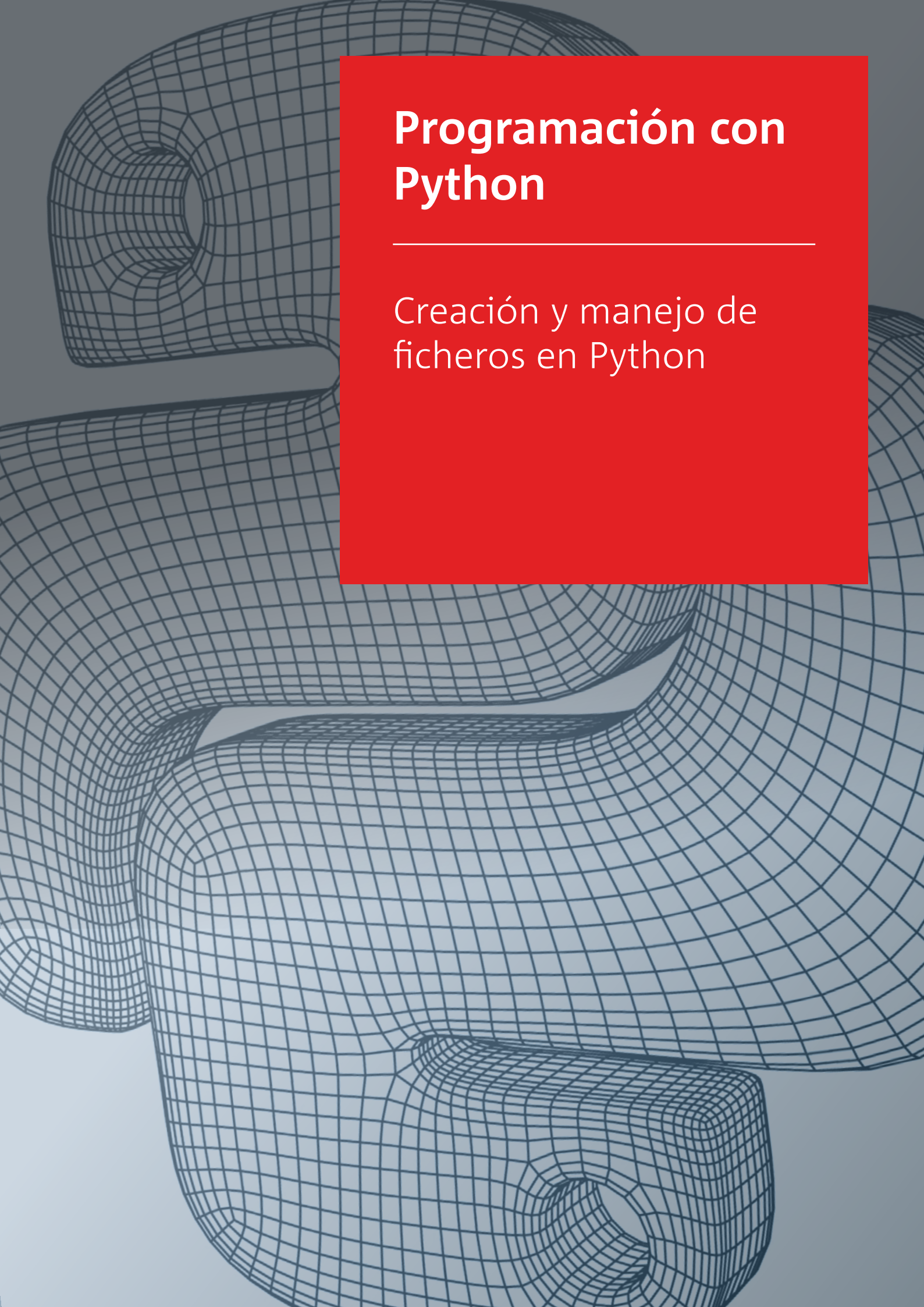

Programación con Python

Creación y manejo de ficheros en Python

INICIATIVA Y COORDINACIÓN
DEUSTO FORMACIÓN

COLABORADORES
Realización:
E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:
Claudio García Martorell
Licenciado IT Telecomunicaciones especialidad Telemática.
Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.
Concejal de Innovación y Tecnología.
Ponente y docente en distintas universidades y eventos.
Josep Estarlich Pau
Técnico de Ingeniería Informática.
Director Área de Software de la empresa Dismuntel.
Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:
Grufium educación y excelencia

Coordinación editorial:
Grufium educación y excelencia

Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)
ISBN: 978-84-1300-694-9 (Creación y manejo de ficheros en Python)

Depósito Legal: B 11038-2021

Impreso por:
SERVINFORM
Avenida de los Premios Nobel, 37
Polígono Casablanca
28850 Torrejón de Ardoz (Madrid)

Printed in Spain
Impreso en España

Esquema de contenido

Introducción

En este módulo aprenderemos a crear, modificar, abrir y cerrar un documento desde Python, así como a controlar desde nuestros programas la posible edición de archivos.

Entraremos en los distintos modelos de apertura y manipulación de ficheros y estudiaremos los métodos y las propiedades del objeto *file*.

JSON es un formato de texto para el intercambio de datos y vamos a aprender a recuperar de un fichero con contenido en JSON para poder construir nuestro objeto deseado y también a la inversa, mediante la serialización de objetos en ficheros, ya sea en JSON o no.

1. Modos de apertura de ficheros

Es esta unidad, estudiaremos qué operaciones se pueden realizar en Python con uno de los elementos básicos de la programación, los ficheros. Pondremos especial atención en qué modos de apertura se pueden llevar a cabo y qué acciones permite cada uno de ellos. Veremos también cuál es el indicador de cada modo de apertura, qué errores pueden producirse y en qué posición aparecerá el puntero en cada caso.

1.1. Operaciones con ficheros

Un **fichero** o **archivo informático** es un conjunto de bytes que se almacenan en un disco duro, organizados en carpetas o directorios. Se los puede identificar por nombre, contenido y extensión.

La denominación de "fichero" se debe a que son el equivalente digital de los archivos y las libretas escritas a mano del entorno tradicional.

Las operaciones que se pueden realizar son:

- **Creación de un fichero:** en esta operación el usuario puede crear un documento, indicando las propiedades, las características, el nombre, el formato y almacenando un contenido para poder abrirlo y trabajar posteriormente.

- **Apertura de un fichero:** en esta operación se busca un archivo que ha solicitado el usuario y le aparece listo para trabajar con él. En algunos sistemas operativos si el fichero no existe, en la misma acción de apertura se realiza la operación de creación.

  - Posibles errores

    - No se encuentra el fichero.

    - No tiene permiso para el acceso.

    - No se puede leer por errores del dispositivo.

- **Cierre de un fichero:** en esta operación indicamos que vamos a dejar de trabajar con el fichero y dejamos de tener el enlace para trabajar con

Certificación

Es importante interiorizar las operaciones con ficheros, pues en numerosas preguntas de la certificación se hace referencia a estas.

este. Al realizar esta operación, el sistema guarda toda la información almacenada en el búfer asociado a dicho fichero y lleva a cabo todas las acciones de limpieza para liberarlo, dejando de ser accesible.

- Posibles errores:
  - El fichero no está abierto.
  - No se puede escribir toda la información por un error en el disco duro (falta de espacio o problema físico).

