
    """ Find the min and max."""
    min = 1.e+37
    max = -1.e+37
    for x in list :
        if x < min :
            min = x
        if x > max :
            max = x
    return min, max

def median( list ) :
    list.sort()
    if len(list) % 2 :
        m = list[len(list)/2]
    else :
        m = (list[len(list)/2 - 1]+list[len(list)/2] ) / 2.0
    return m
```

```
if __name__ == '__main__' :
    foo = [14,8,7,4,9,5,6,1,3,2]
    print foo
    print 'Average', average( foo )
    print 'Min & Max', minmax( foo )
    print 'Median', median( foo )
    print foo
```

Made Reusable

- Note what was added below the last function:

```
if __name__ == '__main__' :
    foo = [14,8,7,4,9,5,6,1,3,2]
    print foo
    print 'Average', average( foo )
    print 'Min & Max', minmax( foo )
    print 'Median', median( foo )
    print foo
```

XSEDE
Extreme Science and Engineering Discovery Environment

Put In Python Load Path

- From IDLE, look at:

    File -> Path Browser

  Should show the directory IDLE was started in, or where last script was run.
- On Linux / MacOS, look at:

    $PYTHONPATH

  This is a variable you can modify to add locations to search.

Load Custom Files

- Use the **import** command:

    import functions

- Will look first for **functions.pyc**
- If it finds functions.py, it will generate **functions.pyc**

  pyc files are pre-interpreted, or byte code compiled, files. Make it much more efficient to execute. Basically happens to every script as it's executed.

Treat As Any Other Module

- Methods:

  f unctions.average()

  functions.median()

  functions.minmax()

- We're almost at the point of creating a class of our own - but as we'll see, it's a bit more complex then just adding some data.

Default Argument Values

- Functions can be defined with default values some or all arguments.
- Revisit 'Hello' function example:

```
def hello( a='Bill' ) :
    return 'Hello, %s!'%(a)
```

```
>>> x = hello('Bob')
>>> print x
Hello, Bob!
>>> x = hello()
>>> print x
Hello, Bill!
```

Using Named Arguments

```
def example( a=1, b=2, c=3 ) :
    statement block

x = example( b=42, a = 97 )
```

- Names must match, but not the order of appearance.
- Unused argument names should have defaults

XSEDE
Extreme Science and Engineering Discovery Environment

Unpacking Returns

```
>>> def x() :
      return 1, 2, 3

>>> print x()
(1, 2, 3)
>>> a, b, c = x()
>>> print a, b, c
1 2 3
>>> def y() :
      return [1, 2, 3]

>>> print y()
[1, 2, 3]
>>> a, b, c = y()
>>> print a, b, c
1 2 3
>>>
```

tuple → (1, 2, 3)
values → 1 2 3
list → [1, 2, 3]
values → 1 2 3

```
>>> def z() :
      return {'x':1,'y':2,'z':3}

>>> print z()
{'y': 2, 'x': 1, 'z': 3}
>>> a, b, c = z()
>>> print a, b, c
y x z
>>> print z()[a]
2
>>>
```

dictionary → {'y': 2, 'x': 1, 'z': 3}
keys → y x z
value → 2

XSEDE
Extreme Science and Engineering Discovery Environment

Exercise

- A statistic left out of the example functions (average, minmax, median) is the ***mode***.
- The mode is the count of the value that occurs most frequently, or ***none*** (technically not 0, since 0 is a value) if no value is repeated.
- Create a function that, given a list of numbers, reports the mode.
- **Do not modify the original list.**

XSEDE
Extreme Science and Engineering Discovery Environment

Functions and Recursion

- Python functions may be recursive - that is, a function may call itself.
- The hard part about recursion is to decide when to stop!
- Classic case is computing factorials:

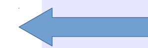
  n! = n * n-1 * . . . * 2 * 1

  0! = 1! = 1

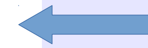
XSEDE
Extreme Science and Engineering Discovery Environment

Factorials

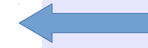
```
def factorial(n) :
    if n < 0 :
        return None
    if n <= 1 :
        return 1
    else :
        return n * factorial( n - 1 )

if __name__ == '__main__' :
    print factorial( 6 )
```



Error check



Stops recursion



Recursive call

Challenge Function

- sin(x) can be estimated with a series:

$$\sin(x) = \sum_{0}^{\infty} \frac{(-1)^k x^{1+2k}}{(1+2k)!}$$

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

- For x on the order of 1, the first 4 terms are usually good enough.
- Compare results with the math module sin().
- Use our factorial function, or the one in math.

XSEDE
Extreme Science and Engineering Discovery Environment