Chap06

August 4, 2020

1 Capítulo 6

1.1 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.

1.1.1 Hazlo tu mismo

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

[97]:
```
# read_test.py
# Lectura simple de un archivo de texto
#

# linea por linea
print('Lectura de archivo: linea por linea')
fname = 'data/test_file.dat'
f = open(fname, 'r')
for line in f:
   print(line, end='')
f.close()

# poner en una lista
print('')
print('Lectura de archivo: en una lista')
f = open(fname, 'r')
f_list = list(f)
print(f_list)
print(f_list[1])
f.close()

# lectura completa
```

```
print('Lectura de archivo: en una variable')
f = open(fname, 'r')
text = f.read()
print(text)
```

```
Lectura de archivo: linea por linea
La primera linea
La 2da linea
Es la tercera
Cuarta linea

Lectura de archivo: en una lista
['La primera linea\n', 'La 2da linea\n', 'Es la tercera\n', 'Cuarta linea\n']
La 2da linea

Lectura de archivo: en una variable
La primera linea
La 2da linea
Es la tercera
Cuarta linea
```

El archivo 'test_file.dat' tiene las siguientes lineas

```
La primera linea
La 2da linea
Es la tercera
Cuarta linea
```

1.1.2 Explicación

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' ".

La primera forma de leer el archivo, se hace 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)
```

1.2 I/O de datos en Python

Para el estudio de datos en geociencias, es necesario 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 o *header* que no tiene valores numéricos, sino texto explicando que significa cada columna.

Para archivos en otros formatos, por ejemplo datos binarios como `segy`, `mseed` o cualquier otro formato propio de cada subcampo de las geociencias, es necesario utilizar herramientas adicionales o saber exactamente el formato para poder leerlo. **Esto no será estudiado en este curso**.

1.2.1 Hazlo tu mismo

El archivo `some_data.dat` tiene una serie de datos organizados así

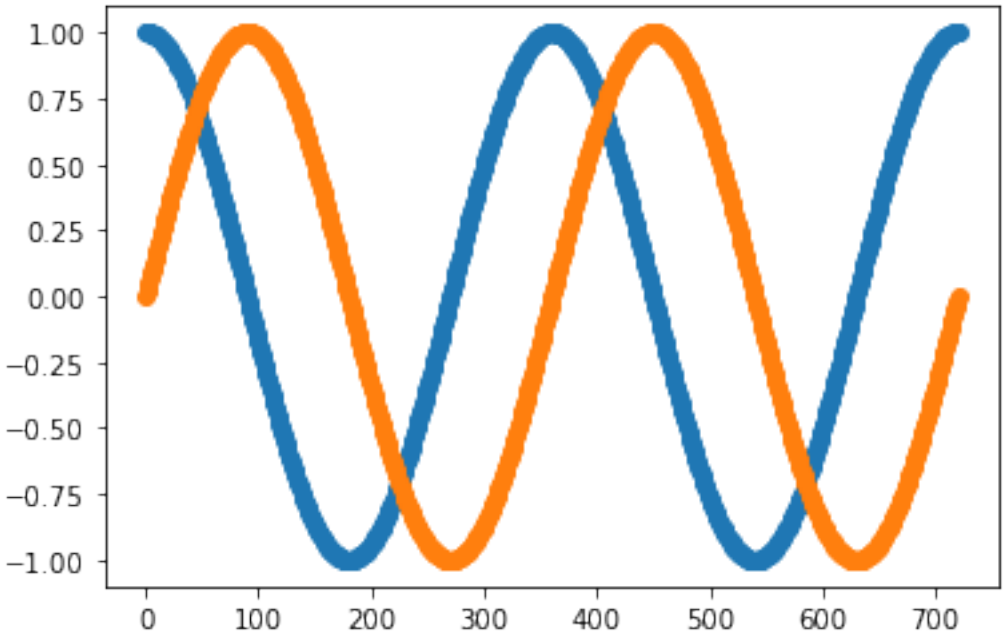
```
   0.0000000e+00   1.0000000e+00   0.0000000e+00
   1.0000000e+00   9.9984770e-01   1.7452406e-02
   2.0000000e+00   9.9939083e-01   3.4899497e-02
   3.0000000e+00   9.9862953e-01   5.2335956e-02
   4.0000000e+00   9.9756405e-01   6.9756474e-02
...
   7.1600000e+02   9.9756405e-01  -6.9756474e-02
   7.1700000e+02   9.9862953e-01  -5.2335956e-02
   7.1800000e+02   9.9939083e-01  -3.4899497e-02
   7.1900000e+02   9.9984770e-01  -1.7452406e-02
   7.2000000e+02   1.0000000e+00  -4.8985872e-16
```

Usando el siguiente modelo, lea el archivo y grafique los resultados.

Nota: Aún no sabemos graficar en Python, el próximo capítulo la haremos. Por ahora, simplemente siga el programa. En muchos casos, para *ver* los datos, las gráficas son la mejor opción.

[98]:
```
# read_data.py
# Leer un archivo plano con datos numéricos en columnas.
#