- **Lectura de un fichero:** en esta operación realizamos la lectura del contenido, ya sea en binario o en texto plano.

- **Edición de un fichero:** en esta operación se modifica el contenido del fichero o se reemplaza por uno nuevo.
  - Posible error:
    - Que no tenga suficiente espacio para almacenar el nuevo contenido.

Antes de empezar con la gestión de ficheros, vamos a aprender los distintos modos de apertura de que disponemos. Además de aprovechar los recursos del *hardware* al ejecutar nuestro código fuente, por ejemplo, para no abrir un fichero en modo lectura y escritura y solamente realizar su lectura sin modificarlo.

1.1.1. El puntero en un fichero

Como su nombre indica, señala la posición desde donde se realizará cualquier operación en un fichero abierto. Es el equivalente a nuestro dedo cuando vamos marcando la lectura de un texto.

Cuando abrimos un fichero con la función “open()”, se creará un puntero, que se posicionará automáticamente dentro del fichero, en un lugar determinado por el modo utilizado (al inicio o al final). Este puntero se puede mover dentro del fichero para realizar lecturas o para escribir.

1.1.2. Modos de apertura y posición del puntero

A continuación, veremos el posicionamiento del puntero según los diferentes modos de apertura:

- **Modo de solo lectura**

  - Utilizará el indicador “r”.

  - El puntero se posicionará al inicio del fichero.

  - En el modo de solo lectura, podemos abrir el archivo y visualizar el contenido, pero no está permitido hacer ninguna edición de ese contenido.

- **El modo de solo lectura en modo binario**

  - Se utilizará el indicador “rb”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo (lectura en modo binario), solo podemos abrir el fichero y visualizar su contenido en binario, pero no se puede realizar ninguna edición.

- **El modo de lectura y escritura**

  - Se utilizará el indicador “r+”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo, podremos realizar una visualización del contenido del fichero y su modificación.

- **El modo de lectura y escritura en modo binario**

  - Se utilizará el indicador “rb+”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo, podremos realizar una visualización del contenido del fichero y su modificación en binario.

- **El modo solo escritura**

  - Se utilizará el indicador “w”.

  - El puntero se posicionará al inicio del fichero.

- En este modo, solo podemos modificar el contenido del fichero; en caso de que ya exista, se sobrescribirá, y si no existe se creará el fichero.

- **El modo solo escritura en modo binario**

  - Se utilizará el indicador “wb”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo, solo podemos modificar el contenido del fichero en binario; en caso de que el fichero exista, se sobrescribirá, y, si no existe, se creará el fichero.

- **El modo lectura y escritura**

  - Se utilizará el indicador “w+”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo podremos visualizar el contenido y modificarlo; en caso de que el fichero exista, será sobrescrito y, si no existe, se creará el fichero.

- **El modo lectura y escritura en modo binario**

  - Se utilizará el indicador “wb+”.

  - El puntero se posicionará al inicio del fichero.

  - En este modo podremos visualizar el contenido y modificarlo en binario; en caso de que el fichero exista, se sobrescribirá y, si no existe, se creará el fichero.

- **El modo para añadir contenido**

  - Se utilizará el indicador “a”.

  - El puntero se posicionará al final del fichero si este existe; en cambio, si no existe, se posicionará al inicio.

  - En este modo, solo se podrá añadir el nuevo contenido al final del fichero si este existe; en caso contrario, creará el fichero.

- **El modo para añadir contenido en modo binario**

  - Se utilizará el indicador “ab”.

  - El puntero se posicionará al final del fichero si este existe; y si no existe, se posicionará al inicio.

  - En este modo, solo se podrá añadir el nuevo contenido en binario al final del fichero, si este existe, pero si no, se creará el fichero.

- **El modo para realizar lectura y añadir contenido**

  - Se utilizará el indicador “a+”.

  - El puntero se posicionará al final del fichero si este existe; y si no existe, se posicionará al inicio.

  - En este modo, solo se podrá añadir el nuevo contenido al final del fichero si este existe. En caso contrario, se creará el fichero y se podrá añadir al inicio, también podremos visualizar el contenido de este fichero.

- **El modo para realizar lectura y añadir contenido en modo binario**

  - Se utilizará el indicador “ab+”.

  - El puntero se posicionará al final del fichero si este existe y si no existe, se posicionará al inicio.

  - En este modo solo se podrá añadir el nuevo contenido en binario al final del fichero, si este existe. En caso contrario, se creará el fichero y se podrá añadir al inicio, también podremos visualizar el contenido de este fichero en binario.

Debemos tener en cuenta que si abrimos un fichero en el modo lectura y no existe, se generará un error (deberemos utilizar un *try except* para capturar la excepción), pero si se abre en modo escritura, Python nos creará el fichero al momento de abrirlo con cualquiera de estos comandos (“w”, “a”, “w+”, “a+”).

En el caso en que no se especifique modo, los ficheros siempre serán abiertos en modo de solo lectura.

Para saber más

Al abrir un fichero que ya existe en modo escritura, todo su contenido será borrado y será reemplazado por lo que se escriba en él.

Resumen

- Un fichero o archivo informático es un conjunto de bytes que se almacenan en un disco duro.
- Las operaciones que pueden realizarse con ficheros son: creación, apertura, cierre, lectura y edición.
- El puntero señala la posición desde donde se realizará cualquier operación en un fichero abierto.
- Según sea el modo de apertura, cambia la posición del fichero:
  - Modo de solo lectura: al inicio.
  - Modo de solo lectura en modo binario: al inicio.
  - Modo de lectura y escritura: al inicio.
  - Modo de lectura y escritura en modo binario: al inicio.
  - Modo solo escritura: al inicio.
  - Modo solo escritura en modo binario: al inicio.
  - Modo lectura y escritura: al inicio.
  - Modo lectura y escritura en modo binario: al inicio.
  - Modo para añadir contenido: al final si existe, o al inicio si no existe.
  - Modo para añadir contenido en modo binario: al final si existe, o al inicio si no existe.
  - Modo para realizar lectura y añadir contenido:al final si existe, o al inicio si no existe.
  - Modo para realizar lectura y añadir contenido en modo binario: al final si existe, o al inicio si no existe.

2. Módulos OS y *subprocess*

En esta unidad, veremos cómo podemos operar de dos formas para trabajar con archivos y directorios desde Python:

- **Desde el módulo OS:** como su nombre indica, nos abstrae de la problemática que podemos encontrar al trabajar en distintos sistemas operativos (OS), siendo distinta la gestión de directorios en Linux, Mac y Windows.

- **Tratando un archivo como un objeto:** esta forma es mucho más simple que la anterior y nos permite acceder al fichero para realizar lecturas y escrituras a nivel de la aplicación.

2.1. Módulo OS

El módulo OS nos facilita la creación de ficheros y directorios sin depender del sistema operativo en el que ejecutemos nuestro código fuente.

Para esto, ofrece cierta portabilidad en nuestro código sin depender de la máquina donde sea ejecutado, aunque no todas las funciones están disponibles en todos los sistemas operativos.

Vamos a estudiar varias funciones para administrar o crear procesos o contenidos en el sistema de ficheros.

