Capítulo 6

Lectura y Generación de Archivos

Hasta ahora, todos los ejemplos en el curso usan entradas del usuario con el teclado y como salida se imprime en pantalla el resultado. Esto es útil para trabajos pequeños, pero cuando se quiere analizar grandes cantidades de datos, es importante poder importarlos a Python. Estos datos pueden estar en tablas de Excel, de Matlab, o en archivos de texto (ascii) o pueden provenir de un equipo de campo o laboratorio. En este capítulo se busca entender como abrir, leer y escribir series de datos. Python tiene paquetes para leer datos de varias fuentes incluyendo Excel, matlab y otros, pero para el objetivo del curso, sólo nos concentraremos en leer archivos de texto plano (ascii). Note que tanto Matlab como excel pueden exportar los datos en dicho formato.

Abajo un ejemplo simple de cómo Python lee archivos de texto de manera rápida.

```
# read_test.py
# Simple read of a text file

# Reading line by line
print('Line by line')
fname = 'test_file.dat'
f = open(fname, 'r')
for line in f:
   print(line, end='')
f.close()

# reading file into list
print('')
print('File as list')
f = open(fname, 'r')
```

```
f_list = list(f)
print(f_list)
print(f_list[1])
f.close()

# Reading and printing all the file
print('Print entire file')
fname = 'test_file.dat'
f = open(fname, 'r')
text = f.read()
print(text)
```

donde el archivo `test_file.dat` tiene las siguientes líneas

```
This is line one
This is line 2
This is line three
This is the fourth line
```

El ejemplo anterior muestra varias de las funciones necesarias para leer archivos. Primero, se debe abrir `open` el archivo

```
f = open(fname, 'r')
```

al cual se le pone un nombre de variable `f`. `fname` proporciona el nombre del archivo que se quiere abrir, y `'r'` le dice a Python que este archivo es sólo de lectura (`read`). Se pueden usar modos de lectura como `'w'` para escribir un archivo (si el archivo existe, será sobre escrito), `'a'` adiciona al archivo al final, y `'r+'` abre el archivo para leer y escribir. Para archivos binarios se usa `'b'`. El modo de lectura es opcional, si no se utiliza, Python asume `'r'`.

El la primera forma de leer el archivo, se lee de manera secuencial (linea por linea) con

```
for line in f:
   print(line, end='')
```

y se imprime cada linea. Python lee el archivo linea por linea hasta llegar a la última linea y se detiene el `for` loop/ En el segundo tipo de lectura del archivo, se lee el archivo completo y cada linea se pone en un elemento de una lista.

```
f_list = list(f)
```

Finalmente está la forma de leer el archivo completo e imprimirlo con el comando

```
text = f.read()
print(text)
```

6.1. I/O de datos en Python

Esto es la sección mas relevante, poder leer (y guardar) archivos en formato plano (*flat file*). En general, esto implica una tabla con datos en filas y columnas, con valores numéricos. En algunos casos, el archivo tiene un encabezado *header* que no tiene valores numéricos, sino texto explicando que significa cada columna.

6.1.1. Lectura de archivos

Un primer ejemplo para leer un archivo de datos

```
# read_data.py
# Read a simple data file with numerical values

import numpy as np
import matplotlib.pyplot as plt

# Assign filename: file
file = 'some_data.dat'

# Import file: data
data = np.loadtxt(file)

# Plot results
plt.scatter(data[:, 0], data[:, 1])
plt.scatter(data[:, 0], data[:, 2])
plt.show()
```

Acá se incluye una figura con los datos leídos. El archivo tiene 3 columnas, y se muestra en la figura la primera contra la segunda y tercera. La Figura 6.1 muestra el resultado. En el siguiente capítulo veremos en detalle el uso de gráficas en Python, por ahora este ejemplo basta.