import numpy as np
import matplotlib.pyplot as plt

# nombre del archivo
fname = 'data/some_data.dat'

# Cargue el archivo con Numpy
data = np.loadtxt(fname)
```

```
# Figura de columnas 1 vs 2, 1, vs 3
plt.scatter(data[:, 0], data[:, 1])
plt.scatter(data[:, 0], data[:, 2])
plt.savefig('../figs/chap06_fig1.pdf')
plt.show()
```



1.2.2 Explicación

Por ahora, olvide la parte relacionada con la generación de la figura.

Para leer el archivo, simplemente se tiene que solicitar a `numpy` que lo lea

`data = np.loadtxt(fname)`

Note que a diferencia del caso anterior, no abrimos el archivo, `loadtxt` lo hace internamente y no tenemos que preocuparnos por eso. Además lo cierra también.

Una vez leído, la variable `data` es un arreglo de numpy con 721 filas y 3 columnas. Numpy automáticamente asigna el tamaño de `data` sin necesidad de que Ud sepa cuantos datos hay.

Precaución. Numpy asume que el archivo está organizado en columnas, separadas por espacios y que todas las columnas estan ocupadas con valores numéricos, tienen la misma longitud y no tienen valores vacíos.

[99]:
```
print(type(data))
print(np.shape(data))
```

```
<class 'numpy.ndarray'>
(721, 3)
```

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 `float`, 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`.

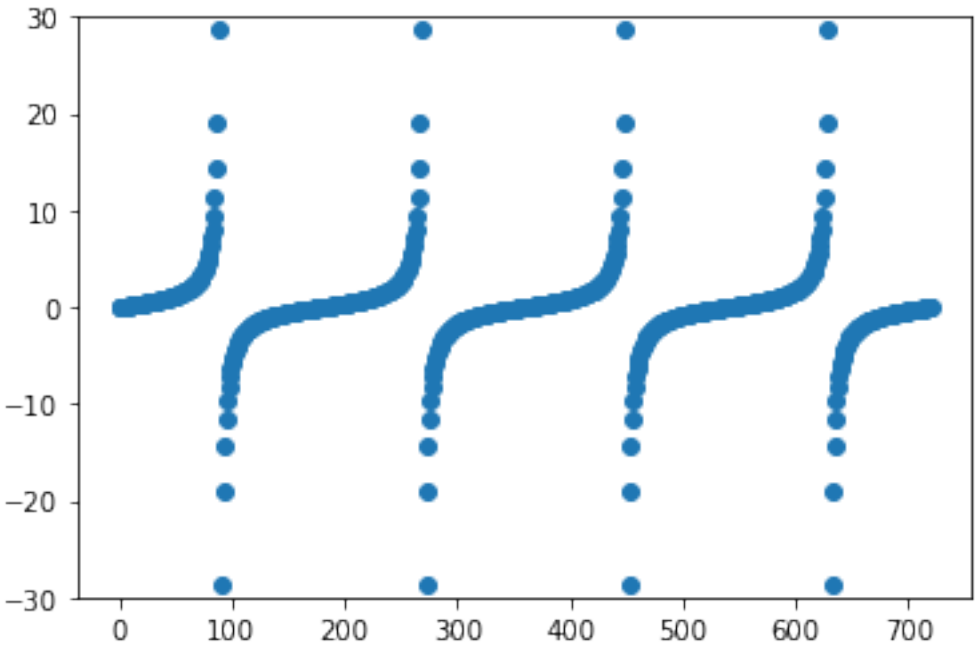
1.2.3 Hazlo tu mismo

El archivo `some_data_header.dat` tiene una serie de datos organizados así