Únicamente deberemos importar este módulo en nuestro código fuente para poder utilizarlo, mediante:

```
import os
```

2.1.1. Métodos más comunes

El módulo OS contiene varios métodos, así que especificaremos los más utilizados:

- **Os.access(ruta, modo_de_acceso):** nos permite saber si podemos acceder a un archivo o directorio en el modo deseado.

Certificación

Es importante interiorizar sobre módulo OS y *subprocess*, pues en numerosas preguntas de la certificación se hace referencia a estos.

- **os.F_OK ¿Existe la ruta del fichero?**

```
import os

existe = os.access("prueba.txt", os.F_OK)
print(existe)
```

- **os.R_OK ¿Tenemos acceso de lectura?**

```
import os

permiso = os.access("prueba.txt", os.R_OK)
print(permiso)
```

- **os.W_OK ¿Tenemos acceso de escritura?**

```
import os

permiso = os.access("prueba.txt", os.W_OK)
print(permiso)
```

- **os.X_OK ¿Tenemos acceso de ejecución?**

```
import os

permiso = os.access("prueba.txt", os.X_OK)
print(permiso)
```

Salida de todos los módulos anteriores: devolverá un booleano ("True" o "False").

- **Os.getcwd():** nos permite conocer la ruta actual.

```
import os

directorio = os.getcwd()
print(directorio)
```

Salida:

```
C:\Proyectos
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

- **Os.chdir(nueva_ruta):** nos cambia el directorio de trabajo, y si el directorio no existe, producirá una excepción.

```
import os

os.chdir('C:\\Users')
directorio = os.getcwd()
print(directorio)
```

Salida:

```
C:\Users
```

- **Os.chroot():** nos cambia el directorio de trabajo raíz.

```
import os

os.chroot("/home")
```

- **Os.chmod(ruta, permisos):** nos cambia los permisos de un archivo o directorio.

```
import os, stat

os.chmod("archivo.txt", stat.S_IREAD)
os.chmod("archivo.txt", stat.S_IROTH)
```

- **Os.chown(ruta, permisos):** nos cambia el propietario de un archivo o directorio.

```
import os

ruta = "archivo.txt"
print(os.stat(ruta).st_uid)
print(os.stat(ruta).st_gid)

uid = 2000
gid = 2000
os.chown(ruta, uid, gid)

print(os.stat(ruta).st_uid)
print(os.stat(ruta).st_gid)
```

Salida

```
0
0
2000
2000
```

- **Os.mkdir(ruta, [, modo])):** nos crea un directorio.

```
import os

directorio  = “Prueba”
padre = “Carpeta”
ruta = os.path.join(padre _dir, directorio )
os.mkdir(ruta )
```

- **Os.makedirs(ruta, [, modo])):** nos crea directorios recursivamente.

```
import os

directorio  = “Prueba”
padre = “Carpeta”
ruta = os.path.join(padre, directorio)
os.makedirs(ruta)
```

- **Os.remove(ruta):** nos elimina un archivo.

```
import os

ruta = “archivo.txt”
os.remove(ruta)
```

- **Os.rmdir(ruta):** nos elimina un directorio.

```
import os

ruta = “Carpeta”
os.rmdir(ruta)
```

- **Os.removedirs(ruta):** nos elimina un directorio recursivamente.

```
import os
```

```
ruta = “Carpeta”
os.removedirs(ruta)
```

- **Os.rename(ruta_fichero_actual, nuevo_nombre):** cambia el nombre de un archivo.

```
import os

os.rename(“archivo.txt”, “archivo_1.txt”)
```

- **Os.symlink(ruta, nombre_destino):** crea un vínculo simbólico.

```
import os

# os.symlink(src, dst)
os.symlink(“archivo.txt”, “arc(link).txt”)
```

Es importante señalar que este módulo dispone de un diccionario que incluye las variables de entorno relacionadas con el sistema operativo en el que ejecutamos nuestro programa, utilizando “os.environ”.

```
import os

for variable, valor in os.environ.items():
    print(“%s: %s” % (variable, valor))
```

2.1.2. Submódulo *path* (os.path)

Este submódulo permite tener acceso a funcionalidades en la ruta de los ficheros y directorios. Las más utilizadas son:

- **Os.path.abspath(ruta):** ruta absoluta (responde una *string*).

```
import os

archivo = “archivo.txt”
print(os.path.abspath(archivo))
```

Salida:

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
/home/Doc/archivo.txt
```

- **Os.path.basename(ruta):** directorio base (responde un *string*).

```
import os

archivo = "/home/Doc"
print(os.path.basename(ruta))
```

Salida:

```
Doc
```

- **Os.path.exists(ruta):** nos devuelve si esta ruta existe (responde con un booleano).

```
import os

archivo = "/home/Doc"
print(os.path.exists(ruta))
ruta = "/home/Doc/archivo.txt"
print(os.path.exists(ruta))
```

Salida:

```
True
False
```

- **Os.path.getatime(ruta):** nos devuelve el último acceso al directorio en segundos, en caso de que no exista se genera un OSError (responde con un *float*).

```
import os
import time

ruta = "archivo.txt"
print(os.path.getatime(ruta))
#Podemos utilizar time.ctime del módulo time
#Mostrará la fecha
print(time.ctime(os.path.getatime(ruta)))
```

Salida:

```
1619816212.8832314
Fri Apr 30 20:56:52 2021
```

- **Os.path.getsize(ruta):** nos devuelve el tamaño del directorio en bytes; en caso de que no exista, se genera un OSError (responde con un entero).

```
import os

archivo = "archivo.txt"
print(os.path.getsize(ruta))
```

Salida:

```
128
```

- **Os.path.isfile(ruta):** nos devuelve si la ruta es un archivo (responde con un booleano).

```
import os

archivo = "archivo.txt"
print(os.path.isfile(ruta))
ruta = "/home/Doc/"
print(os.path.isfile(ruta))
```

Salida:

```
True
False
```

- **Os.path.isabs(ruta):** nos devuelve si una ruta es absoluta (responde con un booleano).

```
import os

archivo = "archivo.txt"
print(os.path.isabs(ruta))
```

```
ruta = "../Doc/"
print(os.path.isabs(ruta))
```

Salida:

```
True
False
```

- **Os.path.isdir(ruta):** nos devuelve si una ruta es un directorio (responde con un booleano).

```
import os

archivo = "archivo.txt"
print(os.path.isdir(ruta))
ruta = "/home/Doc/"
print(os.path.isdir(ruta))
```

Salida:

```
False
True
```

- **Os.path.islink(ruta):** nos devuelve si la ruta es un acceso directo (responde con un booleano).

```
import os

archivo = "/home/Doc/Archivo(original).txt"
print(os.path.islink(ruta))
ruta = "/home/Doc/Archivo(acceso_directo).txt"
print(os.path.islink(ruta))
```

Salida:

```
False
True
```

- **Os.path.ismount(ruta):** nos devuelve si la ruta es un punto de montaje (responde con un booleano):

```
import os

#En Windows
ruta = "C:"
print(os.path.ismount(ruta))
```

Salida:

```
True
```

- **Os.path.split(ruta):** divide la ruta en dos, la parte del fichero y la ruta del directorio.

```
import os

ruta = "/home/Doc/archivo.txt"
print(os.path.split(ruta)[0])
print(os.path.split(ruta)[1])
```

Salida:

```
/home/Doc
archivo.txt
```

2.2. Módulo *subprocess*

Este módulo nos permite trabajar directamente con órdenes del sistema operativo en el que ejecutemos nuestro programa.