El comando `np.loadtxt` tiene una gran variedad de opciones para leer los archivos (digite `help(np.loadtxt)` en Python para ver la documentación). El formato general de la función tiene muchas opciones, acá sólo muestro algunas de ellas

```
loadtxt(fname, dtype='float', comments='#',
        delimiter=',', skiprows=1, usecols=[0,2])
```

donde se le ordena a `loadtxt` leer los datos como `floats`, lineas con comentarios se marcan como `#` y no son leídas, las columnas están separadas por comas (típico de archivos `.csv`), se salta una línea que puede ser el encabezado y sólo se leen la primera y tercera columna. Note que todos los comandos (a excepción del nombre del archivo son opcionales) y Python asume algunos valores. Por ejemplo, las columnas por defecto están separadas por espacios, y los valores se asumen como `float`.

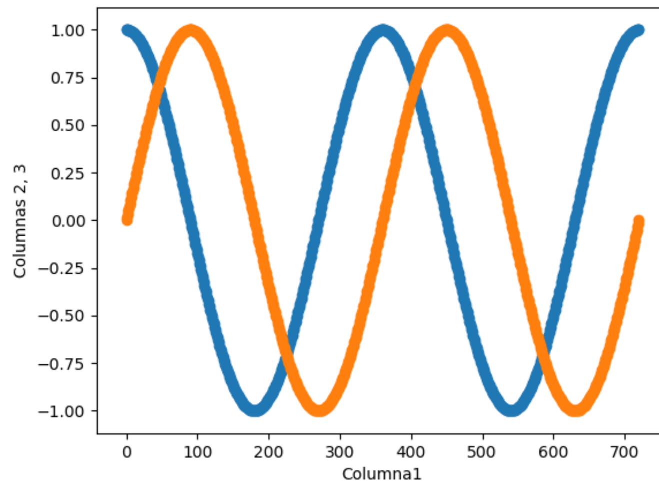

Figura 6.1: Resultado de lectura de archivo `some_data.dat`

Otro ejemplo, para el mismo archivo pero con un encabezado,

```
theta            cos              sin
0.0000000e+00    1.0000000e+00    0.0000000e+00
1.0000000e+00    9.9984770e-01    1.7452406e-02
2.0000000e+00    9.9939083e-01    3.4899497e-02
```

se puede realizar *saltando* el encabezado como se muestra

```
# read_data2.py
# Read a data file with numerical values and a header

import numpy as np
import matplotlib.pyplot as plt

# Assign filename: file
file = 'some_data_header.dat'

# Import file: data
data = np.loadtxt(file, skiprows=1)

# Plot results
plt.scatter(data[:, 0], data[:, 2]/data[:,1])
plt.show()
```

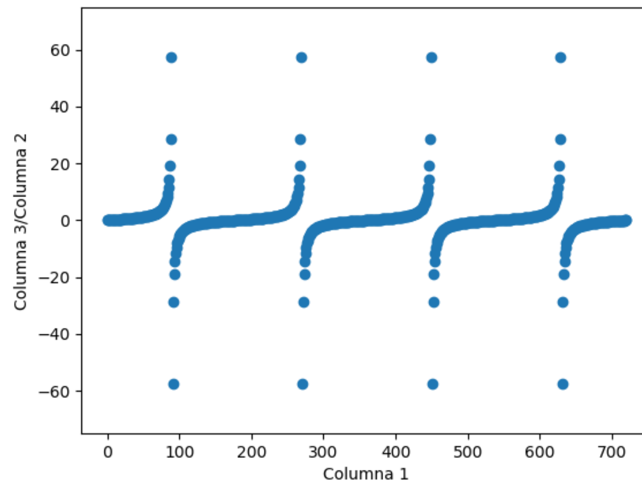

Figura 6.2: Resultado de lectura de archivo `some_data_header.dat`, dividiendo la tercera columna por la primera (función tangente)

Existen archivos más complejos que requieren de mayor trabajo para leerlos. Un ejemplo es un archivo que contiene los nombres de estaciones sismológicas de la red nacional de Colombia y su latitud, longitud y elevación. El archivo tiene el siguiente formato

```
APAC          7.9     -76.58       195
ARGC          9.585   -74.246      117.9
BBAC          2.022   -77.247      1716
...
```

donde las diferentes columnas están separadas por espacios (el número de espacios puede variar). `Numpy` tiene una función que en principio puede leer este tipo de archivos `np.genfromtxt()`, pero a mi personalmente no me gustó como se comportaba con diferentes archivos.

