3-ES-Ficheros_e_interacción_con_el_sistema

July 1, 2021

1 Programación para la ciencia de datos

1.1 Unidad 3: Archivos e interacción con el sistema

1.1.1 Instrucciones de uso

Este documento es un notebook interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, se recomienda, en primer lugar, leer las explicaciones y el código que os proporcionamos. De esta manera tendréis un primer contacto con los conceptos que se exponen. Ahora bien, **¡la lectura es solo el principio!** Una vez que hayáis leído el contenido proporcionado, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes, que os permitan comprobar que habéis entendido su funcionalidad y explorar los detalles de la implementación. Por último, se recomienda también consultar la documentación enlazada para explorar con más detalle las funcionalidades de los módulos presentados.

[1]:
```
%load_ext pycodestyle_magic
```

[2]:
```
# Activamos las alertas de estilo
%pycodestyle_on
```

1.1.2 Introducción

En esta unidad veremos cómo podemos interactuar con el sistema operativo utilizando Python.

En primer lugar, nos centraremos en los ficheros: veremos cómo se puede leer el contenido de un archivo en Python, cómo se pueden crear nuevos archivos y escribir su contenido, cómo podemos realizar otras tareas básicas sobre ficheros (como cambiar el nombre de un fichero, borrarlo o crear carpetas), y algunos otros detalles que se deben tener en cuenta a la hora de trabajar con archivos desde Python.

Seguidamente, explicaremos las nociones básicas para trabajar con archivos comprimidos desde Python, explicando cómo leer y crear archivos comprimidos, así como otras tareas básicas como listar el contenido de un archivo comprimido.

A continuación presentaremos las funciones de carga de datos de más alto nivel, que permiten cargar conjuntos de datos desde ficheros sin necesidad de leer y procesar manualmente su contenido.

Después, veremos una alternativa para conseguir persistencia de datos de nuestras aplicaciones: la serialización con `pickle`.

Ya para terminar, explicaremos cómo podemos interactuar con una base de datos SQL desde nuestro código Python, y cómo podemos ejecutar otros programas también desde nuestro código.

A continuación se incluye la tabla de contenidos, que podéis utilizar para navegar por el documento:

1. Ficheros (con el módulo os)

```
<ul style="list-style-type:none">
```

1.1. Lectura y escritura de ficheros

```
    <ul style="list-style-type:none">
```

1.1.1. El path

1.1.2. El modo

1.1.3. Otros detalles en la apertura de ficheros

1.1.4. Lectura de ficheros grandes

```
    </ul>
```

1.2. Creación de carpetas

1.3. Borrar y renombrar

1.4. Funciones auxiliares de paths

1.5. Listado de directorios

1.6. Patrones de Unix shell

1.7. Obtención de metadatos de los ficheros

```
</ul>
```

2. Trabajo con archivos comprimidos

```
<ul style="list-style-type:none">
```

2.1. Lectura y escritura de ficheros comprimidos

```
    <ul style="list-style-type:none">
        <li> <a href='#2.1.1.-El-path-y-el-modo'> 2.1.1. El <em>path</em> y el modo </a> </li>
```

2.1.2. Otros detalles en la apertura de archivos

2.1.3. Lectura de archivos grandes

```
    </ul>
```

2.2. Borrar, renombrar y crear carpetas

2.3. Funciones auxiliares de paths, listados y metadatos

3. Lectura y escritura de ficheros con pandas

4. Serialización de datos

<ul style="list-style-type:none">

4.1. Serialización de datos con pickle

4.2. Consideraciones sobre la serialización de datos

5. Interacción con bases de datos

6. Interacción con el sistema operativo

7. Ejercicios para practicar

<ul style="list-style-type:none">

7.1. Soluciones a los ejercicios para practicar

8. Bibliografía

<ul style="list-style-type:none">

8.1. Bibliografía básica

8.2. Bibliografía adicional

Nota importante: La ejecución de este notebook modifica los archivos de la carpeta de la unidad 3 (se crean nuevos ficheros, se borran otros, se modifica el contenido de los existentes, etc.). Las explicaciones que se incluyen en este notebook concuerdan con la ejecución lineal de las celdas del notebook la primera vez que se hace esta ejecución. A partir de entonces, si volvéis a ejecutar el notebook (o si alteráis el orden de ejecución de las celdas para hacer pruebas), las explicaciones pueden no coincidir exactamente con los resultados que se producirán de la ejecución, ya que el estado inicial de los archivos no es el mismo.

Si queréis restaurar el estado inicial de todos los archivos que se alteran al ejecutar este notebook (para poder volver a ejecutar el notebook tal como estaba inicialmente), abrid una consola y ejecutad:

```
cd ~/prog_datasci_2/resources/unit_3 && rm -rf files_folder file_2.txt mem_data; unzip files_f
```

---

2 1. Ficheros (con el módulo `os`)

El módulo `os` provee de funciones para interactuar con el sistema operativo. Entre otras, incluye funciones para manipular archivos.

2.1 1.1. Lectura y escritura de ficheros

Para leer y/o escribir un archivo, lo primero que hay que hacer es abrirlo, especificando su *path* y el modo de apertura. Con el archivo abierto, podremos operar sobre él (ya sea leerlo o escribir en él). Finalmente, cuando hayamos finalizado la operación, habrá que cerrarlo, para liberar los recursos.

¡Es importante cerrar los ficheros que abrimos! Más allá de ser una buena práctica, hay que tener en cuenta que ciertos cambios en los ficheros pueden no ser visibles hasta que se cierren los archivos (ya que el sistema operativo implementa *buffers* para optimizar la gestión). Además, aunque Python tiene un sistema automático de cierre de recursos, puede que este falle, por lo que se quedan los archivos abiertos y, en consecuencia, se consume memoria RAM (y, por lo tanto, esto impacta negativamente en el rendimiento del programa), lo que contribuye a los contadores que controlan el número máximo de archivos abiertos.

[3]:
```
# Importamos el módulo os
import os

# Abrimos el archivo test_file.txt para escritura
out = open('files_folder/test_file.txt', 'w')
# Escribimos la palabra 'test' en el archivo
out.write("test")
# Cerramos el archivo
out.close()
```

Este es el flujo tradicional de trabajo con ficheros: se abre el archivo especificando el *path* y el modo, se opera con el archivo (en este caso, hemos escrito la palabra 'test') y, finalmente, se cierra el archivo. Después de ejecutar la celda anterior, abrid el archivo (con vuestro navegador de archivos del sistema operativo) y comprobad que efectivamente contiene la palabra 'test'.

Esta estructura de trabajo con archivos obliga al programador a recordar cerrar el archivo manualmente una vez que ha terminado de operar con él. Como alternativa a esta estructura tradicional, Python permite utilizar la sentencia `with`, que crea un contexto en el que el archivo está abierto, y libera así al programador de la tarea de recordar cerrar el archivo. Así, cuando la ejecución sale del contexto, Python cerrará automáticamente el archivo:

[4]:
```
# Abrimos el archivo test_file_2.txt para escritura
with open('files_folder/test_file_2.txt', 'w') as out:
    # Escribimos 'another test' en el archivo
    out.write("another test")

# Intentamos escribir más contenido en el mismo archivo
try:
    out.write("fail")
except ValueError as e:
    print(e)
```

```
I/O operation on closed file.
```

Fijaos cómo, si intentamos operar con el archivo fuera del contexto del `with`, se genera una excepción, ya que el archivo ya está cerrado.

2.1.1 1.1.1. El *path*

El primer argumento que recibe la función `open` (y el único que es obligatorio) es el *path*. El *path* puede ser absoluto o relativo al directorio donde se está ejecutando el código.

Indicaremos que el *path* es **absoluto** iniciándolo con una barra, `/` (que indica el directorio raíz).

En cambio, si el *path* comienza con un carácter, este será **relativo** al directorio donde se ejecuta el código. Así, en los dos ejemplos anteriores, los *paths* `test_file.txt` y `test_file_2.txt` eran relativos al directorio de ejecución, e indicaban que los archivos se encontraban directamente en el propio directorio.

Del mismo modo que en el sistema operativo, podemos utilizar `.` y `..` para referirnos, respectivamente, al directorio actual y al superior al actual en *paths* relativos. Especificaremos el *path* utilizando también barras `/` después de cada nombre de directorio.

2.1.2 1.1.2. El modo

En relación con el modo de apertura, Python reconoce los siguientes modificadores, que se pueden combinar entre ellos para especificar cómo y con qué finalidad se abre el fichero:

- `r`, modo de lectura (del inglés, ***r**eading*).
- `w`, modo de escritura (del inglés, ***w**riting*), sobrescribe el contenido del archivo si este ya existe, o bien crea el archivo si no existe.
- `x`, modo de creación e**x**clusiva.
- `a`, modo de escritura, escribe al final del archivo, después del contenido ya existente en el archivo (del inglés, ***a**ppend*), o bien crea el archivo si no existe.
- `b`, modo **b**inario.
- `t`, modo de **t**exto (modo predeterminado).
- `+`, modo de actualización (tanto para lectura como para escritura).

Python permite abrir archivo en modo binario (devolviendo los contenidos como bytes, sin descodificarlo) o en modo texto (devolviendo los contenidos como cadenas de texto, obtenidas de descodificar los bytes en función de la plataforma donde se ejecute el código o bien de la codificación especificada). Por defecto (es decir, si no se especifica el modo), los archivos se abren en modo texto, de manera que, por ejemplo, `r` y `rt` son equivalentes.

Tanto el modo `w` como el modo `a` permiten escribir en un archivo. La diferencia entre ellos radica en el tratamiento del contenido existente en el archivo: `w` sobrescribe el contenido del archivo, eliminando el contenido ya existente e incorporando el nuevo; en cambio, `a` escribe a continuación del contenido ya existente en el archivo, añadiendo el nuevo contenido después del contenido ya existente.

El modo de actualización, `+`, permite abrir un archivo para escribir y leer. Así, tanto `w+` como `r+` permitirán leer y escribir un archivo. La diferencia entre ambos modos recae en el comportamiento respecto al contenido existente en el archivo y a la existencia del propio archivo. Si especificamos `w+`, sobrescribiremos el contenido del archivo y crearemos el archivo si este no existe; en cambio, si especificamos `r+`, mantendremos el contenido del archivo y se generará un error si el archivo no existe.

A continuación se presentan algunos ejemplos del funcionamiento de los modos de apertura de archivos:

[5]:
```
# Intentamos abrir para lectura un archivo inexistente, lo
# que generará una excepción
p = 'files_folder/a_new_file.txt'
try:
    with open(p, 'r') as inp:
        pass
except FileNotFoundError as e:
    print(e)
```

```
[Errno 2] No such file or directory: 'files_folder/a_new_file.txt'
```

[6]:
```
# Intentamos abrir el mismo archivo para escritura (creando por tanto
# el archivo) y escribimos dos líneas
with open(p, 'w') as out:
    out.write("The file did not exist\n")
    out.write("It now contains two sentences.\n")
```

[7]:
```
# Volvemos a leer ahora el fichero (ahora se leerá correctamente, ya que
# ha sido creado en la celda anterior)
try:
    with open(p, 'r') as inp:
        content = inp.read()
        # Mostramos el contenido del fichero
        print(content)
except FileNotFoundError as e:
    print(e)
```

```
The file did not exist
It now contains two sentences.
```

[8]:
```
# Volvemos a escribir en el mismo archivo con el modo 'w', por lo
# que se sobrescribirá el contenido anterior
with open(p, 'w') as out:
    out.write("What happens if we write again?\n")
```

Abrid ahora el archivo `a_new_file.txt` de la carpeta `files_folder` y comprobad que solo contiene la frase `What happens if we write again?`, ya que el contenido anterior (`The file did not exist...`) se ha sobrescrito.