Uno de los métodos más comunes es *call* (orden), dentro podremos ejecutar órdenes sencillas como *clear* (limpiar la pantalla) y, si el comando necesita argumentos, este método necesita recibir una lista siendo el primer elemento el comando y los argumentos el segundo, por ejemplo, listar un directorio:

```
from subprocess import call

call('clear')

variable = ['ls', '-lha']
call(variable)
```

Para saber más

Un punto de montaje es una ruta en un sistema de archivos, donde se ha montado un sistema de archivos diferente.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

2.2.1 Submódulo *Popen*

Es un objeto de *subprocess* y para utilizarlo debemos importar de la misma forma que *call*, pero tecleando "Popen", como parámetro le pasaremos un comando o una lista en la cual el primer elemento será el comando y el segundo los argumentos.

Debemos tener en cuenta el hecho de asignar la salida de la llamada a *Popen* y guardarla en una variable para crear el objeto *Popen*, para así poder acceder a sus métodos y evitar que el proceso se quede en ejecución. De este modo llamamos al método wait() de este objeto y esperamos que el proceso finalice.

```
from subprocess import Popen

salida = Popen(['ls', '-lha'])
salida.wait()
```

Popen nos ofrece la forma de capturar la salida mediante *stdout* (salida), *stdin* (entrada), *stderr* (error).

Para completar el objeto "Popen()" y poder capturar la entrada, la salida o un error, debemos pasar como argumento alguna de las anteriores opciones. Para hacerlo debemos importar una tubería que nos proporciona *subprocess* llamada *PIPE*, así podemos a partir de ese momento añadir "stdout=PIPE" y "stderr=PIPE" en la llamada a *Popen*. Al capturar la salida no hace falta utilizar la función "wait()", pues será capturado directamente por la tubería y accederemos mediante la función "read()" para leer su contenido como si de un fichero se tratara.

Debemos recordar cerrar correctamente los ficheros y, si los almacenamos en variables, podemos manipular las lecturas como un *string*.

```
from subprocess import Popen

proceso = Popen(['ls', '-ha'], stdout=PIPE, stderr=PIPE)
error_econtrado = proceso.stderr.read()
proceso.stderr.close()
listado = proceso.stdout.read()
proceso.stdout.close()

print(listado)
```

Resumen

- El módulo OS nos ofrece funciones para la creación de ficheros y directorios independientemente del sistema operativo en el que estemos trabajando. Por lo tanto, nos aporta independencia de la máquina donde esté ejecutándose nuestro *software*.

- El submódulo *path* del módulo OS nos da acceso a funcionalidades "extendidas" para el trabajo con rutas y directorios.

- El módulo *subprocess* nos permite usar funciones nativas y/o propias del sistema operativo en el que estemos trabajando. Si bien no ofrece independencia de la máquina, lo que sí ofrece es mayor rendimiento.

- El submódulo *Popen* del módulo *subprocess* nos brinda la posibilidad de capturar la entrada y salida de datos del fichero, así como el tratamiento de errores.

3. Métodos y propiedades del objeto *file*

En esta unidad vamos a ver los distintos métodos para gestionar archivos desde nuestro código fuente.

El objeto ***file*** se crea a partir de la llamada de la función *open*, incluyendo el modo de apertura y el fichero que se desea abrir. De forma habitual, la almacenaremos en una variable para poder realizar acciones posteriores sobre el contenido del fichero.

3.1. Apertura de ficheros

Tenemos disponibles dos formas para utilizar un fichero, como archivo de texto, procesando línea por línea, o como archivo binario y procesarlo byte a byte.

Para abrir un fichero utilizaremos la función "open()". Esta función admite dos parámetros, que son el fichero que deseamos abrir y el modo en el que queremos abrirlo. Si este último no lo pasamos a la función, por defecto lo abrirá en modo de solo lectura.

Ejemplo:

```
fichero = open("prueba.txt")
```

Certificación

Es importante interiorizar sobre apertura de ficheros, pues en numerosas preguntas de la certificación se hace referencia a este tema.

Esta función devuelve un objeto que nos permite realizar acciones sobre el fichero indicado en la función, y podremos almacenar ese valor en una variable.

La operación inicial en un fichero será leer el contenido y podemos hacerlo de dos formas: una es línea por línea y la otra es mediante la función "readlines()" para guardar directamente todo el contenido en una variable.

Para realizar la acción de lectura línea por línea podemos realizarlo con un bucle *while*, mediante la función "readline()":

```
fichero = open("prueba.txt")
linea = fichero.readline()
print(linea)
while linea != '':
        linea = fichero.readline()
```

Cada vez que se ejecute la función "readline()", el puntero pasará a la siguiente línea, hasta llegar al final del fichero.

Podemos realizar la misma acción utilizando un bucle *for*:

```
fichero = open("prueba.txt")
for linea in fichero:
    print(línea)
```

Con cada iteración del bucle *for*, la variable "línea" irá cogiendo el valor de una línea nueva del fichero, hasta llegar al final.

Si queremos almacenar directamente todo el fichero en una variable para gestionarlo después, utilizaremos la función "readlines()", que obtiene todas las líneas de un fichero en una llamada.

```
fichero = open("prueba.txt")
lineas_fichero = fichero.readlines()
    print(línea)
```

La variable "líneas_fichero" contendrá una lista de cadenas de texto con todas las líneas del fichero, podremos acceder a ellas en cualquier momento.

```
fichero = open("prueba.txt")
lineas_fichero = fichero.readlines()
if(len(lineas_fichero)>5):
    print(líneas fichero[2])
```

Debemos tener cuidado de no realizar un "readline()" antes de hacer la llamada a la función "readlines()", ya que contará desde la ubicación del puntero.

Para saber más

Debemos tener en cuenta que hay que utilizar la función "readlines()" solamente en archivos pequeños, ya que estamos cargando en memoria el archivo completo y podríamos consumir toda la memoria disponible en el *hardware*, dejando a nuestro programa colgado o inestable.

3.2. Cerrar fichero

Una vez que hemos terminado de utilizar nuestro fichero abierto con la función “open()” es recomendable cerrarlo con la función “close()”.

1. En varios sistemas operativos solo pueden ser abiertos por un programa.

2. En otros, lo que se haya escrito no se guarda realmente en el fichero hasta que se realiza la función “close()”.

3. El límite de la cantidad de archivos que podemos abrir simultáneamente en un programa es bajo.

Tras realizar una apertura de un fichero y realizar la lectura línea a línea en un bucle *for*, podemos encontrarnos con la problemática de que las líneas incluyen un fin de línea (“\\n”). Para solucionarlo, tan solo tenemos que eliminarlo, mediante la llamada a *rstrip*.

Ejemplo:

```
fichero = open(“prueba.txt”)
for i, línea in enumerate(fichero):
                        linea = linea.rstrip(“\\n”)
                        print(“Nº%2d - Contenido -> %s” % (i, linea))
fichero.close()
```

El método *enumerate* nos indica la posición de la línea dentro del fichero.