Python tiene un paquete (que debe ser instalado en su equipo) conocido como |tt Pandas. Este paquete tiene la capacidad de leer archivos planos, mixtos (caracteres y números) e incluso archivos de Excel y Matlab. Abajo, un ejemplo de cómo leer el archivo anterior:

```
# read_stations.py
# Read seismic station info from the RSNC
# File is whitespace delimited
```

```
import numpy as np
import pandas as pd

fname = 'rsnc3.dat'
data = pd.read_csv(fname, delim_whitespace=True, header=None)

sta = np.array(data[0])
lat = np.array(data[1])
lon = np.array(data[2])
ele = np.array(data[3])
print(sta)
print(lat)

# Plot results
...
```

donde se imprime los nombres de las estaciones (en un arreglo) y las latitudes (en otro arreglo). Aunque no se muestra el código, la figura 6.3 genera un mapa con la ubicación de las estaciones en Colombia.

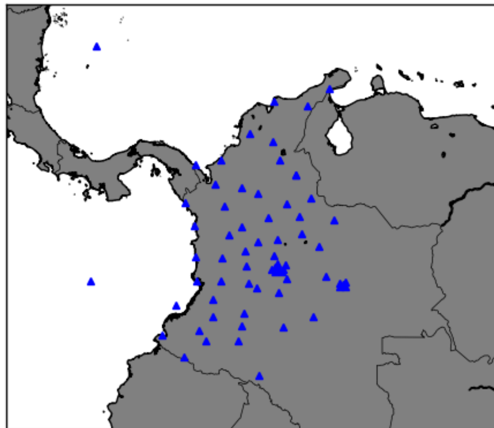

Figura 6.3: Resultado de lectura de archivo `some_data_header.dat`, dividiendo la tercera columna por la primera (función tangente)

Note que el código anterior carga el paquete

```
import pandas as pd
```

y carga los datos con el comando

```
data = pd.read_csv(fname, delim_whitespace=True, header=None)
```

donde se le dice al código que no hay encabezado y que los separadores entre columnas están marcados por espacios libres (pueden ser tabs, comas, etc). Lo curioso de `Pandas` es que tiene su propio tipo de estructura (llamado `DataFrame`) que son una especie de arreglos de matrices, pero que pueden tener diferentes tipos de elementos (str, int, float, etc). Para convertir alguna columna de `data` a un arreglo, simplemente se utiliza la columna correspondiente.

```
lat = np.array(data[1])
```

y esto es lo que se observa en la Figura 6.3

6.1.2. Guardar archivos

Abajo un ejemplo de un programa que lee un archivo (`another_data.dat`) con pares de números, calcula el producto y guarda un archivo con los números originales y su producto and un archivo de salida (que se puede llamar por ejemplo `test.dat`). Tenga cuidado que esto puede significar borrar un archivo existente.

```
# fileinout.py
# Read a simple text file with a pair of numbers
# and output their product in a new file

import numpy as np

fin = input("Enter input file name ")

# Load data and put in individual arrays
data = np.loadtxt(fin)
x = data[:,0]
y = data[:,1]

fout = input("Enter output file name ")

# Product of two arrays
z = x*y

# Put all arrays together, and vertical
data_out = np.vstack((x,y,z)).T

# Save data with given format
np.savetxt(fout, data_out,fmt='%5.2f %5.2f %5.2f')
```

El archivo de entrada se abre con

```
data = np.loadtxt(fin)
```

donde `fin` es el nombre del archivo que se quiere abrir. Si el archivo no existe, Python generará un error. Los valores dentro del archivo son leídos con el comando

```
data = np.loadtxt(fin)
x = data[:,0]
y = data[:,1]
```

y las dos columnas son separadas en las variables `x` y `y`. Creo que esto se puede hacer de manera directa con `np.loadtxt` pero no he podido encontrar la forma de hacerlo. El lector de Python, como se vio en la secci?øn anterior, sabe que tan largo es el archivo y para de leer cuando llega al final.