```
   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
   3.0000000e+00   9.9862953e-01   5.2335956e-02
...

   7.2000000e+02   1.0000000e+00  -4.8985872e-16
```

Lea el archivo y grafique los resultados, pero esta vez grafique la tercera columna dividida por la primera.

[100]:
```
# read_data2.py
# Leer un archivo plano con datos numéricos en columnas.
#
import numpy as np
import matplotlib.pyplot as plt

# nombre del archivo
fname = 'data/some_data_header.dat'

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

# Figura
plt.scatter(data[:, 0], data[:, 2]/data[:,1])
plt.ylim([-30, 30])
plt.savefig('../figs/chap06_fig2.pdf')
plt.show()
```



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). En general, simplemente leerlo con `npo.loadtxt` genera un error, porque la primera columan no es numérica, que es lo que espera la función. `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 conocido como `Pandas`. Este paquete tiene la capacidad de leer archivos planos, mixtos (caracteres y números) e incluso archivos de Excel y Matlab.

Un ejemplo de cómo leer el archivo.

[101]:
```
# read_stations.py
# Read seismic station info from the RSNC
# File is Tab delimited

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

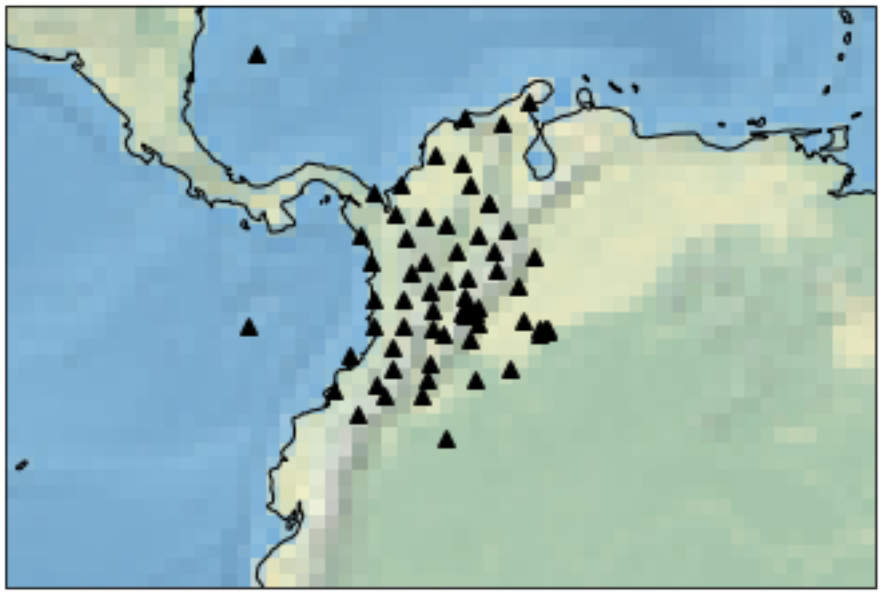
```
fname = 'data/rsnc.dat'
data = pd.read_csv(fname, delim_whitespace=True, header=None)
print('Estructura de Pandas')
print(data)

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