3.3. Lectura o escritura de un fichero en binario

Muchas veces tendremos ficheros que no contendrán texto y, por lo tanto, no podrán ser procesados por líneas. Debido a esto, tendremos la necesidad de recorrer el fichero byte a byte. Además, necesitamos conocer el formato en el que se encuentran los datos para poder procesar el contenido.

De los modos estudiados anteriormente, utilizaremos los que incluyen la *b*, ya sea de solo lectura, escritura o ambos.

Utilizaremos la función “read(n)” para leer *n* bytes del fichero y la función “write(contenido)” para escribir contenido en la posición actual del puntero en el fichero.

En estos archivos binarios, es muy útil conocer la posición del puntero en el archivo utilizando la función "tell()", que nos indicará la cantidad de bytes de que dispone el fichero desde el inicio.

Para editar la posición del puntero utilizaremos la función "seek(inicio, apartir_de)". Esta nos dará el control completo para desplazar el puntero: con el primer parámetro definiremos el desplazamiento en una cantidad de bytes en el fichero y en el segundo, desde dónde empieza a contar el desplazamiento.

```
fichero = open("prueba.txt"","rb")
data = fichero.read()
print(data)

fichero.seek(3,2)
print(fichero.tell()) #Comprobar posición

data2 = fichero.read()
print(data2)

fichero.seek(0) #Mover puntero a la posición 0
print(fichero.tell())#Comprobar posición
data2 = fichero.read()
print(data2)
```

3.4. Escritura de un fichero

Después de realizar la apertura de un archivo en modo escritura, podemos añadir cadenas o modificar el contenido del fichero de dos formas:

- Cadena a cadena utilizando la función "write(cadena)".

- Mediante listas de cadenas utilizando la función "writelines(lista)".

Para añadir contenido en el fichero, debemos añadir a cada línea los caracteres de fin de línea "\n", para poder formarlo correctamente.

```
fichero = open("prueba.txt"","w+") #r+
cadena = "Añadir línea"
fichero.seek(0, 2)
line = fichero.write( cadena )
line = fichero.write( cadena )
line = fichero.write( cadena )
```

Para saber más

Si abrimos un fichero en modo escritura ("w" o "w+") y el fichero existe, se borrará todo su contenido y se sobrescribirá en él. Si no existe, se creará uno nuevo.

```
#fichero.writelines([cadena,cadena,cadena])
fichero.seek(0,0)
for i,linea in enumerate(fichero):
   print("Línea Nº %d - %s" % (i, linea))
fichero.close()
```

Salida:

```
Línea Nº 0 - Añadir línea
Línea Nº 1 - Añadir línea
Línea Nº 2 - Añadir línea
```

Si utilizamos el modo "w+" el fichero será sobrescrito por completo; si utilizamos el modo "r+" las líneas serán añadidas al final.

3.5. Añadir texto a un fichero

Para añadir texto a un fichero sin perder el contenido anterior, debemos utilizar el modo de apertura *(a)*.

Uno de los usos más comunes es realizar un fichero de *log* y poder registrar los distintos eventos ocurridos en un programa, ayudándonos a encontrar una secuencia errónea o desconocida ocurrida en nuestro código fuente.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
import datetime

fichero = open("prueba.txt", "a")
hora_actual = str(datetime.datetime.now())
mensaje = "Error en el sistema"
fichero.write(hora_actual+" "+mensaje+"\\n")
fichero.close()
```

Contenido del fichero prueba.txt

```
2021-05-01 21:34:17.291111 Error en el sistema\n
```

3.6. Propiedades del objeto *file*

Una vez creado el objeto *file*, tendremos disponibles una serie de propiedades que podremos usar para realizar acciones sobre este objeto o que,

en cualquier caso, nos indicarán características sobre el objeto abierto y almacenado en una variable.

- ***closed*:** si el fichero está cerrado devuelve "True"; si no, devuelve "False".

- ***mode*:** devuelve en qué modo ha sido abierto.

- ***name*:** devuelve el nombre del archivo.

- ***encoding*:** devuelve la codificación del archivo de texto.

```
fichero = open("prueba.txt", "r+")
contenido = fichero.read()
nombre = fichero.name
modo = fichero.mode
codificacion = fichero.encoding
fichero.close()
print(contenido," ",nombre," ",modo," ",codificacion)
if fichero.closed:
     print("El archivo está cerrado")
else:
     print("El archivo está abierto")
```

Salida:

```
ContenidoFichero   prueba.txt   r+   UTF-8
El archivo está cerrado
```

3.7. Utilización de *with* para abrir un fichero

Tenemos una segunda opción para abrir nuestro fichero con el comando *open* y es utilizando la instrucción *with* delante. De esta manera, obtenemos un mayor control de excepciones y además no tenemos que estar pendientes de llevar el control de cerrar el fichero previamente abierto, ya que esta tarea se realizará automáticamente.

```
with open('prueba.txt', 'w') as fichero:
    fichero.write('HOLA')
```

El fichero no podrá ser utilizado fuera del *with*. En caso de intentar utilizarlo, saldrá un error, indicando que el fichero está cerrado.

```
with open('prueba.txt', 'w') as fichero:
    fichero.write('HOLA')
fichero.write('ADIOS')
```

Salida:

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

3.8. Errores en la manipulación de ficheros

Durante la manipulación de ficheros en nuestros programas, nos podemos encontrar con distintas excepciones en tiempo de ejecución, que debemos controlar para que nuestro código fuente no termine de forma inesperada y poder generar avisos previamente.

- **El fichero no se puede abrir**
  - `IOError`
- **El fichero está cerrado**
  - `ValueError: I/O operation on closed file.`

    Si se intenta acceder a un fichero que ha sido cerrado anteriormente, ya sea con el método "close()" o fuera de la instrucción *with*.
- **El fichero no se encuentra**
  - `FileNotFoundError: [Errno 2] No such file or directory: 'prueba.txt'`

    Si intentamos acceder a un fichero que no existe en modo lectura, nos devolverá esta excepción.
- **El fichero ya existe**
  - `FileExistsError`
- **Acceder a un fichero sin permisos**
  - `PermissionError`

Ejercicios resueltos

Practica el **ejercicio 1** con los conocimientos adquiridos hasta esta unidad.

El error será mostrado en caso de que no tengamos permisos de alguna acción en dicho fichero.

- **Intentar escribir en un fichero que ha sido abierto en modo lectura:**

  - `UnsupportedOperation: not writable`

  Si abrimos un fichero en modo solo lectura, no podremos escribir en él; por lo tanto, si nuestro código lo intenta, aparecerá una excepción de este tipo.

Como resumen y en previsión de tener trazabilidad de todos y cada uno de los posibles errores que nos podamos encontrar... y para que nuestros programas no terminen de forma inesperada, debemos incorporar la utilidad de *try*, *except*, y para poder manejar las excepciones o realizar mensajes al usuario por pantalla.