Posteriormente se hace la multiplicación de los dos arreglos en un nuevo arreglo `z=x*y`. Para facilitar guardar el nuevo archivo, se genera un arreglo 2D

```
data_out = np.vstack((x,y,z)).T
```

donde los tres arreglos 1D se juntan, en forma vertical. Finalmente, esa nueva matriz `data_out` se guarda en el archivo

```
np.savetxt(fout, data_out,fmt='%5.2f %5.2f %5.2f')
```

con un formato específico. Note que el comando de formato `fmt` es opcional, y si no se usa, Python guarda los datos de la matriz en un formato predefinido.

El encabezado del archivo de entrada y salida se muestra a continuación

```
Entrada
-1.0000   4.0000
-0.9000   3.1500
-0.8000   2.4000
...
Salida
-1.00  4.00 -4.00
-0.90  3.15 -2.83
-0.80  2.40 -1.92
...
```

6.2. I/O veloz en Python

En computación, los cálculos y operaciones matemáticas están optimizadas dentro de los programas, por lo que en muchos casos el tiempo computacional se gasta en procesos I/O, leer y guardar archivos. Para realizar los procesos I/O con mayor rapidez, en muchos casos es mejor guardar los archivos en formato binario, envez de guardarlos como archivos de texto plano.

Suponga que Ud. tiene una matriz grande que quiere guardar. El siguiente programa muestra este ejemplo y el tiempo que tarda dicha operación y cuanto tarde en leerlo nuevamente.

```
# testio.py
# Save (and load) a large matrix in
# Binary or ascii format

import numpy as np
import time

m= 100000
n= 10
a = np.ones((m,n))*1.1

# save file in binary format
start = time.time()
a.tofile("fout_testio.bin")
print ('Binary save time ', time.time()-start, 'seconds.')

# Save in Native Numpy format
start = time.time()
np.save("fout_testio.npy", a)
print ('Numpy  save time ', time.time()-start, 'seconds.')

# save file in plain text
start = time.time()
fout = "fout_testio.dat"
np.savetxt(fout, a)
print ('text   save time ', time.time()-start, 'seconds.')

# Loading those files again

# save file in binary format
start = time.time()
b1 = np.fromfile("fout_testio.bin",dtype='float')
b1 = b1.reshape(m,n)
print ('Binary load time ', time.time()-start, 'seconds.')

# Save in Native Numpy format
start = time.time()
b2 = np.load("fout_testio.npy")
print ('Numpy  load time ', time.time()-start, 'seconds.')

# save file in plain text
start = time.time()
b3 = np.loadtxt("fout_testio.dat")
print ('text   load time ', time.time()-start, 'seconds.')
```

En mi computador (Mac BookPro, Procesador de 2GHz) el proceso tarda para una matriz de $(100000, 10)$

```
Binary save time  0.016 seconds.
Numpy  save time  0.022 seconds.
text   save time  1.213 seconds.
```

guardando

```
Binary load time  0.002 seconds.
Numpy  load time  0.008 seconds.
text   load time  2.639 seconds.
```

volviendo a abrir el archivo. Note como usando archivos de texto, la lectura puede ser muy demorada. El factor de velocidad es de 55 veces comparado binario a ascii. Los archivos además ocupan espacios de memoria muy distinta

```
7.6M  fout_testio.bin
7.6M  fout_testio.npy
24M   fout_testio.dat
```

Note que el archivo en el formato nativo de Python `.npy` ocupa el mismo espacio que un binario normal, pero el archivo binario no *sabe* que el archivo es una matriz, sino que guarda la matriz plana, sin tamaños predefinidos, y toca reorganizarla.

