ESCI 386 – Scientific Programming, Analysis and Visualization with Python

Lesson 6 - File IO

Opening/Closing Files

- A file is opened for reading using the open statement

  f = open(*file_name*, ‘r’)

  – This returns a file object, in this case named f. We could have named it whatever we wanted.

  – The ‘r’ specifies the mode of the file be read only.

  – The different possible modes are

Mode	Meaning	Mode	Meaning
‘r’	Read only (text file)	‘rb’	Read only (binary file)
‘w’	Write only (text file)	‘wb’	Write only (binary file)
‘r+’ or ‘w+’	Read and write (text file)	‘rb+’ or ‘wb+’	Read and write (binary file)
‘a’	Append (text file)	‘ab’	Append (binary file)

Opening/Closing Files

- An ASCII file is opened for reading using the open statement

  f = open(*file_name*, ‘r’)

  - This returns a file object, in this case named f. We could have named it whatever we wanted.
  - The ‘r’ specifies the mode of the file be read only.
  - To open a file for writing we would use ‘w’ instead of ‘r’.
  - Opening a file with ‘a’ will open the file for writing and append data to the end of the file.
  - To open a file for both reading and writing we use either ‘r+’ or ‘w+’.

- You should always close a file when you are done with it. This is done with

  f.close()

Automatically Closing Files

- Python has a shorthand for opening a file, manipulating its contents, and then automatically closing it.
- The syntax is

```
with open(filename, ‘r’) as f:
    [code statements within code block]
```

- This opens the file and gives the file object the name f.
- The file will automatically close when the code block completes, without having to explicitely close it.

Interactive File Selection

- The code below allows interactive selection of file names.

```
import Tkinter as tk
from tkFileDialog import askopenfilename as pickfile
window = tk.Tk()   # Create a window
window.withdraw()  # Hide the window
filename = pickfile(multiple=False)
window.quit()      # quit the window
```

Interactive File Selection (cont.)

- The full path and name of the selected file will be stored as a string in the variable filename, which can then be used to open and manipulate the file.
  - Even on Windows machines the path names will use Linux style forward slashes ‘/’ rather than Windows style back slashes ‘\’.

- Multiple file names can also be selected by setting the multiple keyword to True.
  - If multiple files are selected then filename will be a single string containing all the filenames separated by white space.
  - On Windows machines where files themselves may have white spaces, if any of the filenames have white spaces within them then their filenames are contained in curly braces.

Interactive Directory Selection

- Directories can also be selected interactively.

```
import Tkinter as tk
from tkFileDialog import askdirectory as pickdir
window = tk.Tk()   # Create a window
window.withdraw()  # Hide the window
dirname = pickdir(mustexist=True)
window.quit()      # quit the window
```

Reading from Text Files

- The simplest way to sequentially read all the lines in a file an manipulate them is to simply iterate over the file object. For example, to read the lines of a file and print them to the screen we would use

```
f = open('datafile.txt', 'r')
for line in f:
    print(line)
```

Reading from Text Files

- We can also read a single line at a time using the readline() method.

```
line = f.readline()
```

- If we want to read all the lines in file and place them into a list, with each line being a separate element of the list, we can use the readlines() method

```
lines = f.readlines()
```

  - lines[0] would then contain the entire first line of the file as a string, lines[1] would contain the next line, etc.

Writing Text to Files

- Writing to text files is accomplished with the write() or writelines() methods.
  - The write() method writes a string to the file.
  - The writelines() method writes a list of strings to file.

Writing Text to Files (Example)

```
f = open('myfile.txt', 'w')
f.write('The quick brown fox')
f.write(' jumped over the\n')
f.write('lazy dog.')
f.close()
```

- Results in the contents of myfile.txt

```
The quick brown fox jumped over the
lazy dog.
```

Writing Text to Files

- Both the write() and writelines() methods require strings as the input data type.

- To write numerical data to a file using these methods you first have to convert them to strings.

Reading and Writing CSV Files

- In many data files the values are separated by commas, and these files are known as *comma-separated values* files, or *CSV* files.

- Python includes a csv module that has functions and methods for easily reading and writing CSV files.

Reading and Writing CSV Files

- To read CSV file titled ‘myfile.csv’ we first open the file as usual, and then create an instance of a reader object. The reader object is an iterable object, that can be iterated over the lines in the file.

- IMPORTANT: When opening a file for csv reader or writer, it should be opened as a binary file, using either ‘rb’ or ‘wb’ as the mode.
  - This prevent an extra blank lines being inserted in the output file.

Example Reading CSV File

```
import csv
f = open('myfile.csv', 'rb')  # NOTE:  Used 'rb'
r = csv.reader(f)  # creates reader object
for i, line in enumerate(r):
    if i == 0:
        continue   #  skips header row
    n, rho, mass = line
    volume = float(mass)/float(rho)
    print(n, volume)
f.close()
```

Example Writing CSV File

```
import csv
f = open('myfile-write.csv', 'wb')  # NOTE:  Used 'wb'
w = csv.writer(f)
data = [['Element', 'Atomic Number', 'Molar Mass'],\
        ['Helium', '1', '1.008'],\
        ['Aluminum', '13', '26.98']]
w.writerows(data)
f.close()
```

The csv Module is Flexible

- The csv module can be used with delimiters other than commas.