```
import sys

while (True):
  fichero = input("Introduce el nombre de un fichero: ")
  modo = input("Modo de apertura (r,w)")
  try:
    with open(fichero, modo) as fichero:

        fichero.write('HOLA')
    fichero.write('ADIOS')
  except Exception as ex:
    print(ex)
    exc_type, exc_obj, exc_tb = sys.exc_info()
    fname = fichero.name
    print(exc_type, fname, exc_tb.tb_lineno)
```

Salida:

```
Introduce el nombre de un fichero: w
Modo de apertura (r,w)w
I/O operation on closed file.
<class 'ValueError'> w 10
Introduce el nombre de un fichero: |
```

Capturando las excepciones, nuestro programa nunca finalizará de manera inesperada, ya que informará del error al usuario por pantalla y, dado que tendremos en ejecución un bucle *while* infinito, siempre

volverá a solicitar un nuevo fichero para seguir ejecutando nuestro código.

Si no utilizamos la captura de las excepciones, en el primer error nuestro código dejará de ejecutarse, el flujo de datos se interrumpirá y nos sacará del programa, aunque estuviese dentro de un bucle infinito. Al saltar una excepción no controlada, el programa no puede continuar, porque no tiene un método para capturarla o manejarla, ni para avisarnos de un final inesperado.

Resumen

- Al abrir un fichero, podemos hacerlo como archivo de texto, procesando línea por línea, o como archivo binario y procesarlo byte a byte.

- Una vez que hemos terminado de utilizar nuestro fichero abierto con la función “open()” es recomendable cerrarlo con la función “close()”.

- Después de realizar la apertura de un archivo en modo escritura, podemos añadir cadenas o modificar el contenido del fichero cadena a cadena utilizando la función “write(cadena)” o mediante listas de cadenas utilizando la función “writelines(lista)”.

- Mediante la instrucción *with* obtendremos un mayor control de las excepciones y no tendremos que preocuparnos de estar pendientes de cerrar el fichero.

4. Tratamiento de ficheros JSON

JSON es un formato para el intercambio de datos basado en texto. Es uno de los más populares para serializar datos y se pueden obtener como resultado en muchas de las aplicaciones API REST y aplicaciones web, por ello es muy probable que necesitemos leer ficheros JSON.

En esta unidad, veremos en profundidad los ficheros JSON, qué información nos pueden aportar y cómo tratarlos.

4.1. Características de JSON

JSON es fácilmente legible por una persona, ya que en su origen se encuentra enlazado con JavaScript, pero hoy en día podemos encontrarlo presente como fichero contenedor de información en muchos lenguajes de programación. También, es una alternativa para tener en cuenta a XML, que es otro formato de intercambio de datos, que es más ineficiente con respecto a JSON, porque consume más recursos.

La estructura de ficheros JSON son parejas (tuplas) de nombre + valor, separados por dos puntos. Esas parejas (o tuplas de datos) se separan mediante coma y todas ellas se engloban dentro de llaves.

El valor de una pareja puede ser otro JSON e ir anidando continuamente, para ofrecer mayor flexibilidad en la estructura de la información.

Para crear un fichero JSON en forma sencilla desde Python, debemos exportar los datos que contiene un objeto diccionario. Estos objetos diccionarios pueden contener varios tipos de datos (*int*, *str*, *bool*...) u objetos. Para poder escribir el diccionario en este formato, utilizaremos el paquete disponible JSON, mediante la importación en nuestro programa.

Certificación

Es importante interiorizar sobre las características de JSON, pues en numerosas preguntas de la certificación se hace referencia a este tema.

Vamos a desarrollar un ejemplo, en que crearemos un objeto diccionario y agregaremos varios usuarios con las propiedades “nombre”, “apellidos” y “edad”.

Abriremos el fichero, utilizando el paquete JSON y la función “json.dump(data, fichero, indent=4)” de parámetros que necesita el objeto para pasar a *JSON (data)*, el fichero donde queremos que se almacene,

y en el tercer parámetro indicamos la cantidad de espacios de identificación para que el fichero sea mucho más legible.

```
import json

data = {}
data['usuario'] = []
data['usuario'].append({
    'nombre': 'Josep',
    'apellidos': 'Estarlich',
    'edad': 29})
data['usuario'].append({
    'nombre': 'Claudio',
    'apellidos': 'Garcia',
    'edad': 35})
data['usuario'].append({
    'nombre': 'Claudio',
    'apellidos': 'Gomez',
    'edad': 33})
with open('prueba.txt', 'w') as fichero:
    json.dump(data, fichero, indent=4)
```

Una vez que ejecutamos nuestro código, se crea un fichero llamado prueba.txt; si este no existe, borra todo el contenido del fichero y añade el nuevo contenido.

```
{
  "usuario": [
    {
      "nombre": "Josep",
      "apellidos": "Estarlich",
      "edad": 29
    },
    {
      "nombre": "Claudio",
      "apellidos": "Garcia",
      "edad": 35
    },
    {
      "nombre": "Claudio",
      "apellidos": "Gomez",
      "edad": 33
    }
```

Para saber más

Si no necesitamos que el JSON sea legible por una persona, podemos no utilizar el tercer parámetro y así reduciremos mucho el tamaño del fichero, ya que aparecerá todo en la misma línea.

Para obtener el contenido de un diccionario en JSON dentro de una variable utilizaremos “json.dumps”. Devuelve una cadena de texto con los datos en JSON del diccionario sin necesidad de almacenarlos en un fichero.

```
import json

data = {}
data['usuario'] = []
data['usuario'].append({
    'nombre': 'Josep',
    'apellidos': 'Estarlich',
    'edad': 29})
data['usuario'].append({
    'nombre': 'Claudio',
    'apellidos': 'Garcia',
    'edad': 35})
data['usuario'].append({
    'nombre': 'Claudio',
    'apellidos': 'Gomez',
    'edad': 33})
contenido =  json.dumps(data)
print(contenido)
```

Salida:

```
{"usuario": [{"nombre": "Josep", "apellidos": "Estarlich", "edad": 29}, {"nombre": "Claudio", "apellidos": "Garcia", "edad": 35}, {"nombre": "Claudio", "apellidos": "Gomez", "edad": 33}]}
```

Para leer este tipo de ficheros JSON, utilizaremos el método *json.load(fichero)* y en el único parámetro les pasaremos el fichero que deseamos leer.

Este método nos devolverá un objeto de tipo diccionario sobre el cual podremos iterar.

```
import json

with open('prueba.txt') as fichero:
    data = json.load(fichero)
    for usuario in data['usuario']:
        print(usuario['nombre'])
        print(usuario['apellidos'])
        print(usuario['edad'])
```

Salida:

```
Josep
Estarlich
29
Claudio
Garcia
35
Claudio
Gomez
33
```

De la misma forma que para la lectura, podemos tener la necesidad de convertir una cadena de texto en JSON a un diccionario. Para esto, utilizaremos el método "json.loads(cadena)" y en el parámetro del método insertaremos la cadena que queremos convertir en objeto.