```
Estructura de Pandas
       0       1       2       3
0   APAC   7.900 -76.580   195.0
1   ARGC   9.585 -74.246   117.9
2   BBAC   2.022 -77.247  1716.0
3   BAR2   6.592 -73.184  1783.0
4    BRR   7.107 -73.712   104.0
..   …     …      …      …
63   URI  11.702 -71.993    97.0
64   VIL   4.111 -73.693  1110.0
65  YPLC   5.397 -72.380   696.0
66   YOT   3.983 -76.345  1059.0
67   ZAR   7.492 -74.858   200.0

[68 rows x 4 columns]

['APAC' 'ARGC' 'BBAC' 'BAR2' 'BRR' 'BET' 'CAP2' 'CAQC' 'CRJC' 'TABC' 'CHI'
 'CBOC' 'CRU' 'CVER' 'CUM' 'DBB' 'FLO2' 'FOM' 'GARC' 'GUA' 'GUY2C' 'GR1C'
 'MAP' 'JAMC' 'MACC' 'RGSC' 'CLEJA' 'LCBC' 'MAL' 'NOR' 'OCA' 'OCNC' 'ORTC'
 'PAM' 'PIZC' 'POP2' 'PRA' 'PRV' 'PGA1' 'PGA3' 'PGA4' 'PGA5' 'PGA6' 'PTB'
 'PTGC' 'PTLC' 'PTA' 'QUET' 'RUS' 'SJC' 'URE' 'PAL' 'SML1C' 'SPBC' 'HEL'
 'SMAR' 'SBTC' 'SMORC' 'SOL' 'TAM' 'TOL' 'TUM' 'TVCAC' 'URI' 'VIL' 'YPLC'
 'YOT' 'ZAR']
[ 7.9    9.585  2.022  6.592  7.107  2.723  8.646  4.402 11.02   5.011
  4.63   5.865  1.568  4.521  0.941  7.017  1.583  4.475  2.187  2.542
  5.226  3.003  4.004  3.279  2.145  4.368  3.536  8.857  4.01   5.564
  8.239  8.24   3.909  7.34   4.965  2.54   3.714 13.376  3.746  3.898
  3.868  3.946  3.775  6.54   4.199  0.171  7.142  4.381  5.893  9.897
  7.752  4.906  8.863  5.652  6.191 11.164  4.477  4.575  6.224  6.435
  4.585  1.824  4.718 11.702  4.111  5.397  3.983  7.492]
[-76.58  -74.246 -77.247 -73.184 -73.712 -75.418 -77.359 -73.986 -72.882
 -74.204 -73.732 -76.012 -76.952 -74.074 -77.825 -76.21  -75.653 -73.859
 -75.494 -72.624 -75.383 -78.167 -81.607 -76.674 -73.848 -74.186 -74.041
```

```
 -76.368 -77.332 -74.869 -73.32  -73.319 -75.246 -72.7   -77.36  -76.676
 -74.886 -81.363 -71.572 -71.566 -71.418 -71.315 -71.332 -74.456 -72.134
 -74.797 -77.809 -73.883 -73.083 -75.18  -75.533 -76.283 -73.992 -74.072
 -75.529 -74.225 -74.287 -74.17  -77.405 -71.791 -75.32  -78.727 -74.085
 -71.993 -73.693 -72.38  -76.345 -74.858]
```

donde se imprime la estructura de {Pandas}, y los arreglos de Numpy extraídos.

[102]:
```
# Mapa de los resultados
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

fig = plt.figure()
ax  = fig.add_subplot(1,1,1,projection=ccrs.PlateCarree())
ax.set_extent([-90, -60, -5, 15])
ax.stock_img()
ax.coastlines()
ax.plot(lon,lat,'k^')
plt.savefig('../figs/chap06_fig3.pdf')
```



1.2.4 Explicación

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). Note que el dataframe organiza los datos en columnas y filas numeradas.

Para convertir alguna columna del dataframe data a un arreglo, simplemente se utiliza la columna correspondiente.