[9]:
```
# Volvemos a escribir ahora en el mismo fichero, pero utilizando el modo
# 'a', por lo que escribiremos a continuación del contenido ya existente.
with open(p, 'a') as out:
    out.write("And now, what happens if we write again?\n")
```

[10]:
```
from io import UnsupportedOperation
# Ahora intentaremos añadir otra frase en el archivo, y luego leer
# en el mismo archivo (lo que generará un error)
```

```
try:
    with open(p, 'a') as out:
        out.write("...and again?\n")
        content = out.read()
except UnsupportedOperation as e:
    print(e)
```

```
not readable
```

Si abrís ahora el archivo, veréis que contiene el texto siguiente, resultado de las ejecuciones de las celdas anteriores:

```
What happens if we write again?
And now, what happens if we write again?
... and again?
```

Probemos ahora el comportamiento del modo de lectura con actualización:

[11]:
```
# Abrimos el archivo en modo de lectura con actualización, escribimos una
# frase y leemos el contenido a partir del final de la escritura
with open(p, 'r+') as out:
    out.write("Trying the r+ mode!")
    content = out.read()
    print(content)
```

```
write again?
And now, what happens if we write again?
…and again?
```

En este último ejemplo, el archivo se ha abierto para lectura con actualización. Al escribir, escribimos por tanto al inicio del archivo (y sobrescribimos el contenido anterior conforme vamos escribiendo). Luego, al leer, leemos a partir de donde hemos terminado de escribir, por lo que solo se lee el contenido ya existente (que no hemos sobrescrito). El contenido del archivo es ahora por lo tanto:

```
Trying the r+ mode!write again?
And now, what happens if we write again?
...and again?
```

Fijaos cómo la primera línea contiene la frase que hemos escrito en la celda anterior, seguida de los caracteres que quedaban de la frase original.

2.1.3 1.1.3. Otros detalles en la apertura de ficheros

Además del *path* y el modo, la función `open` acepta otros argumentos opcionales, que gestionan el *buffering* de datos, la codificación, la gestión de los errores, la gestión del salto de línea, etc. El lector interesado puede consultar la documentación oficial de la función `open` (lectura opcional) para descubrir cómo funcionan estos argumentos y qué opciones se encuentran disponibles.

2.1.4 1.1.4. Lectura de ficheros grandes

Como hemos visto, el método `read` lee todo el contenido del archivo. Es evidente, pues, que utilizar este método puede conllevar problemas de memoria y de eficiencia en nuestro código, sobre todo cuando el fichero que vayamos a leer sea grande.

Una alternativa para la lectura de archivos es hacerla línea a línea, de modo que solo una línea del archivo se carga en memoria cada vez:

[12]:
```
from sys import getsizeof

p_big = 'files_folder/somehow_big_file.txt'

# Cargamos el archivo somehow_big_file.txt completo y mostramos
# el tamaño de la variable content en memoria
with open(p_big, 'r') as f:
    content = f.read()
    size_in_bytes = getsizeof(content)
    print("The size of the variable is: {} KB\n\n".format(
        size_in_bytes / 1024))

# Leemos el fichero línea a línea (y únicamente las 5 primeras líneas),
# mostrando el tamaño de la variable line
with open(p_big, 'r') as f:
    counter = 0
    for line in f:
        print(line)
        size_in_bytes = getsizeof(line)
        print("The size of the variable is: {} KB\n\n".format(
            size_in_bytes / 1024))
        counter += 1
        if counter == 5:
            break
```

```
The size of the variable is: 250.087890625 KB


This document gives coding conventions for the Python code comprising the
standard library in the main Python distribution. Please see the companion
informational PEP describing style guidelines for the C code in the C
implementation of Python [1].

The size of the variable is: 0.291015625 KB


This style guide evolves over time as additional conventions are identified and
past conventions are rendered obsolete by changes in the language itself.

The size of the variable is: 0.1982421875 KB
```

```
Many projects have their own coding style guidelines. In the event of any
conflicts, such project-specific guides take precedence for that project.

The size of the variable is: 0.1923828125 KB


One of Guido's key insights is that code is read much more often than it is
written. The guidelines provided here are intended to improve the readability of
code and make it consistent across the wide spectrum of Python code. As PEP 20
says, "Readability counts".

The size of the variable is: 0.3056640625 KB


A style guide is about consistency. Consistency with this style guide is
important. Consistency within a project is more important. Consistency within
one module or function is the most important.

The size of the variable is: 0.240234375 KB
```

2.2 1.2. Creación de carpetas

Hemos visto cómo podemos crear archivos abriéndolos en modo 'w' o 'a'. Para crear una carpeta o directorio, podemos utilizar los métodos `mkdir` o `makedirs`:

[13]:
```python
# Creamos la carpeta a_new_folder dentro de la carpeta
# existente files_folder
new_folder = 'files_folder/a_new_folder'
os.mkdir(new_folder)

# Creamos la carpeta an_empty_folder dentro de la carpeta
# existente files_folder
new_folder = 'files_folder/an_empty_folder'
os.mkdir(new_folder)
```

[14]:
```python
# Intentamos crear la carpeta 2 dentro de la carpeta 1 dentro de la carpeta
# a_new_folder que hemos creado en la celda anterior
try:
    new_folder = 'files_folder/a_new_folder/1/2'
    os.mkdir(new_folder)
except FileNotFoundError as e:
    print(e)
```

```
[Errno 2] No such file or directory: 'files_folder/a_new_folder/1/2'
```

La celda anterior genera una excepción, ya que la carpeta 'files_folder/a_new_folder/1' dentro de la cual queremos crear la carpeta '2' no existe. Si lo que queremos es crear tanto la carpeta '1' como la carpeta '2' dentro de la carpeta '1', podemos utilizar `makedirs`:

[15]:
```
# Creamos todas las carpetas que haya de la siguiente estructura de
# carpetas
new_folder = 'files_folder/a_new_folder/1/2/3'
os.makedirs(new_folder)
```

2.3 1.3. Borrar y renombrar

La librería `os` también provee de funciones para borrar archivos o directorios vacíos:

[16]:
```
# Borramos el archivo a_new_file.txt
os.remove(p)
```

[17]:
```
# Borramos la carpeta vacía an_empty_folder
empty_fold = 'files_folder/an_empty_folder'
os.rmdir(empty_fold)
```

[18]:
```
# Intentamos borrar la carpeta files_folder, lo que generará
# un error, ya que esta contiene archivos
try:
    non_empty_fold = 'files_folder'
    os.rmdir(non_empty_fold)
except OSError as e:
    print(e)
```

```
[Errno 39] Directory not empty: 'files_folder'
```

De manera análoga a la ejecución de un `rmdir` en una consola Linux, no podemos borrar una carpeta si esta contiene archivos. Si necesitamos borrar una carpeta y todos los archivos que esta contiene en Python, podemos o bien hacerlo a mano (borrando primero los archivos dentro de la carpeta y luego la propia carpeta), o bien utilizando alguna función que ya incorpore este comportamiento, como `rmtree` del módulo `shutil`.

La librería `os` también permite renombrar archivos y directorios a través de la función `rename`:

[19]:
```
# Renombramos el archivo original_file.txt a dest_file.txt
os.rename('files_folder/original_file.txt',
          'files_folder/dest_file.txt')
```

2.4 1.4. Funciones auxiliares de *paths*

Más allá de la propia lectura, escritura, borrado y renombrado de ficheros, trabajar con archivos suele requerir otras funciones, a menudo auxiliares, que permiten construir lógicas complejas de gestión de archivos.

Así, por ejemplo, es habitual tener que construir una ruta (un *path*) que indique dónde se encuentra un archivo en el sistema de ficheros a partir de fragmentos de esta ruta (por ejemplo, el nombre de

la carpeta o ruta de carpetas, el nombre del archivo, etc.).

El submódulo `path` (del módulo `os`) implementa la función `join`, que une diferentes partes de la ruta de un archivo de manera *inteligente*, considerando el separador de directorios del sistema en el que se ejecuta el código. Es una buena práctica utilizar esta función a la hora de especificar dónde se encuentra un archivo en vez de concatenar manualmente las diferentes partes de la ruta, ya que esto mejora la compatibilidad del código en diferentes sistemas.

[20]:
```
# Unimos diferentes partes de un path con join
path = "/home"
full_path = os.path.join(path, "User/Desktop", "filename.txt")
print(full_path)
```

```
/home/User/Desktop/filename.txt
```

[21]:
```
another_full_path = os.path.join(path, "User/Public/", "Documents", "")
print(another_full_path)
```

```
/home/User/Public/Documents/
```

En el primer caso, el *path* generado (`full_path`) correspondía a un fichero, mientras que en el segundo caso el *path* obtenido (`another_full_path`) era el de una carpeta o directorio (algo que podemos deducir por la `/` final). Tened en cuenta que, para indicar que el *path* que queremos obtener es el de una carpeta, hemos indicado como último elemento de la llamada a `join` una cadena vacía (`""`).

Por otra parte, a partir de un *path* podemos obtener el nombre de la carpeta, el nombre del archivo o la extensión del fichero con las funciones `dirname`, `basename` y `splitext`, respectivamente.

[22]:
```
# Obtenemos el nombre del directorio
os.path.dirname(full_path)
```

[22]:
```
'/home/User/Desktop'
```

[23]:
```
# Obtenemos el nombre del archivo
os.path.basename(full_path)
```

[23]:
```
'filename.txt'
```

[24]:
```
# Separamos la extensión del path
os.path.splitext(full_path)
```

[24]:
```
('/home/User/Desktop/filename', '.txt')
```

Otra funcionalidad a menudo necesaria en el tratamiento de ficheros es la de poder comprobar si un *path* existe, o si este corresponde a un archivo normal o a una carpeta. Estas tres comprobaciones las podemos hacer con las funciones `exists`, `isfile` e `isdir`, respectivamente.

[25]:
```
e = os.path.exists(full_path)
print("Path {} exists?\t\t\t{}".format(full_path, e))
```

```
path_1 = "./"
e = os.path.exists(path_1)
print("Path {} exists?\t\t\t\t\t\t\t{}".format(path_1, e))

path_2 = "3-ES-Ficheros_e_interacción_con_el_sistema.ipynb"
e = os.path.exists(path_2)
print("Path {} exists?\t{}".format(path_2, e))

path_3 = "/home"
e = os.path.exists(path_3)
print("Path {} exists?\t\t\t\t\t\t{}".format(path_3, e))
```

```
Path /home/User/Desktop/filename.txt exists?                    False
Path ./ exists?                                                 True
Path 3-ES-Ficheros_e_interacción_con_el_sistema.ipynb exists?   True
Path /home exists?                                              True
```

El primer *path* no corresponde a ningún path existente, por lo que `exists` devuelve `False`. En cambio, tanto los *paths* relativos `./` y `3-ES-Ficheros_e_interacción_con_el_sistema.ipynb` como el *path* absoluto`/home` existen en el sistema, de modo que `exists` devuelve `True`. Es importante notar que `exists` reconoce tanto archivos normales como carpetas o directorios. En cambio, la función `isfile` devolverá `True` únicamente si el *path* existe y este es un archivo normal:

[26]:
```
is_f = os.path.isfile(full_path)
print("Path {} is file?\t\t\t{}".format(full_path, is_f))

is_f = os.path.isfile(path_1)
print("Path {} is file?\t\t\t\t\t\t{}".format(path_1, is_f))

is_f = os.path.isfile(path_2)
print("Path {} is file?\t{}".format(path_2, is_f))

is_f = os.path.isfile(path_3)
print("Path {} is file?\t\t\t\t\t\t{}".format(path_3, is_f))
```

```
Path /home/User/Desktop/filename.txt is file?                   False
Path ./ is file?                                                False
Path 3-ES-Ficheros_e_interacción_con_el_sistema.ipynb is file?  True
Path /home is file?                                             False
```

En este caso, `isfile` reconoce únicamente el tercer *path* como archivo, ya que el primero no existe y los otros dos corresponden a carpetas.

Veamos ahora el resultado de llamar `isdir` con los mismos *paths*:

[27]:
```
is_d = os.path.isdir(full_path)
print("Path {} is directory?\t\t{}".format(full_path, is_d))
```

```
is_d = os.path.isdir(path_1)
print("Path {} is directory?\t\t\t\t\t\t{}".format(path_1, is_d))

is_d = os.path.isdir(path_2)
print("Path {} is directory?\t{}".format(path_2, is_d))

is_d = os.path.isdir(path_3)
print("Path {} is directory?\t\t\t\t\t{}".format(path_3, is_d))
```

```
Path /home/User/Desktop/filename.txt is directory?              False
Path ./ is directory?                                           True
Path 3-ES-Ficheros_e_interacción_con_el_sistema.ipynb is directory?     False
Path /home is directory?                                        True
```

Ahora, de nuevo, obtenemos `False` para el primer *path*, ya que este no existe, y obtenemos`True` para los dos *paths* que representan directorios, `./` y `/home`.