```
import json

variable ='''{
    "usuario": [
        {
            "nombre": "Claudio",
            "apellidos": "Garcia",
            "edad": 29
        },
        {
            "nombre": "Isabel",
            "apellidos": "Ferragud",
            "edad": 35
        },
        {
            "nombre": "Sheila",
            "apellidos": "Martinez",
            "edad": 27
        }
    ]
}'''

data = json.loads(variable)
for usuario in data['usuario']:
        print(usuario['nombre'])
        print(usuario['apellidos'])
        print(usuario['edad'])
```

Para saber más

A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un "método__init__()".

Salida:

```
Claudio
Garcia
29
Isabel
Ferragud
35
Sheila
Martinez
27
```

Utilizaremos este método cuando realicemos una llamada a una API REST y la respuesta sea con una cadena de texto, mediante un objeto JSON para poder transformarlo a un diccionario.

4.2. Opciones de la librería JSON

Podemos personalizar el comportamiento de la librería pasando por argumentos distintos valores en la función "json.dumps". Además, se puede ordenar una colección y utilizar una codificación diferente a la inicial por defecto, que es ASCII, etc.

4.2.1 Codificación unicode

El paquete JSON, por defecto y tras ejecutar el método "json.dumps(cadena)", genera el resultado en código ASCII, por lo tanto, si existen caracteres que no son ASCII, serán escapados. Sin embargo llegados a este punto, puede ser más interesante utilizar una codificación diferente: la codificación unicode, para que el fichero generado sea más legible.

Para conseguir este cambio, utilizaremos la variable "ensure_ascii = False".

```
import json

data = {'nombre': 'Paco', 'Apellido': 'Rodríguez'}
print(json.dumps(data))
print(json.dumps(data, ensure_ascii=False))
```

Salida:

```
{“nombre”: “Paco”, “Apellido”: “Rodr\u00edguez”}
{“nombre”: “Paco”, “Apellido”: “Rodríguez”}
```

4.2.2. Ordenación

Los objetos JSON son un conjunto de varias tuplas del estilo [clave / valor]. Los datos se añadirán al diccionario en el mismo orden en el que están definidos en el fichero. Además, el paquete JSON nos permite ordenar mediante la clave y podemos configurarlo con la opción “sort_keys=True”.

```
import json

data = {
    ‘x’: ‘1’,
    ‘y’: ‘2’,
    ‘a’: 3,
    ‘h’: 4}
print(json.dumps(data, sort_keys=True))
```

Salida:

```
{“a”: 3, “h”: 4, “x”: “1”, “y”: “2”}
```

4.2.3 Valores *float* fuera de rango

Si tenemos la necesidad de utilizar un *float* fuera de rango: float(’nan’). Debemos pasar por argumento la variable “allow_nan = True” al método “json.dumps()”.

```
import json

dict = {
    2: ‘3’,
    3: ‘4’,
    4: float(‘nan’)
}

json.dumps(dict, allow_nan=True)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
{“2”: “3”, “3”: “4”, “4”: NaN}
```

4.3. Mapeo de tipos de datos

Para utilizar la serialización en JSON, ya que este tipo de fichero no es específico de Python, debemos tener encuentra las relaciones, porque las mayúsculas y minúsculas son significativas:

Python3	JSON
dictionary	*object*
list	*array*
string	*string*
integer	*integer*
float	*real number*
True	*true*
False	*false*
None	*null*

JSON no tiene soporte para objetos de tipo byte o *array* de bytes. Pero, utilizando el módulo JSON, podemos codificar y decodificar tipos de datos desconocidos.

Si nos encontrásemos con cualquier tipo de datos que no esté soportado de forma nativa, nos veríamos obligados a desarrollar el codificador y decodificador para este tipo de datos.

Definiremos una función para serializar tipos de datos no soportados por JSON. Esta función coge un objeto como parámetro que será el objeto que no es serializable directamente. Después crearemos una comprobación para determinar el tipo de objeto que entra en la función y poder gestionar cada tipo de forma predeterminada. Por último, tendremos que hacer uso de la función “list()” para convertir nuestro objeto byte en una lista de enteros.

```
def to_json(python_object):
  if isinstance(python_object, bytes):
        return {‘__class__’:’bytes’,
```

```
            '__value__': list(python_object)}
  raise TypeError(repr(python_object) + 'no serializable')
```

Para utilizar la función en nuestro código fuente, debemos incluir en los argumentos de la función "json.dump()" la variable "default = to_json" (al nombre de la función).

```
import time,json

entry = {
'comentario': None,
 'id': b'548',
 'titulo': 'Robo en Alemania',
 'publicada': True}
import json
with open('prueba.txt', 'w', encoding='utf-8') as f:
    json.dump(entry, f,  default=to_json)
```

Contenido del fichero prueba.txt:

```
{"comentario": null, "id": {"clase": "bytes", "valor": [53, 52,
56]}, "titulo": "Robo en Alemania", "publicada": true}
```

Del mismo modo para poder deserializar nuestro fichero JSON debemos crear otra función que haga la conversión inversa:

```
def from_json(json_object):
    if '__class__' in json_object:
        if json_object['__ class__'] == 'bytes':
            return bytes(json_object['__value__'])
    return json_object
```

Para utilizar la función en nuestro código debemos incluir en los argumentos de la función "json.load()" la variable "object_hook=from_json".

```
import json

with open('prueba.txt', 'r+', encoding='utf-8') as f:
    print(json.load(f, object_hook=from_json))
```

Salida:

```
{'comentario': None, 'id': b'548', 'titulo': 'Robo en Alemania',
'publicada': True}
```

Ejercicios resueltos

Practica el **ejercicio 2** con los conocimientos adquiridos hasta esta unidad.

Resumen

- JSON es un formato para el intercambio de datos basado en texto y se usa en muchas de las aplicaciones API REST y aplicaciones web.

- Por defecto, el paquete JSON genera el resultado en código ASCII, pero es mucho más funcional utilizar la codificación unicode, para que el fichero generado sea más legible.

- JSON no tiene soporte para objetos de tipo byte o *array* de bytes. Pero, utilizando el módulo JSON, podemos codificar y decodificar tipos de datos desconocidos.

- Si tenemos un tipo de datos que no esté soportado de forma nativa, nos veremos obligados a desarrollar el codificador y decodificador para poder trabajar con él.

5. Serialización de datos en ficheros

La serialización es un proceso de codificación de un objeto, ya sea en un fichero o en una variable con el fin de transmitirlo por internet o a un formato legible. También, se puede utilizar para almacenar una configuración y poder volver a leerla en una ejecución posterior.

Este proceso también es utilizado para transmitir objetos entre aplicaciones o entre distintas aperturas de una misma aplicación.

Para la serialización de ficheros, vamos a estudiar una capacidad, llamada persistencia, para guardar información de un programa y poder utilizarla a posteriori.

5.1. La persistencia

La persistencia suele involucrar un proceso de serialización de datos, así como el proceso de cargar datos a nuestro programa a partir de información serializada en anteriores ejecuciones. Por ejemplo, podemos almacenar la configuración de nuestra base de datos o la información de puntos de un partido de fútbol.

Toda esta información puede ser almacenada de varias formas para facilitar su lectura, por lo que la almacenaremos en un archivo de texto, donde cada equipo ocupará una línea y los valores estarán separados por comas.

Hay que tener en cuenta que, si almacenamos valores numéricos, debemos pasarlos a cadena para guardarlos en el fichero y, al contrario, para tener el valor en numérico.