```
sta = np.array(data[0])
lat = np.array(data[1])
```

En este curso, trabajaremos con arreglos de Numpy, por lo que después de leer el archivo, convertimos las variables a arreglos. El mapa de las estaciones se muestra en la figura anterior (el código para hacer este tipo de mapas lo miraremos en el siguiente capítulo).

1.3 Guardar archivos

En muchos casos, es importante poder leer archivos, pero también es importante poder guardar los resultados de un análisis, de procesamiento o resultados de simulaciones y cálculos, sin la necesidad de repetir el procesamiento cada vez que se quiera revisar los resultados.

Aunque las gráficas son una forma de presentar resultados, no permiten mirar los datos brutos que siempre es bueno tenerlos para poder replicar trabajos de investigación.

1.3.1 Hazlo tu mismo

Escriba un programa que lea el archivo another_data.dat, que tiene dos columnas (con pares de números) y - Calcula el producto de los dos números - Guarda un nuevo archivo con los números originales, y su producto como 3ra columna. - El archivo de salida se puede llamar test.dat

Tenga cuidado que esto puede borrar un archivo existente.

Archivo de entrada

```
-1.0000   4.0000
-0.9000   3.1500
...
1.0000    6.0000
```

[103]:
```
# fileinout.py
# Lea un archivo con pares de números
# y guarde un archivo con el producto de los números
#
import numpy as np

fin  = 'data/another_data.dat'
```

```
fout = 'dout/test.dat'

# Cargar los datos, dos arreglos x y y
data = np.loadtxt(fin)
x = data[:,0][:,None]
y = data[:,1][:,None]

# producto de los dos arreglos
z = x*y

# pegue arreglos en una sola matriz
data_out = np.hstack((x,y,z))

# Guarde un nuevo archivo, con formato
np.savetxt(fout, data_out,fmt='%5.2f %5.2f %5.2f')
```

[104]:
```
print('Tamaño arreglos de entrada')
print(np.shape(x), np.shape(y))
print('Tamaño arreglo producto X*Y')
print(np.shape(z))
print('Tamaño arreglo de salida')
print(np.shape(data_out))
```

```
Tamaño arreglos de entrada
(21, 1) (21, 1)
Tamaño arreglo producto X*Y
(21, 1)
Tamaño arreglo de salida
(21, 3)
```

```
Archivo de salida
```

```
-1.00  4.00 -4.00
-0.90  3.15 -2.83
...
 1.00  6.00  6.00
```

1.3.2 Explicación

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][:,None]
y = data[:,1][:,None]
```

y las dos columnas son separadas en las variables x y y. En este ejemplo introduzco una notación nueva, donde x[:,None] lo que busca es generar un arreglo vertical (un vector) con tamaño (21, 1). Si esto no se hace, el arreglo tendría la forma (21,) y no permitiría hacer operaciones matriciales. Es probable que esto se pueda lograr directamente con np.loadtxt, pero no he podido encontrar la forma de hacerlo. La función loadtxt, como se vio en la sección anterior, sabe que tan largo es el archivo.

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.hstack((x,y,z))

donde los tres arreglos 1D se juntan, pegándolos horizontalmente, es decir cada arreglo representa una columna. El resultado data_out tiene un tamaño (21,3).

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.

1.4 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 consume 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, en vez de guardarlos como archivos de texto plano.

1.4.1 Hazlo tu mismo

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.