2.5 1.5. Listado de directorios

Para obtener todos los contenidos que hay en un directorio podemos utilizar la función `scandir`:

[28]:
```
# Mostramos todas las entradas de la carpeta files_folder
folder_name = 'files_folder/'
with os.scandir(folder_name) as dir_list:
    for entry in dir_list:
        print(entry.name)
```

```
test_file_2.txt
dest_file.txt
somehow_big_file.txt
file_2.txt
zip_with_multiple_files.zip
a_folder_in_a_zip
a_new_folder
test_file.txt
echo_script.sh
file_1.txt
a_number.txt
echo_read_script.sh
endless_script.sh
```

Si queremos obtener una lista de los archivos de una carpeta, podemos combinar `scandir` con la función `isfile` que hemos visto en una sección anterior:

[29]:
```
# Mostramos todos los archivos de la carpeta files_folder
with os.scandir(folder_name) as dir_list:
    for entry in dir_list:
        if os.path.isfile(entry.path):
            print(entry.name)
```

```
test_file_2.txt
dest_file.txt
somehow_big_file.txt
file_2.txt
zip_with_multiple_files.zip
test_file.txt
echo_script.sh
file_1.txt
a_number.txt
echo_read_script.sh
endless_script.sh
```

Alternativamente, podemos utilizamos el método `is_file` de las propias entradas, ya que `scandir` devuelve un iterador que recorre objetos de tipo `DirEntry`, que implementan este método:

[30]:
```
# Mostramos todos los archivos de la carpeta files_folder
with os.scandir(folder_name) as dir_list:
    for entry in dir_list:
        if entry.is_file():
            print(type(entry), entry.name)
```

```
<class 'posix.DirEntry'> test_file_2.txt
<class 'posix.DirEntry'> dest_file.txt
<class 'posix.DirEntry'> somehow_big_file.txt
<class 'posix.DirEntry'> file_2.txt
<class 'posix.DirEntry'> zip_with_multiple_files.zip
<class 'posix.DirEntry'> test_file.txt
<class 'posix.DirEntry'> echo_script.sh
<class 'posix.DirEntry'> file_1.txt
<class 'posix.DirEntry'> a_number.txt
<class 'posix.DirEntry'> echo_read_script.sh
<class 'posix.DirEntry'> endless_script.sh
```

2.6 1.6. Patrones de Unix *shell*

Combinando la posibilidad de listar el contenido de directorios con las expresiones regulares que vimos en la unidad de estructuras de datos avanzadas (o, incluso, tal vez solo con otras funciones básicas de cadenas, como `startswith`), podemos seleccionar un subconjunto de archivos que cumplan alguna característica concreta. Así, por ejemplo, podríamos procesar solo los ficheros con extensión .txt de la misma carpeta que en el ejemplo anterior, haciendo:

[31]:
```
# Mostramos todos los archivos de la carpeta files_folder
with os.scandir(folder_name) as dir_list:
    for entry in dir_list:
        if entry.is_file() and entry.name.endswith(".txt"):
            print(entry.name)
```

```
test_file_2.txt
dest_file.txt
```

```
somehow_big_file.txt
file_2.txt
test_file.txt
file_1.txt
a_number.txt
```

Una alternativa para hacer este tipo de operaciones es utilizar el módulo `glob`, que permite obtener los paths que siguen un determinado patrón especificado utilizando la sintaxis de una *shell* de Unix (es decir, la misma sintaxis que utilizaríamos si estuviéramos navegando por el sistema de ficheros desde una *shell*).

[32]:
```
import glob

# Obtenemos una lista con los nombres de los ficheros de notebook de Python
glob.glob('*.ipynb')
```

[32]:
```
['3-CAT-Fitxers_i_interacció_amb_el_sistema.ipynb',
 '3-ES-Ficheros_e_interacción_con_el_sistema.ipynb']
```

[33]:
```
# Listamos el contenido de la carpeta files_folder
glob.glob('./files_folder/*')
```

[33]:
```
['./files_folder/test_file_2.txt',
 './files_folder/dest_file.txt',
 './files_folder/somehow_big_file.txt',
 './files_folder/file_2.txt',
 './files_folder/zip_with_multiple_files.zip',
 './files_folder/a_folder_in_a_zip',
 './files_folder/a_new_folder',
 './files_folder/test_file.txt',
 './files_folder/echo_script.sh',
 './files_folder/file_1.txt',
 './files_folder/a_number.txt',
 './files_folder/echo_read_script.sh',
 './files_folder/endless_script.sh']
```

[34]:
```
# Listamos los archivos .txt que hay dentro de la carpeta files_folder
glob.glob('./files_folder/*.txt')
```

[34]:
```
['./files_folder/test_file_2.txt',
 './files_folder/dest_file.txt',
 './files_folder/somehow_big_file.txt',
 './files_folder/file_2.txt',
 './files_folder/test_file.txt',
 './files_folder/file_1.txt',
 './files_folder/a_number.txt']
```

[35]:
```
# Listamos todos los archivos que hay en el path actual, buscando
# recursivamente dentro de las carpetas
glob.glob('**/*', recursive=True)
```

[35]:
```
['mem_data',
 'README.md',
 '3-CAT-Fitxers_i_interacció_amb_el_sistema.ipynb',
 'file_2.txt',
 'pdf',
 'data',
 '3-ES-Ficheros_e_interacción_con_el_sistema.ipynb',
 'files_folder',
 'files_folder.zip',
 'mem_data/20210701.zip.zip',
 'mem_data/20210701.zip',
 'mem_data/20210701',
 'mem_data/20210701/16_25_50',
 'mem_data/20210701/16_26_18',
 'mem_data/20210701/16_25_44',
 'mem_data/20210701/16_26_08',
 'mem_data/20210701/16_23_36',
 'mem_data/20210701/16_26_15',
 'mem_data/20210701/16_25_29',
 'mem_data/20210701/16_26_05',
 'mem_data/20210701/16_22_48',
 'mem_data/20210701/16_22_54',
 'mem_data/20210701/16_26_21',
 'mem_data/20210701/16_23_39',
 'mem_data/20210701/16_23_00',
 'mem_data/20210701/16_25_35',
 'mem_data/20210701/16_23_33',
 'mem_data/20210701/16_23_24',
 'mem_data/20210701/16_23_42',
 'mem_data/20210701/16_25_32',
 'mem_data/20210701/16_25_53',
 'mem_data/20210701/16_23_06',
 'mem_data/20210701/16_26_12',
 'mem_data/20210701/16_23_27',
 'mem_data/20210701/16_25_47',
 'mem_data/20210701/16_25_23',
 'mem_data/20210701/16_23_21',
 'mem_data/20210701/16_23_15',
 'mem_data/20210701/16_25_26',
 'mem_data/20210701/16_25_59',
 'mem_data/20210701/16_23_45',
 'mem_data/20210701/16_23_30',
 'mem_data/20210701/16_23_03',
```

```
 'mem_data/20210701/16_25_38',
 'mem_data/20210701/16_22_57',
 'mem_data/20210701/16_25_56',
 'mem_data/20210701/16_25_41',
 'mem_data/20210701/16_23_12',
 'mem_data/20210701/16_23_09',
 'mem_data/20210701/16_26_02',
 'mem_data/20210701/16_22_51',
 'mem_data/20210701/16_23_18',
 'pdf/3-ES-Ficheros_e_interacción_con_el_sistema.pdf',
 'pdf/3-CAT-Fitxers_i_interacció_amb_el_sistema.pdf',
 'data/marvel-wikia-data.csv',
 'files_folder/test_file_2.txt',
 'files_folder/dest_file.txt',
 'files_folder/somehow_big_file.txt',
 'files_folder/file_2.txt',
 'files_folder/zip_with_multiple_files.zip',
 'files_folder/a_folder_in_a_zip',
 'files_folder/a_new_folder',
 'files_folder/test_file.txt',
 'files_folder/echo_script.sh',
 'files_folder/file_1.txt',
 'files_folder/a_number.txt',
 'files_folder/echo_read_script.sh',
 'files_folder/endless_script.sh',
 'files_folder/a_folder_in_a_zip/file_3.txt',
 'files_folder/a_new_folder/1',
 'files_folder/a_new_folder/1/2',
 'files_folder/a_new_folder/1/2/3']
```

2.7 1.7. Obtención de metadatos de los ficheros

El submódulo `path` también contiene funciones para obtener metadatos de los archivos, por ejemplo su tamaño (`getsize`), o el instante de la última modificación (`getmtime`).

[36]:
```
# Obtenemos el tamaño del archivo p_big
p_big_size = os.path.getsize(p_big)
print("The file {} is {} KB".format(p_big, p_big_size / 1024))
```

```
The file files_folder/somehow_big_file.txt is 250.0400390625 KB
```

[37]:
```
from datetime import datetime

# Obtenemos la fecha de última modificación
unx_ts_mtime = os.path.getmtime(p_big)
print("The last modification time is: {} (unix ts)".format(unx_ts_mtime))
```

```
print("which is: {}".format(
    datetime.utcfromtimestamp(unx_ts_mtime).strftime('%Y-%m-%d %H:%M:%S')))
```

```
The last modification time is: 1584526066.0 (unix ts)
which is: 2020-03-18 10:07:46
```

También podemos obtener otros metadatos, como los que obtendríamos haciendo un `stat` de Linux sobre el archivo:

[38]:
```
import stat

# Mostramos los bits de protección del archivo
print(oct(stat.S_IMODE(os.stat(p_big).st_mode)))
```

```
0o664
```

Si tenéis curiosidad por saber cómo funcionan los bits de permiso de los ficheros en unix, os recomendamos leer las tres partes de la serie de artículos sobre los permisos (1, 2, y 3), todas ellas lecturas opcionales.

3 2. Trabajo con ficheros comprimidos

Un archivo comprimido es un archivo que contiene uno o varios archivos y/o carpetas, codificados de tal manera que ocupan menos espacio en disco que los archivos originales. Se utilizan ficheros comprimidos, por ejemplo, para transferir más rápidamente contenido a través de una red, o para aprovechar mejor el espacio de disco.

Distinguimos entre compresión **sin pérdida** y compresión **con pérdida**. La compresión sin pérdida se caracteriza por permitir recuperar la totalidad de los datos de los archivos originales a partir de los ficheros comprimidos. En cambio, en la compresión con pérdida, se pueden perder algunos bits de información. Se utiliza compresión con pérdida principalmente en imágenes y vídeo, donde a menudo las personas que visualizan este contenido no llegan a notar la pérdida. En cambio, en ficheros de texto se suele utilizar compresión sin pérdida.

En este notebook, se explicará cómo trabajar con archivos comprimidos zip, uno de los formatos más populares de compresión sin pérdida, utilizando el módulo `zipfile`. La otra alternativa muy popular para comprimir sin pérdida es usar gzip. El notebook no incluye explicaciones sobre cómo trabajar con archivos gzip, ya que el funcionamiento es muy similar al que se describe para zip. El lector interesado puede leer la documentación del módulo `gzip` para conocer las funciones que permiten trabajar con archivos gzip desde Python (lectura opcional) .

3.1 2.1. Lectura y escritura de ficheros comprimidos

De manera similar a los archivos genéricos, lo primero que hay que hacer para leer o crear un archivo comprimido zip es abrirlo, especificando su *path* y el modo de apertura. Cuando hayamos finalizado la operación, también habrá que cerrarlo. Podemos utilizar la misma sintaxis que hemos visto en el apartado anterior para gestionar la apertura y el cierre de los archivos comprimidos:

[39]:
```
import zipfile as zf
```

```
zip_file = 'files_folder/compressed_file.zip'
# Creamos el fichero zip_file especificando el modo de compresión ZIP_DEFLATED
with zf.ZipFile(zip_file, 'w', compression=zf.ZIP_DEFLATED) as zip_f:
    # Añadimos el fichero p_big al zip
    zip_f.write(p_big)
```

En la celda anterior, hemos abierto un archivo zip en modo de escritura (especificado por 'w'), de manera que el archivo se creará si no existe, o se sobrescribirá si ya existía; y hemos especificado el método de compresión `ZIP_DEFLATED`. Con el archivo abierto, le hemos añadido el archivo ya existente `p_big`. Comprobamos el resultado de la compresión:

[40]:
```
# Obtenemos el tamaño del archivo original p_big
p_big_size = os.path.getsize(p_big)
print("The file {} is {} KB".format(p_big, p_big_size / 1024))

# Obtenemos el tamaño del archivo zip_file (que contiene el archivo
# p_big comprimido)
zip_file_size = os.path.getsize(zip_file)
print("The file {} is {} KB".format(zip_file, zip_file_size / 1024))
```

```
The file files_folder/somehow_big_file.txt is 250.0400390625 KB
The file files_folder/compressed_file.zip is 1.876953125 KB
```

Así pues, el archivo original de 249 KB ha pasado a ocupar solo 1.3 KB al ser comprimido.