- To do this, we set the delimiter keyword as shown below:

```
r = csv.reader(f, delimiter = ‘;’)  # semicolon delimited file
r = csv.reader(f, delimiter = ‘ ‘)  # space delimited file
```

Using the with Statement

- The *with* statement is a shorthand way of always making certain that your files are closed when you are done with them.

- The with statement is used as follows

```
with open(filename, ‘r’) as f:
    code block that uses the file object f
```

- When the code block is exited, the file object *f* will be automatically closed

with Statement Example

```
filename = ‘my_input_file’
f = open(filename, ‘r’):
data = f.readlines()
f.close()
for line in data:
    print(line)
```

File closure is automatic using the ‘with’ statement

```
filename = ‘my_input_file’
with open(filename, ‘r’) as f:
    data = f.readlines()
for line in data:
    print(line)
```

Writing and Reading Numpy Arrays to Text Files

- Numpy Arrays can be written to text files using
  - *a*.tofile()
  - numpy.savetxt()

Very basic

Can write headers/footers

- Numpy Arrays can be read from text files using
  - numpy.fromfile()
  - numpy.loadtxt()
  - numpy.genfromtxt()

Can skip headers, specify columns

Most flexible. Handles missing data. Skip footers and headers. Much more.

- These methods can also read/write binary files, but they won’t be machine portable.

Examples of Reading Data

- On the next slide we show four example programs for reading numerical data from a text file of the form shown below, with a header row and commas separating values.

```
Gender, Weight (lbs)
Male, 171.0
Male, 179.6
Male, 174.7
Male, 172.5
Female, 161.4
Male, 192.7
Male, 190.4
Male, 193.4
```

Examples of Reading Data

Iterating over file object

```
filein = 'weight-data.txt'
weight = [] # empty list to hold data

with open(filein, 'r') as f:
  for i, line in enumerate(f):
    if i == 0:
      pass # skips header
    else:
      data = line.split(',') # Splits line
      weight.append(float(data[-1]))

print(weight)
```

Using readlines()

```
filein = 'weight-data.txt'
weight = [] # empty list to hold data

with open(filein, 'r') as f:
  file_data = f.readlines()
  for line in file_data[1:]: # Skips head
    data = line.split(',')  # Splits line
    weight.append(float(data[-1])

print(weight)
```

Examples of Reading Data

Using CSV Reader

```
import csv

filein = 'weight-data.txt'
weight = [] # empty list

with open(filein, 'rb') as f:
  w = csv.reader(f, delimiter = ',')
  for i, line in enumerate(w):
    if i == 0:
      pass # Skip first row
    else:
      weight.append(float(line[-1]))

print(weight)
```

Using numpy.loadtxt()

```
import numpy as np


filein = 'weight-data.txt'


# Use loadtxt skipping 1 row and
#  only using second column
weight = np.loadtxt(filein,
               dtype = np.float_,
               delimiter = ',',
               skiprows = 1,
               usecols = (1,))


print(weight)
```

Note comma (needed if only one column is used)

.npy and .npz files

- NumPy provides its own functions to read and write arrays to binary files. This is accomplished with either:
  - np.save() function, which writes a single array to a NumPy .npy file.
  - np.savez() function, which archives several arrays into a NumPy .npz file.

Example

```
import numpy as np
a = np.arange(0, 100)*0.5
b = np.arange(-100, 0)*0.5
np.save('a-file', a)
np.save('b-file', b)
np.savez('ab-file', a=a, b=b)
```

- Creates three files:
  - ‘a-file.npy’ which contains the values for a
  - ‘b-file.npy’ which contains the values for b
  - ‘ab-file.npz’ which is an archive file containing both the a and b values

Loading .npy Files

- To retrieve the values from the .npy files we use the np.load() function

```
a = np.load('a-file.npy')
b = np.load('b-file.npy')
```

Loading .npz Files

- To retrieve the values from the .npz files we also use the np.load() function to load all the data into a dictionary that contains the archived arrays.

```
z = np.load('ab-file.npz')
a = z[‘a’]
b = z[‘b’]
```

Loading .npz Files

- To find the names of the arrays used in the dictionary, use the files attribute of the dictionary

```
>>> z.files
[‘a’, ‘b’]
```

Working with Pathnames

- The os.path module contains some nice functions for manipulating path and file names.

- I usually import os.path aliased to pth

```
import os.path as pth
```

Some `os.path` Functions

```
>>> import os.path as pth
>>> p = 'C:\\data\\temperature\\jun11.dat'
>>> pth.abspath(p)
'C:\\data\\temperature\\jun11.dat'
>>> pth.basename(p)
'jun11.dat'
>>> pth.dirname(p)
'C:\\data\\temperature’
```

Some `os.path` Functions

```
>>> pth.exists(p)
False
>>> pth.isfile(p)
False
>>> pth.isdir(p)
False
```

Some `os.path` Functions

```
>>> pth.split(p)
('C:\\data\\temperature', 'jun11.dat')
>>> pth.splitdrive(p)
('C:', '\\data\\temperature\\jun11.dat')
>>> pth.splitext(p)
('C:\\data\\temperature\\jun11', '.dat')
```