[105]:
```
# testio.py
# Guardar y cargar una matrix grande
# formatos binarios y ascii
#
import numpy as np
import time

# tamaño de matriz
m= 100000
n= 10
a = np.ones((m,n))*1.1

# Guardar datos
f1 = 'dout/fout_testio.bin'
```

```
f2 = "dout/fout_testio.npy"
f3 = "dout/fout_testio.dat"
print('Tiempo requerido para guardar')

# formato binario
start = time.time()
a.tofile(f1)
print ('Binary: %8.5f segundos.' %(time.time()-start))

# formato Numpy nativo (binario)
start = time.time()
np.save(f2, a)
print ('Numpy:  %8.5f segundos.' %(time.time()-start))

# formato plano de texto
start = time.time()

np.savetxt(f3, a)
print ('text:   %8.5f segundos.' %(time.time()-start))

# Cargar datos
print('')
print('Tiempo requerido para cargar')

# formato binario
start = time.time()
b1 = np.fromfile(f1,dtype='float')
b1 = b1.reshape(m,n)
print ('Binary: %8.5f segundos.' %(time.time()-start))

# formato Numpy nativo (binario)
start = time.time()
b2 = np.load(f2)
print ('Numpy:  %8.5f segundos.' %(time.time()-start))

# formato plano de texto
start = time.time()
b3 = np.loadtxt(f3)
print ('text:   %8.5f segundos.' %(time.time()-start))
```

```
Tiempo requerido para guardar
Binary:  0.01576 segundos.
Numpy:   0.01829 segundos.
text:    0.93194 segundos.

Tiempo requerido para cargar
Binary:  0.00231 segundos.
```

```
Numpy:   0.00370 segundos.
text:    1.73240 segundos.
```

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)
```

1.4.2 Hazlo tu mismo

Genere un programa donde se usen los comando de numpy para guardar un **único** archivo binario en formato {.npz}, y volver a cargarlo.

Genere los siguientes arreglos

```
m= 100000
n= 10
a = np.ones((m,n))*1.1
y = np.random.rand(n)
x = np.random.rand(m)
z = 1.5
```

[106]:
```
# testio2.py
# Guarde y cargue varios arreglos en un único archivo
# en formato NPZ de Numpy
#
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)
z = 1.5
```

```
print('Variables guardadas')
print('Shape de a, x, y, size(z)')
print(np.shape(a),np.shape(x),np.shape(y),np.size(z))
print('')

# Guardar en formato npz numpy
f1 = "dout/fout_testio2b.npz"
f2 = "dout/fout_testio2.npz"

np.savez(f1, a,x,y,m,n,z)
np.savez(f2, a=a,x=x,y=y,m=m,n=n,z=z)

# Cargar archivos, confirmar tamaños
npfile = np.load(f2)
n  = npfile.f.n
m  = npfile.f.m
x0 = npfile.f.x
y0 = npfile.f.y
a0 = npfile.f.a
z0 = npfile.f.z
print('Variables cargadas')
print('Shape de a, x, y, size(z)')
print(np.shape(a0),np.shape(x0),np.shape(y0),np.size(z0))
print('')

npfile2 = np.load(f1)
n1  = npfile2.f.arr_3
m1  = npfile2.f.arr_4
x1  = npfile2.f.arr_1
y1  = npfile2.f.arr_2
a1  = npfile2.f.arr_0
z1  = npfile2.f.arr_5
print('Variables cargadas 2')
print(npfile2.files)
print('Shape de a, x, y, size(z)')
print(np.shape(a1),np.shape(x1),np.shape(y1),np.size(z1))
```

```
Variables guardadas
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1

Variables cargadas
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1

Variables cargadas 2
```

```
['arr_0', 'arr_1', 'arr_2', 'arr_3', 'arr_4', 'arr_5']
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1
```

Es importante notar que se pueden guardar un número indefinido de arreglos o variables, en un formato eficiente (rápido) y con bajo consumo de memoria. Ambos archivos ocupan apenas 8.4Mb. La gran ventaja es que el programa puede guardar varios arreglos y al cargarlos, mantiene la información de cada arreglo (su tamaño y shape).

Sin embargo, cuando el archivo es guardado con el comando

```
np.savez("fout_testio2b.npz", a,x,y,m,n,z)
```

al cargarlo cada variable o vector recibe un nombre

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

de acuerdo al orden en que fueron guardados (a, x, y, m, n, z). En este caso, la dificultad es que uno debe saber el orden en el que fue guardada cada variable.

Una buena idea es 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
```

1.5 Archivos ASCII vs Binarios

Como se ve en este capítulo, hay un *tradeoff* (compensación 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.