Comprobamos ahora que podemos recuperar el archivo original a partir del archivo comprimido. En primer lugar, vamos a calcular un hash del contenido del archivo, para poder asegurar que el archivo que recuperaremos es exactamente el mismo que el archivo original.

[41]:
```
import hashlib


def sha256_file_content(p):
    """
    Returns the sha256 hash of the content of the file p.
    """
    with open(p, 'rb') as f:
        content = f.read()
        h = hashlib.sha256(content).hexdigest()
    return h


# Obtenemos el hash del contenido del archivo p_big
orig_hash = sha256_file_content(p_big)
print("sha256: {}".format(orig_hash))
```

```
sha256: 2093ab905569669d33c944efc6a528bcb6f9cd5d2c08a203e70f4d2a27f17a29
```

[42]:
```
# Borramos el archivo p_big
os.remove(p_big)

# Comprobamos que se ha borrado
is_f = os.path.isfile(p_big)
print("Path {} is file? {}".format(p_big, is_f))
```

```
Path files_folder/somehow_big_file.txt is file? False
```

[43]:
```
# Abrimos el archivo zip en modo de lectura
with zf.ZipFile(zip_file, 'r') as zip_f:
    # Mostramos el contenido del zip
    print(zip_f.printdir())
    # Descomprimimos todo el contenido del zip
    zip_f.extractall()
```

```
File Name                                             Modified             Size
files_folder/somehow_big_file.txt             2020-03-18 11:07:46       256041
None
```

[44]:
```
# Comprobamos que el archivo se ha descomprimido
is_f = os.path.isfile(p_big)
print("Path {} is file? {}".format(p_big, is_f))

# Comprobamos que el contenido del fichero es exactamente el mismo
uncomp_hash = sha256_file_content(p_big)
print("sha256: {}".format(uncomp_hash))
print("Hash are equal?: {}".format(uncomp_hash == orig_hash))
```

```
Path files_folder/somehow_big_file.txt is file? True
sha256: 2093ab905569669d33c944efc6a528bcb6f9cd5d2c08a203e70f4d2a27f17a29
Hash are equal?: True
```

3.1.1 2.1.1. El *path* y el modo

En cuanto al *path* de los ficheros comprimidos, las mismas consideraciones que se han hecho sobre ficheros son aplicables en el caso de los archivos comprimidos.

Ya hemos visto cómo los modos de lectura 'r' y escritura 'w' de archivos zip funcionan de manera similar a los del módulo `os` . Así, también disponemos de un modo de concatenación, 'a' (del inglés, *append*), que permite añadir contenido a un zip sin sobrescribir el contenido existente; y del modo de creación y escritura exclusiva, 'x', que, a diferencia del modo de escritura, generará una excepción si el archivo que se intenta crear ya existe.

3.1.2 2.1.2. Otros detalles en la apertura de ficheros

Además del *path* y el modo, `ZipFile` acepta otros argumentos opcionales, que gestionan el formato de compresión, la extensión para permitir archivos mayores de 4 GB, y la comprobación de *timestamps*. El lector interesado puede consultar la documentación oficial de la clase `ZipFile`

(lectura opcional) para descubrir cómo funcionan estos argumentos y qué opciones se encuentran disponibles.

3.1.3 2.1.3. Lectura de archivos grandes

Hemos visto que podemos utilizar el método `extractall` para extraer todo el contenido de un zip. Ahora bien, si el archivo zip es muy grande, contiene varios archivos, y solo necesitamos un subconjunto de estos, será más eficiente descomprimir únicamente los archivos que necesitemos:

[45]:
```
# Abrimos el archivo zip en modo de lectura
zip_with_multiple_files = "files_folder/zip_with_multiple_files.zip"
with zf.ZipFile(zip_with_multiple_files, 'r') as zip_f:
    # Descomprimimos únicamente el archivo file_2.txt
    zip_f.extract("file_2.txt")
```

Fijaos cómo, para extraer un único archivo, tendremos que saber cómo se llama este archivo, información que podemos saber ya o que podemos obtener, como hemos visto, con `printdir`. Más adelante, veremos también otra alternativa para obtener esta información.

3.2 2.2. Borrar, renombrar y crear carpetas

El módulo `zipfile` no permite, de manera nativa, borrar o renombrar ficheros que se encuentran dentro de un zip. Por tanto, lo que habrá que hacer si necesitamos realizar estas acciones será implementar manualmente a partir de los modos de lectura y escritura que hemos visto en el apartado anterior. Así, por ejemplo, si queremos borrar un cierto archivo de un zip, habrá que descomprimir el zip y volver a crear un nuevo zip con el mismo contenido que el archivo original, pero sin incluir el fichero que queremos borrar.

Para crear una carpeta dentro de un archivo zip, procederemos a crear la carpeta fuera de este, y lo añadiremos después como si fuera un archivo ordinario, por ejemplo, con la función `write`.

3.3 2.3. Funciones auxiliares de paths, listados y metadatos

De manera análoga a las funciones que permitían comprobar si un archivo era un directorio o un archivo normal, el módulo `zipfile` dispone del método `is_zipfile`, que permite comprobar si un archivo es un archivo zip válido:

[46]:
```
izf = zf.is_zipfile(zip_file)
print("The file {} is a zip file?: {}".format(zip_file, izf))
```

```
The file files_folder/compressed_file.zip is a zip file?: True
```

Por otra parte, ya hemos visto cómo el método `printdir` nos permite obtener un listado de los contenidos de un archivo zip:

[47]:
```
# Abrimos el archivo zip en modo de lectura
zip_with_mult_files = "files_folder/zip_with_multiple_files.zip"
with zf.ZipFile(zip_with_mult_files, 'r') as zip_f:
    # Mostramos el contenido del zip
    print(zip_f.printdir())
```

```
File Name                                             Modified             Size
a_folder_in_a_zip/                             2020-01-20 16:42:00            0
a_folder_in_a_zip/file_3.txt                   2020-01-20 16:42:00           24
file_1.txt                                     2020-01-20 16:41:14          823
file_2.txt                                     2020-01-20 16:41:42           17
None
```

Otra alternativa para obtener información sobre los contenidos de un zip es utilizar la clase `ZipInfo`, que nos devuelve precisamente este tipo de información:

[48]:
```python
with zf.ZipFile(zip_with_mult_files, 'r') as zip_f:
    # Obtenemos un objeto ZipInfo del archivo zip_with_mult_files
    info_list = zip_f.infolist()

    # Para cada archivo dentro del zip, mostramos la
    # información del fichero
    for info in info_list:
        print("Filename: {}".format(info.filename))
        print("\tFile size: {} bytes".format(info.file_size))
        print("\tIs dir?: {}".format(info.is_dir()))
        print("\tDate and time: {}".format(info.date_time))
        print("\tCompression type: {}".format(info.compress_type))
        print("\tCRC: {}\n".format(info.CRC))
```

```
Filename: a_folder_in_a_zip/
        File size: 0 bytes
        Is dir?: True
        Date and time: (2020, 1, 20, 16, 42, 0)
        Compression type: 0
        CRC: 0

Filename: a_folder_in_a_zip/file_3.txt
        File size: 24 bytes
        Is dir?: False
        Date and time: (2020, 1, 20, 16, 42, 0)
        Compression type: 0
        CRC: 2817114606

Filename: file_1.txt
        File size: 823 bytes
        Is dir?: False
        Date and time: (2020, 1, 20, 16, 41, 14)
        Compression type: 8
        CRC: 3521977432

Filename: file_2.txt
        File size: 17 bytes
        Is dir?: False
```

```
Date and time: (2020, 1, 20, 16, 41, 42)
Compression type: 0
CRC: 742541709
```

Hay que remarcar que un archivo zip es un archivo en toda regla. Por lo tanto, podemos obtener metadatos del propio archivo zip con las funciones que hemos visto anteriormente del módulo `os`. Por ejemplo, podríamos obtener el tamaño del zip utilizando `getsize`:

[49]:
```
# Obtenemos el tamaño del archivo zip_with_mult_files
s_zip_with_mult_files = os.path.getsize(zip_with_mult_files)
print("The file {} is {} KB".format(zip_with_mult_files,
                                    s_zip_with_mult_files / 1024))
```

```
The file files_folder/zip_with_multiple_files.zip is 1.0263671875 KB
```

4 3. Lectura y escritura de ficheros con pandas

En algunas situaciones, podremos utilizar librerías de más alto nivel para leer y/o escribir archivos. Así, por ejemplo, la librería pandas permite cargar datos de un archivo CSV a un *dataframe* a través de la función `read_csv`.

Ahora cargaremos los datos del fichero `marvel-wikia-data.csv`, que contiene datos sobre personajes de cómic de Marvel. El conjunto de datos original en el que se basa el que utilizaremos fue creado por la web FiveThirtyEight, que escribe artículos basados en datos sobre deportes y noticias, y que pone a disposición pública los conjuntos de datos que recoge para sus artículos.

[50]:
```
import pandas as pd

# Cargamos los datos del fichero "marvel-wikia-data.csv" en un 'dataframe'
data = pd.read_csv("data/marvel-wikia-data.csv")
```

La función `read_csv` acepta un gran abanico de parámetros opcionales que permiten configurar con detalle cómo se tiene que realizar la importación del archivo csv. A continuación, veremos algunos de ellos, ajustando la importación de los datos de Marvel.

Fijémonos, en primer lugar, en la importación de las columnas numéricas:

[51]:
```
data.describe()
```

[51]:

	page_id	APPEARANCES	Year
count	31.000000	30.000000	31.000000
mean	12936.774194	1677.466667	1.885871
std	21056.994013	821.949228	0.358674
min	1073.000000	0.000000	0.000000
25%	1835.000000	1179.000000	1.961500
50%	2223.000000	1322.500000	1.963000
75%	8911.000000	2001.500000	1.964000
max	65255.000000	4043.000000	1.975000

Como se puede apreciar, ha habido un problema en la importación de los años en los que los personajes aparecen por primera vez en los cómics, ya que la media es 1.88 y, en cambio, los cómics aparecieron en el siglo XX. Observando el contenido del archivo csv:

```
page_id, name, urlslug, ID, ALIGN, EYE, HAIR, SEX, GSM, ALIVE, Appearances, FIRST Appearance, Y
1678, Spider-Man (Peter Parker), \ / Spider-Man_ (Peter_Parker), Secreto Identity, Good Charact
7139, Captain America (Steven Rogers), \ / Captain_America_ (Steven_Rogers), Public Identity, G
...
```

podemos ver cómo los años se expresan usando un punto como separador de millares, que pandas está interpretando como separador decimal. Por lo tanto, para asegurar que los años se importan correctamente, podemos indicar que el separador de millares sea el punto (`.`) con el parámetro `thousands`:

[52]:
```
data = pd.read_csv("data/marvel-wikia-data.csv", thousands=".")
data.describe()
```

[52]:

	page_id	APPEARANCES	Year
count	31.000000	30.000000	31.000000
mean	12936.774194	1677.466667	1885.870968
std	21056.994013	821.949228	358.674016
min	1073.000000	0.000000	0.000000
25%	1835.000000	1179.000000	1961.500000
50%	2223.000000	1322.500000	1963.000000
75%	8911.000000	2001.500000	1964.000000
max	65255.000000	4043.000000	1975.000000

Ahora, parece que los años se han importado correctamente, aunque la media sigue siendo más baja del valor que esperaríamos. Observando los datos para el campo año, podemos comprobar cómo la última fila contiene un 0, valor que hace bajar la media:

[53]:
```
data[["Year"]].tail()
```

[53]:

	Year
26	1963
27	1963
28	1964
29	1975
30	0

Podemos indicar que queremos omitir esta fila en la carga de datos con el parámetro `skiprows`:

[54]:
```
data = pd.read_csv("data/marvel-wikia-data.csv", thousands=".", skiprows=[31])
data[["Year"]].tail()
```

[54]:

	Year
25	1963
26	1963
27	1963

```
28  1964
29  1975
```

Como podemos observar, esto hace que la media suba de 1885.87 a 1948.73, y el mínimo pase de ser 0 (el valor de la fila que hemos omitido) a 1528:

[55]:
```
data.describe()
```

[55]:
```
            page_id  APPEARANCES         Year
count     30.000000    29.000000    30.000000
mean   13034.700000  1735.310345  1948.733333
std    21409.788049   771.859772    79.730552
min     1073.000000  1047.000000  1528.000000
25%     1834.000000  1230.000000  1962.000000
50%     2194.500000  1338.000000  1963.000000
75%     7652.000000  2017.000000  1964.000000
max    65255.000000  4043.000000  1975.000000
```