En algunos casos, el usuario puede querer guardar varios arreglos en un sólo archivo, manteniendo la información de las dimensiones de los arreglos. Esto se puede hacer con el comando

```
np.savez(outfile, x, y)
```

donde se guardan los arreglos `x` y `y`, los cuales después se pueden cargar con

```
npzfile = np.load(outfile)
```

El programa muestra como guardar un archivo binario en formato `.npz`, y como volver a cargarlo.

```
# testio2.py
# Save (and load) a number of arrays and variables large matrix in
# Binary NPZ format

import numpy as np

m= 100000
n= 10
a = np.ones((m,n))*1.1
y = np.random.rand(n)
x = np.random.rand(m)
```

```
# Save in Native Numpy format
np.savez("fout_testio2b.npz", a,x,y,m,n)
np.savez("fout_testio2.npz", a=a,x=x,y=y,m=m,n=n)

# Load the file and look at arrays
npzfile = np.load("fout_testio2.npz")
n  = npzfile.f.n
m  = npzfile.f.m
x0 = npzfile.f.x
y0 = npzfile.f.y
a0 = npzfile.f.a

npzfile = np.load("fout_testio2b.npz")
print(npzfile.files)
```

Es importante notar que se pueden guardar un número indefinido de arreglos o variables. Sin embargo, cuando carga el archivo, a cada variable o vector le pone un nombre

```
['arr_1', 'arr_0', 'arr_3', 'arr_2', 'arr_4']
```

y es una buena idea guardar las variables con un nombre que pueda ser recordado al cargar el archivo nuevamente, asi:

```
np.savez("fout_testio2.npz", a=a,x=x,y=y,m=m,n=n)
```

y cada arreglo se puede cargar

```
m  = npzfile.f.m
x  = npzfile.f.x
```

6.3. Archivos Ascii vs. Binarios

Como se ve en este capítulo, hay un *tradeoff* (compensaci?øn o sacrificio) entre trabajar con archivos ascii o binarios. Los archivos `ascii` son más fáciles de trabajar, se pueden cargar en Excel o matlab con facilidad e incluso se puede abrir el archivo y mirar los datos. Es el método sugerido cuando el tamaño del archivo o velocidad de los cálculos no son un problema.

Sin embargo, para bases de datos muy grandes y procesamiento de datos pesados, es mejor usar formatos binarios por que permite transferencia I/O rápido y ocupando menos memoria en el computador (un problema cada vez menos importante supongo). De hecho Excel muchas veces se vara cuando tiene que procesar bases de datos *no tan* grandes. Adicionalmente, en el formato binario, no se tiene que decidir de antemano el formato (y la precisión que se quiere guardar los números; cuantos decimales quiero guardar?). Los archivos

binarios son menos portátiles (pasarlos de un mac a un windows puede causar problemas) y se requiere conocer en muchos casos el formato del archivo para poder leerlo.

Problemas

6.1. Escriba un programa que lea un archivo de texto plano que contenga 5 números por línea. Calcule el promedio de los 5 números y luego substraiga el promedio a cada una de los 5 números originales. Guarde un nuevo archivo con los valores con el promedio removido (*demeaned*). Permite que el usuario pueda especificar el nombre del archivo de entrada y salida.

Por ejemplo, si el archivo de entrada es

```
10 20 30 40 50
1 2 3 4 5
2 4 6 8 10
5 5 5 5 5
```

el programa deberá escribir un nuevo archivo así.

```
-20.00 -10.00  0.00 10.00 20.00
-2.00 -1.00  0.00  1.00  2.00
-4.00 -2.00  0.00  2.00  4.00
 0.00  0.00  0.00  0.00  0.00
```

6.2. En Geociencias hay múltiples fuentes para obtener información de todo tipo (geofísica, geología, fallas, geoquímica, etc). Una de ellas se llama GEOROC (`http://georoc.mpch-mainz.gwdg.de/georoc/`) que contiene información sobre geoquímica de rocas en el mundo. De ahí se extrajo un archivo llamado `ANDEAN_ARC_1.csv`, que contiene una gran cantidad de información, incluyendo latitud y longitud de muestreo y contenido en peso de SiO_2 (entre otras muchas variables).

Utilizando su método preferido (por ejemplo con `pandas`), lea el archivo e imprima los primeros 20 líneas con la Latitud, longitud y porcentaje de SiO_2 en peso, que los encuentra en las columnas (4, 6, y 27 respectivamente).

Ayuda: Si usa `pandas`, el `DataFrame` se puede convertir a un arreglo de `numpy` con el comando

```
data = data.as_matrix()
```

Tenga también en cuenta que algunas variables tienen valores inexistentes, así que dependiendo del método, debe ajustar dichos casos.