ANyP, Práctica 1e.

Elementos de programación Python: Formatos y E/S por archivos.

Esta práctica tiene como objetivo ampliar nuestras habilidades para presentar información en pantalla o guardarla en un archivo, asegurando un formato adecuado en la salida. Se recomienda utilizar preferentemente el formato `f-string`, tal como se explica en el apunte teórico de la cátedra. `f-string` fue introducido en Python 3.6 para permitir insertar expresiones dentro de cadenas de caracteres literales. El formato provee una manera legible y sencilla de construir cadenas en forma dinámica.

Ejercicio 1. En su forma más sencilla, la impresión usando `f-string` requiere colocar las variables entre llaves (`{}`) y Python interpretará que se desea insertar el valor que contienen las mismas dentro de la cadena de caracteres. Para verificarlo, cree un programa que asigne algún valor a dos variables y luego imprima en pantalla un cartel que incluya dichos valores.

Ejercicio 2. La salida usando `f-string` permite también darle un formato a los datos que se vuelcan en pantalla o que se guardan en un archivo. Para constatar su implementación, calcule el área de una elipse de semiejes mayor y menor $a = 4$ y $b = 3$, respectivamente. Imprima el resultado en pantalla, con dos decimales.

Ejercicio 3. Defina una variable con el nombre de una roca y muestre su valor en pantalla usando `f-string` y un alineamiento a derecha e izquierda, en un campo de 20 caracteres. Alinear en un campo mayor permite realizar salidas tabuladas que facilitan la lectura de datos y resultados.

Ejercicio 4. El formato `f-string` también permite realizar cálculos en su interior, colocando expresiones en lugar de variables. Las densidades aproximadas del granito y del basalto son de 2.7 y 3.2 g/cm^3, respectivamente. Calcule, dentro de un `f-string`, la densidad media de ambos minerales y muestre el resultado en pantalla.

Entrada/salida por archivos.

Hasta ahora, los datos que necesita un programa han sido ingresados por teclado durante la ejecución del mismo y los resultados se han mostrado por pantalla. Es claro que esta forma de proceder es adecuada sólo si la cantidad de datos de entrada/salida es relativamente pequeña. Para problemas que involucren grandes cantidades de datos resulta más conveniente que los mismos sean guardados en *archivos*. En lo que sigue veremos las instrucciones que proporciona Python para trabajar con archivos.
Un archivo es un conjunto de datos almacenado en un dispositivo (tal como un disco rígido) al que se le ha dado un nombre. Para la mayoría de las aplicaciones los únicos tipos de archivos que nos interesa considerar son los *archivos de texto*. Un archivo de texto consta de una serie de líneas o *registros* separadas por una marca de fin de línea (*newline*, en inglés). Cada línea consta de uno o más *datos* que es un conjunto de caracteres alfanuméricos que, en el procesamiento de lectura o escritura, se trata como una sola unidad. El acceso a los datos del archivo de texto procede en forma *secuencial*, esto es, se procesan línea por línea comenzando desde la primera línea hacia la última. Esto implica que no es posible acceder a una línea específica sin haber pasado por las anteriores.

Lectura de datos de un archivo.

Consideremos el siguiente archivo (*Datos.txt*) que contiene 3 columnas con números:

```
1   1   1
2   4   8
3   9  27
4  16  64
```

La extensión del archivo no es importante para el lenguaje: da lo mismo que ésta sea *.dat*, *.txt*, *.doc* o incluso que no tenga extensión.
Python dispone de varias formas de leer este archivo:

1. Línea por línea.
2. Como una lista.
3. Por contenido completo.

El siguiente programa muestra la manera de leer línea por línea:

```
print('Lectura de archivo: línea por línea.')
archivo = open('Datos.txt','r')
for linea in archivo:
  print(linea,end='')
archivo.close()
```

✍ En el ejemplo aparecen algunas funciones nuevas, necesarias para trabajar con archivos. Primero, se usa `open(`*nombre_de_archivo*`)` para "abrir" el archivo y vincularlo con una variable (`archivo`) que se usará para referirnos a él. El archivo deberá encontrarse en el directorio donde se está trabajando o, si estuviera en otra ubicación, deberá indicarse la ruta completa.