Si seguimos observando los datos que se han cargado, podemos observar también cómo aparecen varias inversa (\), por ejemplo, antes de comillas dobles (") o de las barras (/):

[56]:
```
data.head(n=5)
```

[56]:
```
   page_id                                name  \
0     1678           Spider-Man (Peter Parker)
1     7139     Captain America (Steven Rogers)
2    64786  Wolverine (James \"Logan\" Howlett)
3     1868    Iron Man (Anthony \"Tony\" Stark)
4     2460                  Thor (Thor Odinson)

                                urlslug                ID  \
0             \/Spider-Man_(Peter_Parker)   Secret Identity
1       \/Captain_America_(Steven_Rogers)   Public Identity
2  \/Wolverine_(James_%22Logan%22_Howlett)   Public Identity
3    \/Iron_Man_(Anthony_%22Tony%22_Stark)   Public Identity
4                  \/Thor_(Thor_Odinson)  No Dual Identity

                ALIGN         EYE        HAIR              SEX       GSM  \
0     Good Characters  Hazel Eyes  Brown Hair  Male Characters  dontknow
1     Good Characters   Blue Eyes  White Hair  Male Characters       NaN
2  Neutral Characters   Blue Eyes  Black Hair  Male Characters  dontknow
3     Good Characters   Blue Eyes  Black Hair  Male Characters       NaN
4     Good Characters   Blue Eyes  Blond Hair  Male Characters       NaN

               ALIVE  APPEARANCES FIRST APPEARANCE  Year
0  Living Characters       4043.0           Aug-62  1962
1  Living Characters       3360.0           Mar-41  1941
2  Living Characters       3061.0           Oct-74  1974
```

```
3  Living Characters       2961.0           Mar-63  1963
4  Living Characters       2258.0           Nov-50  1950
```

Estas barras invertidas se están usando para escapar caracteres especiales, lo que también podemos indicar a la función de carga del csv, con el parámetro `escapechar`:

[57]:
```
data = pd.read_csv("data/marvel-wikia-data.csv", thousands=".", skiprows=[31],
                   escapechar="\\")
data.head(n=5)
```

[57]:
```
   page_id                                  name  \
0     1678            Spider-Man (Peter Parker)
1     7139      Captain America (Steven Rogers)
2    64786  Wolverine (James "Logan"" Howlett)"
3     1868    Iron Man (Anthony "Tony"" Stark)"
4     2460                  Thor (Thor Odinson)

                                urlslug                ID  \
0             /Spider-Man_(Peter_Parker)   Secret Identity
1       /Captain_America_(Steven_Rogers)   Public Identity
2  /Wolverine_(James_%22Logan%22_Howlett)   Public Identity
3    /Iron_Man_(Anthony_%22Tony%22_Stark)   Public Identity
4                   /Thor_(Thor_Odinson)  No Dual Identity

                ALIGN         EYE        HAIR              SEX       GSM  \
0     Good Characters  Hazel Eyes  Brown Hair  Male Characters  dontknow
1     Good Characters   Blue Eyes  White Hair  Male Characters       NaN
2  Neutral Characters   Blue Eyes  Black Hair  Male Characters  dontknow
3     Good Characters   Blue Eyes  Black Hair  Male Characters       NaN
4     Good Characters   Blue Eyes  Blond Hair  Male Characters       NaN

               ALIVE  APPEARANCES FIRST APPEARANCE  Year
0  Living Characters       4043.0           Aug-62  1962
1  Living Characters       3360.0           Mar-41  1941
2  Living Characters       3061.0           Oct-74  1974
3  Living Characters       2961.0           Mar-63  1963
4  Living Characters       2258.0           Nov-50  1950
```

Otro detalle que se debe considerar es que se utiliza tanto el valor `NaN` como la cadena de caracteres `'dontknow'` para indicar valores desconocidos o perdidos. Podemos indicar que hay que interpretar esta cadena como valor perdido con el método `na_values`:

[58]:
```
data = pd.read_csv("data/marvel-wikia-data.csv", thousands=".", skiprows=[31],
                   escapechar="\\", na_values=["dontknow"])
data.head(n=5)
```

[58]:
```
   page_id                                  name  \
0     1678            Spider-Man (Peter Parker)
1     7139      Captain America (Steven Rogers)
2    64786  Wolverine (James "Logan"" Howlett)"
3     1868    Iron Man (Anthony "Tony"" Stark)"
4     2460                  Thor (Thor Odinson)

                                urlslug                ID  \
0             /Spider-Man_(Peter_Parker)   Secret Identity
1       /Captain_America_(Steven_Rogers)   Public Identity
2  /Wolverine_(James_%22Logan%22_Howlett)   Public Identity
3    /Iron_Man_(Anthony_%22Tony%22_Stark)   Public Identity
4                   /Thor_(Thor_Odinson)  No Dual Identity

                ALIGN         EYE        HAIR              SEX  GSM  \
0     Good Characters  Hazel Eyes  Brown Hair  Male Characters  NaN
1     Good Characters   Blue Eyes  White Hair  Male Characters  NaN
2  Neutral Characters   Blue Eyes  Black Hair  Male Characters  NaN
3     Good Characters   Blue Eyes  Black Hair  Male Characters  NaN
4     Good Characters   Blue Eyes  Blond Hair  Male Characters  NaN

                ALIVE  APPEARANCES FIRST APPEARANCE  Year
0  Living Characters       4043.0           Aug-62  1962
1  Living Characters       3360.0           Mar-41  1941
2  Living Characters       3061.0           Oct-74  1974
3  Living Characters       2961.0           Mar-63  1963
4  Living Characters       2258.0           Nov-50  1950
```

Finalmente, otra de las funcionalidades bastante potentes de la función `read_csv` es la de aplicar alguna función a los elementos de cada columna antes de incorporarlo al *dataframe*. Así, por ejemplo, la columna `FIRST APPEARANCE` contiene el mes (abreviado), un guion y el año en formato de dos dígitos. Si quisiéramos que esta columna contuviera únicamente al año y en formato de 4 dígitos, podríamos crear una función anónima que hiciera la conversión, y pasar esta función a `read_csv` con el parámetro`converters`:

[59]:
```
data = pd.read_csv(
    "data/marvel-wikia-data.csv",
    thousands=".", skiprows=[31], escapechar="\\", na_values=["dontknow"],
    converters={"FIRST APPEARANCE": lambda x: int(x.split("-")[1])+1900}
)
data[["FIRST APPEARANCE"]].head()
```

[59]:
```
   FIRST APPEARANCE
0              1962
1              1941
2              1974
3              1963
```

```
4                1950
```

Más allá de los ficheros csv, hay otros formatos que también se utilizan a menudo para intercambiar o guardar datos. Pandas dispone de varias funciones para cargar datos provenientes de los formatos de datos más populares, tales como json (`read_json`) o excel (`read_excel`).

5 4. Serialización de datos

La **serialización** de datos es el proceso de convertir datos estructurados (por ejemplo, una lista o un diccionario) en algún formato que permita almacenarlos o transmitirlos, de manera que sea posible después, a partir de un proceso de deserialización, recuperar los datos con su estructura original.

La serialización puede ser útil, por ejemplo, para transmitir un conjunto de datos a través de la red, para almacenar datos que serán tratados desde Python, o como método para crear *checkpoints* que permitan recuperar el estado de nuestros *scripts* en códigos que tardan mucho tiempo en finalizar.

El método principal de serialización en Python se encuentra implementado por el módulo `pickle` de la librería estándar.

5.1 4.1. Serialización de datos con pickle

Vamos a ver un ejemplo sencillo de serialización y deserialización de una estructura de datos simple, un diccionario:

[60]:
```
import pickle

# Creamos un diccionario
a_dict = {"Spain": 34, "United Kingdom": 44}

# Serializamos el diccionario y mostramos el resultado de la serialización
serialized_dict = pickle.dumps(a_dict)
print(serialized_dict)
```

```
b'\x80\x04\x95"\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x05Spain\x94K"\x8c\x0eUnit
ed Kingdom\x94K,u.'
```

[61]:
```
# Deserializamos el diccionario, creando un nuevo diccionario a partir del
# contenido serializado
des_dict = pickle.loads(serialized_dict)

# Comprobamos si el contenido del diccionario original y del deserializado son
# iguales
des_dict == a_dict
```

[61]:
```
True
```

Después de ver un ejemplo de serialización de un diccionario, veremos otro que se acerca más al uso real de `pickle` en la minería de datos. Supongamos que hemos entrenado un modelo de aprendizaje

automático a partir de unos datos, y que ahora lo queremos guardar para usarlo posteriormente para hacer predicciones. En este caso, puede resultar útil almacenar el modelo aprendido utilizando pickle, y recuperarlo después cuando sea necesario hacer predicciones. Veamos un ejemplo con la creación de un árbol de decisión para el conjunto de datos de flores de Iris:

[62]:
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
import pandas as pd

# Cargamos los datos
data = datasets.load_iris()
iris_df = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Creamos los conjuntos de test y aprendizaje
x_train, x_test, y_train, y_test = train_test_split(iris_df, y, test_size=0.1)

# Entrenamos un árbol de decisión
dec_tree = DecisionTreeClassifier(splitter="random", criterion="entropy")
dec_tree.fit(x_train, y_train)

# Probamos el clasificador con los datos de test
y_test_predicted = dec_tree.predict(x_test)
print("Predicted classes: \t" + str(y_test_predicted))
print("Accuracy: \t\t" + str(dec_tree.score(x_test, y_test)))

# Guardamos el modelo aprendido serializado en un fichero
p = "files_folder/dec_tree_model.pickle"
with open(p, "wb") as f:
    pickle.dump(dec_tree, f)
```

```
Predicted classes:      [2 1 1 0 0 1 0 2 2 1 2 1 2 1 0]
Accuracy:               0.9333333333333333
```

[63]:
```
# Eliminamos la variable dec_tree (para simular el hecho de terminar la
# ejecución de nuestro script de entrenamiento o bien la transmisión del
#  archivo a otra máquina)
del dec_tree
```

[64]:
```
# Cargamos el modelo guardado
with open(p, "rb") as f:
    dec_tree_des = pickle.load(f)

# Comprobamos que el modelo guardado se ha recuperado correctamente
y_test_predicted_des = dec_tree_des.predict(x_test)
print("Predicted classes: \t" + str(y_test_predicted_des))
print("Accuracy: \t\t" + str(dec_tree_des.score(x_test, y_test)))
```