```
def guardar_en_fichero(nombre_archivo, datos):

    fichero = open(nombre_archivo, "w")
    for equipo, goles in datos:
        fichero.write(equipo+","+str(goles)+"\n")
    fichero.close()

def leer_de_fichero(nombre_archivo):
    datos = []
    fichero = open(nombre_archivo, "r")
```

Campus online

Accede al Campus online para ver el videotutorial *Generación de ficheros a través de scrapping desde web.*

```
    for línea in fichero:
        equipo, goles = línea.rstrip("\n").split(",")
        datos.append((equipo,int(goles)))
    fichero.close()
    return datos

datos = [("Madrid", 108), ("Vilareal", 102), ("Valencia", 115)]
guardar_en_fichero("almacen.txt",datos)
leer_de_fichero("almacen.txt")
```

Salida:

```
[('Madrid', 108), ('Vilareal', 102), ('Valencia', 115)]
```

También, podemos guardar el estado actual de un programa para realizarlo en un archivo de texto o en un archivo binario.

Utilizaremos un archivo de texto siempre que necesitemos que sea legible o cuando queramos tener la posibilidad de poder editarlo manualmente. Sin embargo, hay que tener en cuenta que, en este formato estaremos desperdiciando espacio, de ahí que aplicaciones de audio o vídeo almacenen ficheros en formato binario, ya que tienen la necesidad imperiosa de optimizar al máximo el espacio.

También, hay que destacar que en nuestro código anterior no realizamos ninguna comprobación si nos incluyen un nombre de equipo con una ",". Por eso, siempre es recomendable utilizar el módulo de CSV, por ejemplo, ya que tendrá en cuenta mucha casuística que nuestro código no tenía previsto.

Por lo tanto, vamos a ver dos variantes de persistencia: en **archivos CSV** y en **archivos binarios**.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

5.1. Archivos CSV

El archivo CSV es un formato que se utiliza para compartir datos entre aplicaciones, estos pueden ser leídos o procesados directamente.

En este formato, la primera línea indica el nombre de las propiedades y las siguientes su contenido.

Cada campo está separado por coma y los campos que son cadena, pueden incluirse tal como están entre comillas dobles. En el caso de que nos encontremos con alguna comilla doble, se escaparía de la forma \" y la contrabarra se pondría doble \\.

Para gestionar desde Python este tipo de ficheros, vamos a utilizar el módulo CSV, que nos ayudará a realizar la lectura o la escritura de forma muy sencilla, evitándonos la escritura de muchísimo código para contemplar una gran cantidad de casos a tener en cuenta, como ya se ha dicho.

Utilizando el mismo código anterior, pero mediante el módulo CSV:

```
import csv

def guardar_en_fichero(nombre_archivo, datos):

    fichero = open(nombre_archivo, "w")
    fichero_csv = csv.writer(fichero)
    fichero_csv.writerows(datos)
    fichero.close()

def leer_de_fichero(nombre_archivo):
    datos = []
    fichero = open(nombre_archivo, "r")
    fichero_csv = csv.reader(fichero)
    for equipo, goles in fichero_csv:
        datos.append((equipo,int(goles)))
    fichero.close()
    return datos

datos = [("Madrid", 108), ("Vilareal,Mallorca", 102), ("Valencia", 115)]
guardar_en_fichero("almacen.txt",datos)
leer_de_fichero("almacen.txt")
```

Salida:

```
[('Madrid', 108), ('Vilareal,Mallorca', 102), ('Valencia', 115)]
```

5.2. Archivos binarios

Para almacenar nuestros datos en archivos binarios utilizaremos la herramienta llamada *pickle*, porque nos permite de una forma sencilla realizar la lectura y la escritura en este formato.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Por otro lado, hay que tener en cuenta que no será fácil acceder a los datos de este documento desde cualquier otro programa que no esté desarrollado en Python.

```
import pickle

def guardar_en_fichero(nombre_archivo, datos):

    fichero = open(nombre_archivo, "wb")
    pickle.dump(datos,fichero)
    fichero.close()

def leer_de_fichero(nombre_archivo):
    datos = []
    fichero = open(nombre_archivo, "rb")
    datos = pickle.load(fichero)
    fichero.close()
    return datos

datos = [("Madrid", 108), ("Vilareal,Mallorca", 102), ("Valencia", 115)]
guardar_en_fichero("almacen.txt",datos)
leer_de_fichero("almacen.txt")
```

Salida:

```
[('Madrid', 108), ('Vilareal,Mallorca', 102), ('Valencia', 115)]
```

Con *pickle* podemos almacenar:

- Todos los tipos de Python nativos.
- Listas, diccionarios, conjuntos de varios objetos, etc.
- Funciones e instancias de clases con algunas limitaciones.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

5.3. Serializar un objeto creado

Después de todo lo aprendido, nos vamos a encontrar con la necesidad de serializar objetos creados por nosotros mismos.

Como vamos a utilizar la herramienta *pickle*, debemos abrir el fichero en modo binario, añadiendo la letra "b" a cualquier modo.

Debemos tener presente que estamos utilizando los ficheros con codificación binaria, por lo que hay que recordar que no serán legibles a simple vista.

```
class Animal():
    def __init__(self, tipo, modelo,nombre):
        self.tipo = tipo
        self.color = color
        self.nombre = nombre
        self.ruido = False
        self.acelera = False
        self.frena = False

    def ruido(self):
        self.ruido = True

    def correr(self):
        self.acelera = True

    def parar(self):
        self.frena = True

    def estado(self):
        print("Raza:", self.tipo, "\nColor:", self.color, "\nRuido:",
              self.ruido, "\nAcelerando:", self.acelera, "\nFrenando:",
              self.frena)

import pickle

animal1 = Animal("Gato", "Gris", "Quasi")
animal2 = Animal("Pez", "Marron", "Rabbit")
animal3 = Animal("Perro", "Rojo", "Pluto")

animales = [animal1, animal2, animal3]

fichero = open("prueba.txt", "wb")

pickle.dump(animales, fichero)

fichero.close()

del fichero
```

Como hemos dicho anteriormente, el fichero se vuelve completamente ilegible para una persona y solo podrá ser convertido en Python (figura 5.1).

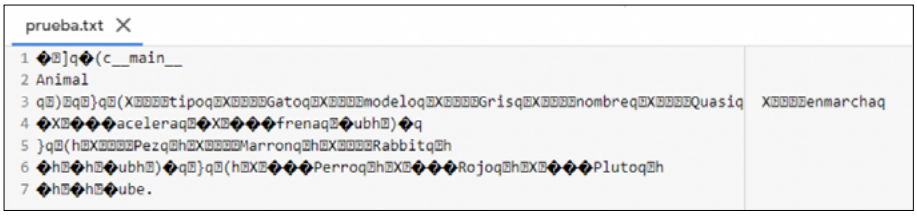

Figura 5.1
Fichero con los datos de la clase Animal serializados.

Resumen

- La serialización es un proceso de codificación de un objeto con el fin de transmitirlo por internet o convertirlo a un formato legible. También, para transmitir objetos entre aplicaciones o entre distintas aperturas de una misma aplicación.

- Además, la serialización se puede utilizar para almacenar una configuración y volver a leerla en una nueva ejecución posterior.

- El formato de fichero CSV, muy relacionado con bases de datos, se utiliza sobre todo para compartir datos entre aplicaciones, que pueden ser leídos o procesados directamente.

- Para almacenar datos en archivos binarios utilizaremos la herramienta *pickle*, ya que nos permite realizar la lectura/escritura de forma sencilla en este formato.

- No será nada operativo acceder a los datos de un documento binario creado con *pickle* desde cualquier otro programa que no esté desarrollado en Python.

Índice