✍ La función `open()` incluye un segundo argumento (`'r'`) que indica que la apertura del archivo se realiza con el propósito de *lectura* (`'r'`, por *read*). Otra posibilidad es abrir el archivo para *escritura* (`'w'`, por *write*) o para el *agregado al final* (`'a'`, por *append*). En caso de querer abrir el archivo para lectura y escritura, se usa como segundo argumento `'r+'`. Por omisión, Python abre los archivos para lectura.

✍ El bucle `for` permite leer el archivo línea por línea hasta llegar a la última, donde se detiene la repetición.

✍ Cada línea leída es considerada una cadena de caracteres. Python no detecta en esta instancia si se trata de números, símbolos o letras.

✍ Noten que en el programa anterior, una de las funciones `print()` incluye el parámetro `end`. Esto se debe a que cada línea leída del archivo incluye al final de la cadena un salto de línea. Como, por defecto, cada llamada a `print()` imprime su salida seguida de un salto de línea (\n) introduciendo `end=' '` se le indica a Python que no agregue dicho salto. Así evitamos que la salida se produzca con un espaciado doble.

Al ejecutar el programa se obtiene la siguiente salida:

```
Lectura de archivo: línea por línea.
1   1   1
2   4   8
3   9  27
4  16  64
```

Si deseamos realizar la lectura *como una lista* modificamos el programa de la siguiente forma:

```
print('Lectura de archivo: en una lista.')
archivo = open('Datos.txt','r')
f_list = list(archivo)
print(f_list)
archivo.close()
```

El código anterior nos permite leer el archivo completo, guardando cada línea como un elemento de lista. Al ejecutar el programa, obtenemos la siguiente salida en pantalla (notar el carácter de salto de línea al final de cada línea):

```
Lectura de archivo: en una lista.
['1   1   1\n', '2   4   8\n',
  '3   9  27\n', '4  16  64\n']
```

Finalmente, para leer el archivo *por contenido completo* el programa debería ser el siguiente:

```
print('Lectura de archivo: en una variable.)
archivo = open('Datos.txt','r')
text = archivo.read()
print(text)
archivo.close()
```

con el cual obtendremos, al ejecutarlo:

```
Lectura de archivo: en una variable.
1   1   1
2   4   8
3   9  27
4  16  64
```

Como puede verse de los ejemplos anteriores, Python interpreta los datos leídos como cadenas de caracteres, lo que **no resulta conveniente cuando se pretende trabajar con valores numéricos**. Para el trabajo numérico, Python nos permite importar módulos, tales como `NumPy` o `Pandas`, con los que se facilita la operatoria.
El siguiente ejemplo muestra cómo realizar la lectura de los datos del archivo usando `NumPy`:

```
import numpy as np

archivo = open('Datos.txt','r')
datos = np.loadtxt(archivo)
print(datos)
archivo.close()
```

Ejecutando el programa se obtiene la siguiente salida en pantalla:

```
[[ 1.  1.  1.]
 [ 2.  4.  8.]
 [ 3.  9. 27.]
 [ 4. 16. 64.]]
```

Acá hay varios puntos para comentar:

✍ Numpy asigna el tamaño de la variable `datos` de forma automática, sin necesidad de que el usuario le indique cuántas filas tiene el archivo.

✍ `np.loadtxt` supone que el archivo está organizado en columnas, separadas por espacios, y que todas las columnas están ocupadas con valores numéricos (si falta un dato en alguna columna, la ejecución aborta señalando el error que se produjo).

✍ El arreglo resultante es (por omisión) del tipo real, incluso si los datos leídos fueron todos enteros.