```
Predicted classes:      [2 1 1 0 0 1 0 2 2 1 2 1 2 1 0]
Accuracy:               0.9333333333333333
```

Hay dos detalles importantes que hay que notar en el código anterior. Por un lado, tened en cuenta que para leer o escribir el resultado de una deserialización o serialización, utilizamos `load` y `dump` (respectivamente) en vez de `loads` y `dumps` (a diferencia del ejemplo del diccionario). Las funciones `load` y `dump` permiten leer y escribir el resultado de una serialización o deserialización en un fichero, mientras que `loads` y `dumps` devuelven una cadena de bytes. Por otra parte, notad que es necesario abrir los archivos en modo binario (`rb` para leer o `wb` para escribir).

5.2 4.2. Consideraciones sobre la serialización de datos

¡Es importante ser conscientes de las situaciones en las que es útil utilizar serialización con `pickle`! Hay que tener en cuenta que `pickle` solo sirve para transferir datos entre programas hechos con Python, que puede haber incompatibilidades entre versiones, y que no es un método seguro de transferencia de datos. En este sentido, tenemos que asegurar que solo deserializamos datos de confianza, ya que la deserialización puede provocar ejecución de código indeseado.

En cuanto a qué tipos de datos podemos serializar con Python, en general encontraremos que es inmediato serializar tipos básicos (enteros, flotantes, cadenas de caracteres, booleanos, etc.), tipos compuestos de tipo serializables (diccionarios, tuplas, listas, conjuntos, etc.) y clases y funciones definidas en el ámbito global.

Teniendo en cuenta las características que hemos descrito, ¿cuándo será pues una buena alternativa utilizar serialización con pickle en vez de utilizar un formato de datos estándar tales como JSON, XML o CSV? La siguiente tabla recoge la alternativa que se debe utilizar en función de las propiedades de la situación:

Propiedad | Pickle | JSON / XML / CSV / Otros |
—- |: —-: |: —-: |
Intercambio de datos sin confianza entre las partes |. | x |
Compatibilidad con otros lenguajes de programación (diferentes de Python) u otros programas |. | x |
Almacenamiento a largo plazo (posibles cambios de versión) |. | x |
Ejecución en diferentes máquinas, que tienen diferentes versiones de las librerías que proporcionan los tipos de datos almacenados |. | x |
Necesidad de lectura por parte de humanos |. | x |
Necesidad de almacenar objetos de manera rápida (por el programador), sin tener que decidir cómo representar el objeto | x |. |
Almacenamiento temporal (por ejemplo, *checkpoints* en la ejecución de un *script*) | x |. |
Necesidad de guardar el estado de nuestro programa (en vez de conjuntos de datos con cierta homogeneidad) | x |. |
Necesidad de almacenar funciones | x |. |

6 5. Interacción con bases de datos

El PEP249 describe la especificación de la API para acceder a bases de datos desde Python. La mayoría de los sistemas gestores de bases de datos (SGBD) más populares siguen esta especificación, por lo que hay bastante similitud en cómo se accede a los diferentes motores de bases de datos desde Python.

Así pues, el procedimiento general para interactuar con una base de datos desde Python consta de los siguientes pasos: 1. Importar el módulo que provee de la interfaz con la base de datos (específico para cada SGBD). 2. Crear una conexión con la base de datos, proveyendo datos sobre la localización de la base de datos y, en su caso, la autenticación. 3. Obtener un cursor sobre la conexión creada en el paso anterior. 4. Ejecutar las sentencias SQL deseadas. 5. Si las sentencias eran de consulta, recuperar los datos devueltos por la base de datos. 6. Cerrar la conexión.

Para ejemplificar el proceso, vamos a crear una nueva base de datos SQLite, donde guardaremos los datos de los personajes de Marvel (que cargaremos previamente del csv como hemos hecho anteriormente) y haremos consultas sobre estos datos utilizando SQL.

[65]:
```
# Cargamos los datos del csv de Marvel en un dataframe de pandas
data = pd.read_csv(
    "data/marvel-wikia-data.csv", decimal=",", escapechar="\\",
    na_values=["dontknow"],
    converters={"FIRST APPEARANCE": lambda x: int(x.split("-")[1])+1900},
    skiprows=[31])

# Obtenemos una cadena con la lista de columnas, que utilizaremos para crear
# la tabla de la base de datos
cols = ", ".join(data.columns)
print(cols)
```

```
page_id, name, urlslug, ID, ALIGN, EYE, HAIR, SEX, GSM, ALIVE, APPEARANCES,
FIRST APPEARANCE, Year
```

[66]:
```
# Importamos SQLite3
import sqlite3

# Creamos una conexión con la base de datos marvel.db
conn = sqlite3.connect('files_folder/marvel.db')

# Obtenemos un cursor sobre la conexión
cur = conn.cursor()

# Creamos una tabla utilizando los nombres de las columnas del dataframe
sql_create_stm = "CREATE TABLE marvel_chars ({})".format(cols)
print(sql_create_stm)

# Ejecutamos la sentencia SQL de creación de la tabla
cur.execute(sql_create_stm)

# Guardamos los cambios (hacemos un commit en la db)
conn.commit()
```

```
CREATE TABLE marvel_chars (page_id, name, urlslug, ID, ALIGN, EYE, HAIR, SEX,
GSM, ALIVE, APPEARANCES, FIRST APPEARANCE, Year)
```

Para crear la tabla de la base de datos, hemos utilizado una sentencia SQL de creación de tablas

(`CREATE TABLE`).

Una vez que hemos creado la tabla, podemos consultar su contenido ejecutando un `SELECT`:

[67]:
```
# Ejecutamos SELECT * sobre toda la tabla
sql_sel_stm = "SELECT * FROM marvel_chars"
cur.execute(sql_sel_stm)

# Recuperamos los resultados
results = cur.fetchall()

# Mostramos los resultados: la tabla de la base de datos está vacía
print(results)
```

```
[]
```

Como acabamos de crear la tabla, esta está vacía, y el `SELECT` no devuelve ninguna fila. Ahora, insertaremos los datos del *dataframe* en la base de datos, iterando por cada fila del *dataframe* e insertando cada fila en la tabla con un `INSERT`:

[68]:
```
# Iteramos por cada fila de la tabla
for index, row in data.iterrows():
    # Insertamos los datos en la tabla marvel_chars
    cur.execute("INSERT INTO marvel_chars VALUES "
                "(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)", row)

# Hacemos un commit de los cambios
conn.commit()
```

Una vez insertadas los datos en la base de datos, podemos repetir el `SELECT *` que hemos realizado anteriormente, y recuperar así todos los datos que hemos insertado en la tabla:

[69]:
```
# Ejecutamos SELECT * sobre toda la tabla
sql_sel_stm = "SELECT * FROM marvel_chars"
cur.execute(sql_sel_stm)

# Recuperamos los resultados
results = cur.fetchall()

# Mostramos los resultados: la tabla de la base de datos contiene los datos
# de los personajes de Marvel
print(results)
```

```
[(1678, 'Spider-Man (Peter Parker)', '/Spider-Man_(Peter_Parker)', 'Secret
Identity', 'Good Characters', 'Hazel Eyes', 'Brown Hair', 'Male Characters',
None, 'Living Characters', 4043.0, 1962, '1.962'), (7139, 'Captain America
(Steven Rogers)', '/Captain_America_(Steven_Rogers)', 'Public Identity', 'Good
Characters', 'Blue Eyes', 'White Hair', 'Male Characters', None, 'Living
Characters', 3360.0, 1941, '1.941'), (64786, 'Wolverine (James "Logan""
Howlett)"', '/Wolverine_(James_%22Logan%22_Howlett)', 'Public Identity',
```

'Neutral Characters', 'Blue Eyes', 'Black Hair', 'Male Characters', None,
'Living Characters', 3061.0, 1974, '1.974'), (1868, 'Iron Man (Anthony "Tony""
Stark)"', '/Iron_Man_(Anthony_%22Tony%22_Stark)', 'Public Identity', 'Good
Characters', 'Blue Eyes', 'Black Hair', 'Male Characters', None, 'Living
Characters', 2961.0, 1963, '1.963'), (2460, 'Thor (Thor Odinson)',
'/Thor_(Thor_Odinson)', 'No Dual Identity', 'Good Characters', 'Blue Eyes',
'Blond Hair', 'Male Characters', None, 'Living Characters', 2258.0, 1950,
'1.950'), (2458, 'Benjamin Grimm (Earth-616)', '/Benjamin_Grimm_(Earth-616)',
'Public Identity', 'Good Characters', 'Blue Eyes', 'No Hair', 'Male Characters',
None, 'Living Characters', 2255.0, 1961, '1.961'), (2166, 'Reed Richards
(Earth-616)', '/Reed_Richards_(Earth-616)', 'Public Identity', 'Good
Characters', 'Brown Eyes', 'Brown Hair', 'Male Characters', None, 'Living
Characters', 2072.0, 1961, '1.961'), (1833, 'Hulk (Robert Bruce Banner)',
'/Hulk_(Robert_Bruce_Banner)', 'Public Identity', 'Good Characters', 'Brown
Eyes', 'Brown Hair', 'Male Characters', None, 'Living Characters', 2017.0, 1962,
'1.962'), (29481, 'Scott Summers (Earth-616)', '/Scott_Summers_(Earth-616)',
'Public Identity', 'Neutral Characters', 'Brown Eyes', 'Brown Hair', 'Male
Characters', None, 'Living Characters', 1955.0, 1963, '1.963'), (1837, 'Jonathan
Storm (Earth-616)', '/Jonathan_Storm_(Earth-616)', 'Public Identity', 'Good
Characters', 'Blue Eyes', 'Blond Hair', 'Male Characters', None, 'Living
Characters', 1934.0, 1961, '1.961'), (15725, 'Henry McCoy (Earth-616)',
'/Henry_McCoy_(Earth-616)', 'Public Identity', 'Good Characters', 'Blue Eyes',
'Blue Hair', 'Male Characters', None, 'Living Characters', 1825.0, 1963,
'1.963'), (1863, 'Susan Storm (Earth-616)', '/Susan_Storm_(Earth-616)', 'Public
Identity', 'Good Characters', 'Blue Eyes', 'Blond Hair', 'Female Characters',
None, 'Living Characters', 1713.0, 1961, '1.961'), (7823, 'Namor McKenzie
(Earth-616)', '/Namor_McKenzie_(Earth-616)', 'No Dual Identity', 'Neutral
Characters', 'Green Eyes', 'Black Hair', 'Male Characters', None, 'Living
Characters', None, 1961, '1.528'), (2614, 'Ororo Munroe (Earth-616)',
'/Ororo_Munroe_(Earth-616)', 'Public Identity', 'Good Characters', 'Blue Eyes',
'White Hair', 'Female Characters', None, 'Living Characters', 1512.0, 1975,
'1.975'), (1803, 'Clinton Barton (Earth-616)', '/Clinton_Barton_(Earth-616)',
'Public Identity', 'Good Characters', 'Blue Eyes', 'Blond Hair', 'Male
Characters', None, 'Living Characters', 1394.0, 1964, '1.964'), (1396, 'Matthew
Murdock (Earth-616)', '/Matthew_Murdock_(Earth-616)', 'Public Identity', 'Good
Characters', 'Blue Eyes', 'Red Hair', 'Male Characters', None, 'Living
Characters', 1338.0, 1964, '1.964'), (55534, 'Stephen Strange (Earth-616)',
'/Stephen_Strange_(Earth-616)', 'Public Identity', 'Good Characters', 'Grey
Eyes', 'Black Hair', 'Male Characters', None, 'Living Characters', 1307.0, 1963,
'1.963'), (1978, 'Mary Jane Watson (Earth-616)',
'/Mary_Jane_Watson_(Earth-616)', 'No Dual Identity', 'Good Characters', 'Green
Eyes', 'Red Hair', 'Female Characters', None, 'Living Characters', 1304.0, 1965,
'1.965'), (1872, 'John Jonah Jameson (Earth-616)',
'/John_Jonah_Jameson_(Earth-616)', 'No Dual Identity', 'Neutral Characters',
'Blue Eyes', 'Black Hair', 'Male Characters', None, 'Living Characters', 1266.0,
1963, '1.963'), (35350, 'Robert Drake (Earth-616)', '/Robert_Drake_(Earth-616)',
'Secret Identity', 'Good Characters', 'Brown Eyes', 'Brown Hair', 'Male
Characters', None, 'Living Characters', 1265.0, 1963, '1.963'), (1557, 'Henry

```
Pym (Earth-616)', '/Henry_Pym_(Earth-616)', 'Public Identity', 'Good
Characters', 'Blue Eyes', 'Blond Hair', 'Male Characters', None, 'Living
Characters', 1237.0, 1962, '1.962'), (65255, 'Charles Xavier (Earth-616)',
'/Charles_Xavier_(Earth-616)', 'Public Identity', 'Good Characters', 'Blue
Eyes', 'Bald', 'Male Characters', None, 'Deceased Characters', 1233.0, 1963,
'1.963'), (1073, 'Warren Worthington III (Earth-616)',
'/Warren_Worthington_III_(Earth-616)', 'Public Identity', 'Good Characters',
'Blue Eyes', 'Blond Hair', 'Male Characters', None, 'Living Characters', 1230.0,
1963, '1.963'), (1346, 'Piotr Rasputin (Earth-616)',
'/Piotr_Rasputin_(Earth-616)', 'Secret Identity', 'Good Characters', 'Blue
Eyes', 'Black Hair', 'Male Characters', None, 'Living Characters', 1162.0, 1975,
'1.975'), (2512, 'Wanda Maximoff (Earth-616)', '/Wanda_Maximoff_(Earth-616)',
'Public Identity', 'Good Characters', 'Green Eyes', 'Brown Hair', 'Female
Characters', None, 'Living Characters', 1161.0, 1964, '1.964'), (1671, 'Nicholas
Fury (Earth-616)', '/Nicholas_Fury_(Earth-616)', 'No Dual Identity', 'Neutral
Characters', 'Brown Eyes', 'Brown Hair', 'Male Characters', None, 'Living
Characters', 1137.0, 1963, '1.963'), (1976, 'Janet van Dyne (Earth-616)',
'/Janet_van_Dyne_(Earth-616)', 'Public Identity', 'Good Characters', 'Blue
Eyes', 'Auburn Hair', 'Female Characters', None, 'Living Characters', 1120.0,
1963, '1.963'), (65219, 'Jean Grey (Earth-616)', '/Jean_Grey_(Earth-616)',
'Public Identity', 'Good Characters', 'Green Eyes', 'Red Hair', 'Female
Characters', None, 'Deceased Characters', 1107.0, 1963, '1.963'), (6545,
'Natalia Romanova (Earth-616)', '/Natalia_Romanova_(Earth-616)', 'Public
Identity', 'Good Characters', 'Green Eyes', 'Red Hair', 'Female Characters',
'Bisexual Characters', 'Living Characters', 1050.0, 1964, '1.964'), (2223, 'Kurt
Wagner (Earth-616)', '/Kurt_Wagner_(Earth-616)', 'Secret Identity', 'Good
Characters', 'Yellow Eyes', 'Blue Hair', 'Male Characters', None, 'Living
Characters', 1047.0, 1975, '1.975')]
```

Podemos ejecutar consultas SQL sobre la tabla que acabamos de crear. Así, por ejemplo, si queremos recuperar solo el nombre de los personajes femeninos, podríamos hacer:

[70]:
```
# Ejecutamos la consulta SQL
sql_sel_stm = "SELECT name FROM marvel_chars WHERE sex = 'Female Characters'"
cur.execute(sql_sel_stm)