El método `loadtxt` tiene una gran variedad de opciones; nosotros sólo veremos las más básicas.

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

`arch` es una cadena que indica el nombre del archivo de datos, si éste se encuentra en el directorio actual, o la ruta completa, si se encuentra en otro lugar.
`dtype` (opcional) permite explicitar el tipo de dato del arreglo resultante. El valor por omisión es `float`.
`comments` (opcional) permite indicar el símbolo del archivo que será interpretado como comentario. El valor por omisión es '#'.
`delimiter` (opcional) indica el carácter usado para separar valores. El valor por omisión es el espacio en blanco.
`skiprows` (opcional) permite indicar la cantidad de líneas iniciales que hay que saltar, incluyendo las líneas de comentarios. El valor por defecto es cero (0).
`usecols` (opcional) indica por medio de una lista, o tupla, qué columnas leer, siendo la primera la columna cero (0). El valor por defecto es `None`, que indica que deberán leerse todas las columnas.
El módulo `NumPy` resulta sumamente conveniente cuando los datos provienen de archivos como el que trabajamos en el ejemplo de más arriba. Cuando se trabaja con archivos provenientes de bases de datos (usualmente generados por distintos instrumentos de medición), el formato de los datos suele ser más complejo. Para facilitar la lectura en esos casos, Python cuenta con el módulo `Pandas`, que lamentablemente excede los contenidos de este curso.

Escritura hacia un archivo.

Vamos a procesar la información leída del archivo `Datos.txt` y enviar los resultados a otro archivo llamado `Salida.txt`. Para ilustrar la operatoria, vamos a requerir al programa lo siguiente:

1. Leer el archivo `Datos.txt`.
2. Para cada fila, calcular la suma de los datos de las columnas 2 y 3.
3. Guardar el resultado en un nuevo archivo, llamado `salida.txt`, que replicará el archivo de datos agregando una columna adicional con el resultado propiamente dicho.

Atención: la acción de escribir en un archivo puede significar que se sobreescriba uno existente, lo que implica perder el contenido previo.

```
import numpy as np

archivo = open('Datos.txt')
datos = np.loadtxt(archivo)
archivo.close()

# Suma las columnas 2 y 3.
suma = datos[:, 1] + datos[:, 2]

# Envia salida a archivo.
np.savetxt('salida.txt', np.column_stack(
    (datos[:, 0],suma))), fmt='%.6g')

print("Resultados guardados en salida.txt")
```

Ejercicio 5. El archivo Tabla-1.txt contiene una tabla con datos de temperatura, presión atmosférica y velocidad del viento medidos a lo largo de un día. Los datos están expresados en $°F$, pulgadas de mercurio y millas/hora.

Escribir un programa que lea la tabla, e imprima en pantalla, y guarde en un archivo, los datos expresados en el *Sistema Internacional* (mks). La conversión de todos los datos debe hacerse en una única función, que debe recibirlos en forma de arreglos.

Ejercicio 6. El archivo Tabla-2.txt contiene datos, basados en el *Modelo de Referencia Preliminar de la Tierra* (*PREM*, por sus siglas en inglés), de valores promedio de densidad $\rho\ [kg/m^3]$, módulo de volumen $K\ [GPa]$ y módulo de corte $\mu\ [GPa]$, para distintas profundidades de la Tierra.

Escribir un programa que lea los datos, compute y guarde en un archivo las velocidades de onda compresional V_p y de corte V_s correspondientes, sabiendo que,

$$V_p = \sqrt{\frac{K + \frac{4}{3}\mu}{\rho}} \qquad V_s = \sqrt{\frac{\mu}{\rho}}$$

Ambas velocidades deben ser computadas para todos los datos en una única función, y deben estar expresadas en m/s. ¿Qué se observa al aumentar la profundidad considerada? ¿Por qué motivo es nulo el módulo de corte en la anteúltima fila de la tabla?
Nota: usar el sistema SI de unidades (mks).