# Mostramos los resultados
results = cur.fetchall()
print(results)
```

```
[('Susan Storm (Earth-616)',), ('Ororo Munroe (Earth-616)',), ('Mary Jane Watson
(Earth-616)',), ('Wanda Maximoff (Earth-616)',), ('Janet van Dyne
(Earth-616)',), ('Jean Grey (Earth-616)',), ('Natalia Romanova (Earth-616)',)]
```

Una vez que hemos terminado de trabajar con la base de datos, es importante recordar cerrar el recurso:

[71]:
```
# Cerramos la conexión
conn.close()
```

Es importante notar que si hubiéramos querido hacer lo mismo con una base de datos MySQL en vez de SQLite, habría que cambiar la librería que se debería usar (el `import` del inicio del código), pero luego el código sería en general el mismo, y funcionaría haciendo únicamente pequeñas adaptaciones (por ejemplo, en la creación de la conexión).

7 6. Interacción con el sistema operativo

Hasta ahora hemos visto cómo interactuar con el sistema operativo a través de la lectura o escritura de archivos. En esta sección, veremos cómo podemos ejecutar programas externos o comandos en la consola del sistema desde nuestro código Python, utilizando el módulo `subprocess`.

El módulo `subprocess` permite ejecutar comandos con la función `run`. La función recibe como argumento el comando y devuelve un objeto `CompletedProcess`. Por defecto, no captura la salida del comando ejecutado, solo el código de retorno (tradicionalmente, se utiliza el 0 para indicar que el proceso ha finalizado con éxito). El siguiente ejemplo ejecuta un comando de listado, `ls`:

[72]:
```
import subprocess as sp

# Ejecutamos ls
exe_out = sp.run(["ls"])

# Mostramos el resultado que devuelve run
print(exe_out)
print("The args were: {}".format(exe_out.args))
print("The return code was: {}".format(exe_out.returncode))
```

```
CompletedProcess(args=['ls'], returncode=0)
The args were: ['ls']
The return code was: 0
```

El resultado es un objeto `CompletedProcess` que contiene tanto los argumentos como el código de retorno.

Podemos pasar argumentos adicionales añadiendo elementos a la lista que se envía como primer parámetro:

[73]:
```
# Ejecutamos ls -l
sp.run(["ls", "-l"])
```

[73]:
```
CompletedProcess(args=['ls', '-l'], returncode=0)
```

Con el fin de recuperar no solo el código de retorno sino también la salida de la ejecución del comando, usaremos el parámetro `stdout`, que permite indicar a dónde enviamos la salida:

[74]:
```
# Ejecutamos ls -l y capturamos la salida de la ejecución
exe_out = sp.run(['ls', '-l'], encoding='utf-8', stdout=sp.PIPE)

# Mostramos el resultado que devuelve run
print(exe_out)
```

```
print("\nThe args were: {}".format(exe_out.args))
print("\nThe return code was: {}".format(exe_out.returncode))
print("\nThe output was:\n{}".format(exe_out.stdout))
```

```
CompletedProcess(args=['ls', '-l'], returncode=0, stdout='total 352\n-rw-rw-r--
1 cperez cperez 162507 jul  1 16:26 3-CAT-
Fitxers_i_interacció_amb_el_sistema.ipynb\n-rw-rw-r-- 1 cperez cperez 161777 jul
1 16:24 3-ES-Ficheros_e_interacción_con_el_sistema.ipynb\ndrwxrwxr-x 2 cperez
cperez   4096 sep  8  2020 data\n-rw-rw-r-- 1 cperez cperez     17 jul  1 16:27
file_2.txt\ndrwxrwxr-x 4 cperez cperez   4096 jul  1 16:27 files_folder\n-rw-
rw-r-- 1 cperez cperez   5103 sep  8  2020 files_folder.zip\ndrwxrwxr-x 3 cperez
cperez   4096 jul  1 16:26 mem_data\ndrwxrwxr-x 2 cperez cperez   4096 sep  8
2020 pdf\n-rw-rw-r-- 1 cperez cperez    284 sep  8  2020 README.md\n')

The args were: ['ls', '-l']

The return code was: 0

The output was:
total 352
-rw-rw-r-- 1 cperez cperez 162507 jul  1 16:26 3-CAT-
Fitxers_i_interacció_amb_el_sistema.ipynb
-rw-rw-r-- 1 cperez cperez 161777 jul  1 16:24 3-ES-
Ficheros_e_interacción_con_el_sistema.ipynb
drwxrwxr-x 2 cperez cperez   4096 sep  8  2020 data
-rw-rw-r-- 1 cperez cperez     17 jul  1 16:27 file_2.txt
drwxrwxr-x 4 cperez cperez   4096 jul  1 16:27 files_folder
-rw-rw-r-- 1 cperez cperez   5103 sep  8  2020 files_folder.zip
drwxrwxr-x 3 cperez cperez   4096 jul  1 16:26 mem_data
drwxrwxr-x 2 cperez cperez   4096 sep  8  2020 pdf
-rw-rw-r-- 1 cperez cperez    284 sep  8  2020 README.md
```

Ahora, el objeto `CompletedProcess` contiene también un atributo `stdout` con el resultado de la ejecución del proceso, que, en este caso, consiste en el listado del directorio donde se ha ejecutado.

Podemos utilizar `run` también para ejecutar otros programas, que no sean comandos del sistema. Por ejemplo, en la carpeta `files_folder` hay un pequeño *script* de bash con el siguiente código:

```
#!/bin/bash

echo "Script executed!"
```

Es decir, el *script* simplemente muestra por pantalla el mensaje 'Script executed!' cada vez que es ejecutado.

Antes de ejecutar la celda siguiente, daremos permisos de ejecución al *script* y probaremos su funcionamiento:

1. En primer lugar, abrid una consola del sistema, situaos en la carpeta `files_folder` y ejecutad el siguiente comando: `chmod +x *.sh`

Esto dará permisos de ejecución al *script*, ya que por defecto los archivos no pueden ejecutarse (por cuestiones de seguridad).

2. Ahora, desde la misma consola, ejecutad el *script*: `./echo_script.sh`

Esto ejecutará el *script* y mostrará el mensaje 'Script executed!' en vuestro terminal.

Veamos pues cómo podemos ejecutar este mismo *script* desde nuestro programa Python con `run`:

[75]:
```
# Ejecutamos echo_script y capturamos la salida de la ejecución
exe_out = sp.run(["./files_folder/echo_script.sh"],
                 encoding='utf-8', stdout=sp.PIPE)
print(exe_out)
```

```
CompletedProcess(args=['./files_folder/echo_script.sh'], returncode=0,
stdout='Script executed!\n')
```

La función `run` también permite ejecutar un programa externo y pasarle datos de entrada. Por ejemplo, el *script* `echo_read_script.sh` tiene el siguiente código:

```
#!/bin/bash

echo "Enter a number"
read number
echo "Your number was $number"
```

Es decir, el *script* muestra primero el mensaje 'Enter a number', pide al usuario que entre un número y muestra por pantalla el mensaje 'Your number was' y el número introducido por el usuario.

Ejecutamos ahora el *script* `echo_read_script.sh`, pasándole como entrada el contenido del archivo`a_number.txt` (que contiene el número 42):

[76]:
```
# Ejecutamos echo_read_script, capturando la salida de la ejecución
# y pasando como entrada el contenido del archivo a_number
with open("files_folder/a_number.txt", "r") as f:
    exe_out = sp.run(["./files_folder/echo_read_script.sh"],
                     encoding='utf-8',
                     stdin=f,
                     stdout=sp.PIPE)
    # Mostramos el resultado que devuelve run
    print(exe_out)
    print("\nThe args were: {}".format(exe_out.args))
    print("The return code was: {}".format(exe_out.returncode))
    print("The output was:\n{}".format(exe_out.stdout))
```

```
CompletedProcess(args=['./files_folder/echo_read_script.sh'], returncode=0,
stdout='Enter a number\nYour number was 42\n')

The args were: ['./files_folder/echo_read_script.sh']
```

```
The return code was: 0
The output was:
Enter a number
Your number was 42
```

Ya para terminar, es importante notar que la función `run` ejecuta lo que le hayamos indicado y luego espera (por defecto de manera indefinida) a que finalice la ejecución del comando. Por lo tanto, una llamada a `run` puede bloquear nuestro programa si el programa que ejecutamos no acaba nunca. Así, por ejemplo, la ejecución del *script* `endless_script.sh` con el siguiente código que contiene un bucle infinito:

```
#!/bin/bash

while true
do
    sleep 1
done
```

haría que nuestro programa Python se quedara indefinidamente esperando. Para evitarlo, `run` tiene un parámetro`timeout`, que permite especificar el tiempo máximo (en segundos) que queremos esperar a la ejecución de un programa externo. Pasado este tiempo, si la ejecución no ha finalizado se genera una excepción.

[77]:
```python
# Ejecutamos ls con timeout (la ejecución finaliza sin
# generar excepción)
try:
    exe_out = sp.run(["ls"], timeout=3)
except sp.TimeoutExpired as e:
    print(e)
```

[78]:
```python
# Ejecutamos el script endless_script con timeout
# (la ejecución finaliza con un timeout a los 3 segundos)
try:
    exe_out = sp.run(["./files_folder/endless_script.sh"],
                     timeout=3)
except sp.TimeoutExpired as e:
    print(e)
```

```
Command '['./files_folder/endless_script.sh']' timed out after
2.9999396099992737 seconds
```

Hemos visto pues cómo la función `run` es bastante versátil, y nos permite ejecutar programas externos desde nuestro código Python.

8 7. Ejercicios para practicar

A continuación encontraréis un conjunto de problemas que os pueden servir para practicar los conceptos explicados en esta unidad. Os recomendamos que intentéis realizar estos problemas vosotros mismos y que, una vez realizados, comparéis la solución que proponemos con vuestra

solución. No dudéis en dirigir todas las dudas que surjan de la resolución de estos ejercicios, o de las soluciones propuestas, al foro del aula.

1. Cread un código que permita monitorizar el consumo de memoria RAM de la máquina en la que se ejecute. El código guardará los datos de la memoria total y utilizada del sistema durante un periodo de tiempo, capturando los datos en intervalos periódicos.

Estos datos se guardarán en archivos de texto, utilizando un fichero para los datos capturados en cada momento. Así, dentro de la carpeta de datos, habrá una carpeta para los datos de cada día (que tendrá por nombre el año, el mes y el día, escritos seguidos, por ejemplo, `20200318`). Dentro de la carpeta de cada día, habrá un archivo para cada instante de tiempo en el que se hayan obtenido datos (que tendrá por nombre la hora, el minuto y el segundo, separados por guiones bajos, por ejemplo, `14_45_55`). El contenido del archivo serán los dos valores (memoria total y utilizada) separados por comas (por ejemplo, `15571, 4242`).

Cread también el código que permita recuperar todos los datos almacenados y obtener una descripción estadística básica (media, mediana y desviación estándar).

Para ello, implementaremos una serie de funciones que se detallan a continuación.

1.1. Cread una función que reciba como parámetro el nombre de una carpeta (que será `mem_data` por defecto) y cree las carpetas necesarias para almacenar datos para el día actual. Es decir, el código deberá crear, si no existe ya, una carpeta de datos con el nombre que ha recibido como parámetro (o usar `mem_data` si no se ha especificado ningún nombre), y otra carpeta dentro de esta que tenga por nombre el día actual (en el formato año de 4 cifras, mes de 2 cifras, día de 2 cifras, seguidos sin separadores, por ejemplo, `20200318`).

[79]:
```
# Respuesta
```

1.2. Implementad una función que reciba como parámetro el *path* con la carpeta de la fecha actual (que se ha creado en el apartado anterior) y escriba un fichero con los datos de consumo de memoria del sistema actuales. El archivo debe tener por nombre la hora actual (en el formato `hora_minuto_segundo`, con los ítems separados por guiones bajos, por ejemplo,`14_45_55`). El contenido del archivo serán los dos valores (memoria total y utilizada) en megabytes separados por comas (por ejemplo, `15571, 4242`).

Para obtener los datos del consumo de memoria, recordad que podéis ejecutar comandos del sistema con el módulo `subprocess` (seguramente necesitaréis buscar información sobre cómo obtener estos datos con comandos de *unix*).

[80]:
```
# Respuesta
```

1.3. Implementad una función que reciba como parámetros el número de muestras que capturar y el intervalo de tiempo entre cada una de las muestras (en segundos), y que capture los datos del consumo de memoria tantas veces como se haya especificado, esperando el tiempo indicado entre capturas. La función hará uso de las dos funciones definidas anteriormente.

[81]:
```
# Respuesta
```

1.4. Llamad a la función definida en el apartado 1.3 y capturad 20 muestras de consumo de memoria, utilizando un intervalo de 3 segundos entre cada captura.

[82]:
```
# Respuesta
```

1.5. Implementad una función que lea todos los datos que se han capturado, almacenados en una carpeta que recibirá como parámetro (y que, de nuevo, tomará como valor por defecto `mem_data`), y que muestre los siguientes datos: * El número de muestras leídas. * La media de la memoria total y utilizada. * La mediana de la memoria total y utilizada. * La desviación estándar de la memoria total y utilizada. * La fecha y hora de la primera y última capturas de las que tenemos datos.

Llamad a la función anterior para obtener un resumen de los datos capturados.

[83]:
```
# Respuesta
```

1.6. Implementad una función que cree un archivo comprimido con todos los datos almacenados para cada día. La función recibirá como argumento el nombre de la carpeta de datos (por defecto, `mem_data`) y creará tantos ficheros comprimidos como días de los que disponemos datos. Cada archivo comprimido contendrá todos los archivos de datos de ese día.

Llamad a la función anterior y comprobad que se generan los ficheros comprimidos correctamente.

[84]:
```
# Respuesta
```

8.1 7.1. Soluciones a los ejercicios para practicar

1. Cread un código que permita monitorizar el consumo de memoria RAM de la máquina en la que se ejecute. El código guardará los datos de la memoria total y utilizada del sistema durante un periodo de tiempo, capturando los datos en intervalos periódicos.

Estos datos se guardarán en archivos de texto, utilizando un fichero para los datos capturados en cada momento. Así, dentro de la carpeta de datos, habrá una carpeta para los datos de cada día (que tendrá por nombre el año, el mes y el día, escritos seguidos, por ejemplo, `20200318`). Dentro de la carpeta de cada día, habrá un archivo para cada instante de tiempo en el que se hayan obtenido datos (que tendrá por nombre la hora, el minuto y el segundo, separados por guiones bajos, por ejemplo, `14_45_55`). El contenido del archivo serán los dos valores (memoria total y utilizada) separados por comas (por ejemplo, `15571, 4242`).

Cread también el código que permita recuperar todos los datos almacenados y obtener una descripción estadística básica (media, mediana y desviación estándar).

Para ello, implementaremos una serie de funciones que se detallan a continuación.

1.1. Cread una función que reciba como parámetro el nombre de una carpeta (que será `mem_data` por defecto) y cree las carpetas necesarias para almacenar datos para el día actual. Es decir, el código deberá crear, si no existe ya, una carpeta de datos con el nombre que ha recibido como parámetro (o usar `mem_data` si no se ha especificado ningún nombre), y otra carpeta dentro de esta que tenga por nombre el día actual (en el formato año de 4 cifras, mes de 2 cifras, día de 2 cifras, seguidos sin separadores, por ejemplo, `20200318`).

[85]:
```
import datetime
```

```
def create_folder(data_folder='mem_data'):
    today = datetime.datetime.now().strftime("%Y%m%d")
    full_path = os.path.join(data_folder, today, "")
    # Creamos tanto la carpeta data_folder como la subcarpeta con el día
    # y evitamos obtener errores si ya existen
    os.makedirs(full_path, exist_ok=True)
    return full_path
```

1.2. Implementad una función que reciba como parámetro el *path* con la carpeta de la fecha actual (que se ha creado en el apartado anterior) y escriba un fichero con los datos de consumo de memoria del sistema actuales. El archivo debe tener por nombre la hora actual (en el formato `hora_minuto_segundo`, con los ítems separados por guiones bajos, por ejemplo,`14_45_55`). El contenido del archivo serán los dos valores (memoria total y utilizada) en megabytes separados por comas (por ejemplo, `15571, 4242`).

Para obtener los datos del consumo de memoria, recordad que podéis ejecutar comandos del sistema con el módulo `subprocess` (seguramente necesitaréis buscar información sobre cómo obtener estos datos con comandos de *unix*).

[86]:
```
def save_current_data(folder_name):
    # Construimos el path del fichero a escribir
    file_name = datetime.datetime.now().strftime("%H_%M_%S")
    full_path = os.path.join(folder_name, file_name)
    with open(full_path, 'w') as f:
        # Recuperamos los datos de memoria ejecutando free -tm y quedándonos
        # únicamente con los datos de memoria RAM (línea 1), y formatamos
        # la salida como pide el enunciado (con los valores separados por
        # comas)
        exe_out = sp.run(['free', '-tm'], encoding='utf-8', stdout=sp.PIPE)
        text = ", ".join(exe_out.stdout.split("\n")[1].split()[1:3])
        # Escrivim el resultat al fitxer
        f.write(text)
```

1.3. Implementad una función que reciba como parámetros el número de muestras que capturar y el intervalo de tiempo entre cada una de las muestras (en segundos), y que capture los datos del consumo de memoria tantas veces como se haya especificado, esperando el tiempo indicado entre capturas. La función hará uso de las dos funciones definidas anteriormente.

[87]:
```
import time


def store_data(num_samples, interval):
    full_path = create_folder()
    for _ in range(num_samples):
        save_current_data(full_path)
        time.sleep(interval)
```

1.4. Llamad a la función definida en el apartado 1.3 y capturad 20 muestras de consumo de

memoria, utilizando un intervalo de 3 segundos entre cada captura.

[88]:
```
store_data(num_samples=20, interval=3)
```

1.5. Implementad una función que lea todos los datos que se han capturado, almacenados en una carpeta que recibirá como parámetro (y que, de nuevo, tomará como valor por defecto `mem_data`), y que muestre los siguientes datos: * El número de muestras leídas. * La media de la memoria total y utilizada. * La mediana de la memoria total y utilizada. * La desviación estándar de la memoria total y utilizada. * La fecha y hora de la primera y última capturas de las que tenemos datos.

Llamad a la función anterior para obtener un resumen de los datos capturados.

[89]:
```
import numpy as np


def read_data(data_folder='mem_data'):
    # Recuperamos todos los archivos de datos
    all_files = glob.glob(data_folder + '/*/*', recursive=True)
    # Cargamos los datos de todos los archivos en listas
    # (usamos una lista para cada tipo de datos que guardar)
    total_mem, used_mem, dts = [], [], []
    for file_name in all_files:
        with open(file_name) as f:
            # Leemos la línea del fichero
            line = f.readline()
            # Separamos los dos valores (memoria total y memoria utilizada)
            t_mem, u_mem = line.split(",")
            # Añadimos los datos a las listas
            total_mem.append(int(t_mem))
            used_mem.append(int(u_mem))
            # Añadimos también la fecha y hora del fichero a la lista dts
            dt = datetime.datetime.strptime(file_name[-17:], "%Y%m%d/%H_%M_%S")
            dts.append(dt)

    # Mostramos el resumen de los datos leídos
    print("{} samples read\n".format(len(total_mem)))
    print("Total memory:\t\t{} (avg), {} (median), {} std".format(
        np.mean(total_mem), np.median(total_mem), np.std(total_mem)))
    print("Used memory:\t\t{} (avg), {} (median), {} std".format(
        np.mean(used_mem), np.median(used_mem), np.std(used_mem)))
    print("First sample is from:\t{}".format(min(dts)))
    print("Last sample is from:\t{}".format(max(dts)))


read_data()
```

```
60 samples read
```

```
Total memory:           7495.0 (avg), 7495.0 (median), 0.0 std
Used memory:            5175.0 (avg), 5185.0 (median), 35.44808410431608 std
First sample is from:   2021-07-01 16:22:48
Last sample is from:    2021-07-01 16:28:18
```

1.6. Implementad una función que cree un archivo comprimido con todos los datos almacenados para cada día. La función recibirá como argumento el nombre de la carpeta de datos (por defecto, `mem_data`) y creará tantos ficheros comprimidos como días de los que disponemos datos. Cada archivo comprimido contendrá todos los archivos de datos de ese día.

Llamad a la función anterior y comprobad que se generan los ficheros comprimidos correctamente.

[90]:
```python
def compress_data(data_folder='mem_data'):

    date_folders = glob.glob(data_folder + '/*')
    for date_folder in date_folders:
        # Creamos el fichero .zip
        zip_file = os.path.join(date_folder + '.zip')
        # Añadimos al fichero .zip todos los ficheros que hay en la carpeta
        # date_folder que estamos procesando
        with zf.ZipFile(zip_file, 'w', compression=zf.ZIP_DEFLATED) as zip_f:
            zip_f.write(date_folder)
            sample_files = glob.glob(date_folder + '/*')
            for sample_file in sample_files:
                zip_f.write(sample_file)


compress_data()
```

9 8. Bibliografía

9.1 8.1. Bibliografía básica

La codificación es uno de los detalles importantes que se debe considerar cuando hay que leer y/o escribir un archivo y, a menudo, es el origen de dolores de cabeza en muchos programadores (sobre todo en lenguajes de más bajo nivel que Python). Para entender qué es la codificación de caracteres, conocer cuáles son las codificaciones de caracteres más habituales y saber cómo gestiona Python 3 la codificación, leed ahora la guía de este enlace.

9.2 8.2. Bibliografía adicional (ampliación de conocimientos)

Esta unidad presenta una introducción a cómo interactuar con el sistema de archivos y, en general, con el sistema operativo, desde Python. Así, como introducción, presenta algunas cuestiones de manera inicial y abre la puerta a explorarlas con más detalle. A continuación se listan algunos enlaces que os servirán para seguir explorando algunos de los temas que trabajamos en la unidad, ya sean puramente de programación en Python como del sistema operativo:

- **El sistema de ficheros de Linux**: en la unidad hablamos de interactuar con el sistema de archivos de Linux, pero no entramos a explicar cómo es este sistema de ficheros. Si deseáis

leer una introducción a este sistema, este *Overview* os puede resultar muy útil.

- **Permisos sobre los ficheros en unix**: si tenéis curiosidad por saber cómo funcionan los bits de permiso de los ficheros en unix, os recomendamos leer las tres partes de la serie de artículos sobre los permisos (1, 2 y 3).
- **Apertura de archivos desde Python**: la función `open` acepta otros argumentos opcionales que no hemos presentado, y que gestionan detalles como el *buffering* de datos, la codificación, la gestión de los errores, la gestión del salto de línea, etc. El lector interesado puede consultar la documentación oficial de la función `open` para descubrir cómo funcionan estos argumentos y qué opciones se encuentran disponibles.
- **Compresión de archivos**: existen otros formatos de compresión de datos aparte de los que hemos visto en esta unidad. El lector interesado puede leer la documentación del módulo `gzip` para conocer las funciones que permiten trabajar con archivos gzip desde Python.
- **Lectura de ficheros con pandas**: más allá de los ficheros csv, hay otros formatos que también se utilizan a menudo para intercambiar o guardar datos. Pandas dispone de varias funciones para cargar datos provenientes de los formatos de datos más populares, tales como json (`read_json`) o excel (`read_excel`).

También os recomendamos revisar la documentación oficial de las funciones y clases descritas en esta unidad, que encontraréis enlazadas en cada uno de los apartados que las describen, con el fin de conocer qué parámetros permiten ajustar su funcionamiento.