

Programación en Python

1. Introducción a los lenguajes de programación

2. Datos y expresiones

3. Clases

4. Estructuras algorítmicas

5. Estructuras de Datos

6. Tratamiento de errores

7. Entrada/salida

- Escritura de texto con formato. Lectura de números con formato. Ficheros. Lectura de ficheros de texto. Escritura de ficheros de texto.

8. Herencia y Polimorfismo

7.1. Escritura de texto con formato

Hemos usado los `f-strings` para generar texto con formato

- por ejemplo, podemos usar `f-strings` al escribir datos con `print()`
- tienen texto libre y plantillas para datos delimitadas con `{}`
- el formato se indica tras el dato separado por `:`


define un f-string

plantilla para un dato

especificación de formato

Ejemplo

```
print (f"Soy {nombre} de {edad:4d} años")
```



Produce la salida (suponiendo `nombre="Pedro"`, `edad=18`)

```
Soy Pedro de    18 años
```

Notas:

En muchas ocasiones es preciso controlar de forma precisa el formato de los datos que se muestran en pantalla o, como veremos más adelante, los que se escriben en un fichero.

Por ejemplo, puede ser preciso escribir datos en columnas, para una visualización más cómoda. Ejemplo:

```
Santander
Datos meteorológicos promedio
```

```
Mes Máx / Mín Lluvias
Septiembre 22° / 15° 9 días
Octubre 19° / 13° 11 días
Noviembre 16° / 10° 12 días
Diciembre 13° / 8° 11 días
```

a) Datos sin formatear

```
Santander
Datos meteorológicos promedio
```

```
Mes           Máx / Mín      Lluvias
Septiembre    22° / 15°      9 días
Octubre       19° / 13°     11 días
Noviembre     16° / 10°     12 días
Diciembre     13° / 8°      11 días
```

b) Los mismos datos, formateados en columnas

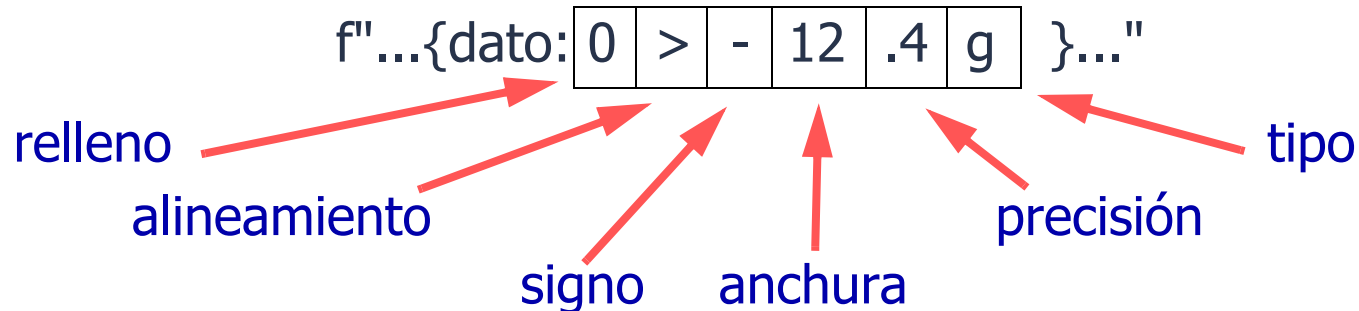
Para obtener datos en columnas podemos indicar en las plantillas para datos de los *f-strings* la anchura mínima de cada dato (en caracteres).

- Esto se hace poniendo una especificación de formato tras el símbolo ":" en la plantilla para datos.
- Como en el ejemplo de la "edad" en la página anterior.

Vamos a estudiar estas especificaciones de formato.

Especificaciones de formato habituales

Se componen de los siguientes campos opcionales, en este orden:



- *relleno*: cualquier carácter, que servirá para rellenar los caracteres del dato; por defecto: " "
- *alineamiento*: carácter que indica cómo alinear el dato

>	derecha	por defecto
<	izquierda	
^	centrado	

Notas:

Cuando se escriben datos con una especificación de anchura mínima, si el valor a escribir ocupa menos que la anchura mínima el sistema debe añadir caracteres de relleno.

- Por defecto son espacios en blanco,
- pero podemos cambiarlos por otros caracteres, por ejemplo ceros.
- También podemos especificar el alineamiento izquierdo, derecho o centrado.

Supongamos que queremos escribir las variables *a*, *b* y *c*, que valen 9, 19 y 1199. Usaremos una anchura mínima de 5 caracteres:

Descripción	Parámetro del print()	Salida
Relleno con espacios en blanco, alineamiento izquierdo	<code>f" {a:<5d} {b:<5d} {c:<5d} "</code>	9 19 1199
Relleno con ceros, alineamiento derecho	<code>f" {a:0>5d} {b:0>5d} {c:0>5d} "</code>	00009 00019 01199

Especificaciones de formato habituales (cont.)

- *tipo*: una letra que indica cómo presentar el dato

Dato	Tipo	Descripción	Comentario
str	s	string	por defecto para strings
int	d	entero decimal	por defecto para enteros
	n	número en el formato local	
float	e o E	real con notación <i>exponencial</i>	
	f o F	real con coma <i>fija</i>	
	g o G	real con formato <i>general</i> (exponencial o coma fija según el tamaño)	por defecto para reales
	n	número general (como la 'g') pero en el formato local	

Notas:

Una de las especificaciones de formato más importantes es el tipo, representado por una letra.

- Permite controlar el uso de notación exponencial o no.
- Permite controlar el uso de formato local o inglés (comas o puntos decimales, etc.).
- Si se utiliza una especificación de un tipo incorrecto, ocurrirá un error.

Ejemplos de representaciones de los diversos tipos:

Dato	Parámetro del print()	Salida
texto = "Hola"	f" {texto:s} "	Hola
	f" {texto:d} "	<i>error</i>
entero = 1000	f" {entero:d} "	1000
	f" {entero:f} "	1000.000000
real = math.pi	f" {real:f} "	3.141593
	f" {real:e} "	3.141593e+00
	f" {real:d} "	<i>error</i>
	f" {real:g} "	3.14159
otro = 1_000_000	f" {otro:g} "	1e+06

Especificaciones de formato habituales (cont.)

- *signo*:

+	siempre, sea + o -	
-	solo si -	por defecto

- *anchura*: un número entero, que indica la anchura mínima total, incluyendo punto decimal, signo, exponente si lo hay, etc.
- *.precisión*: un punto seguido de un número entero, que indica
 - el número de decimales para números reales: tipos fijo (f, F) y exponencial (e, E)
 - o el número de dígitos significativos: tipos general (g, G) y número (n)

Notas:

La especificación de signo permite poner el signo "+" a los números positivos.

- Como es obvio, los números negativos siempre tendrán el signo "-".

La especificación de anchura mínima se expresa en caracteres.

- Es la anchura total, incluyendo los posibles signo, punto decimal y exponente.
- Si el valor ocupa más caracteres, se usan todos los necesarios.
 - Por ello hay que cuidar el cálculo de esta anchura y ser generosos con ella.

Según el tipo, la precisión puede indicar el número de decimales o el número de cifras significativas (incluyendo la parte entera y la decimal).

Ejemplos de especificación de anchura y precisión

Para números puede ponerse la especificación de anchura mínima y (si corresponde) el número de decimales; ejemplos con la constante `math.pi` y el valor `i=18`

Dato	Parámetro del print()	Salida
float	<code>f"Pi= {math.pi:4.0f}"</code>	Pi= 3
	<code>f"Pi= {math.pi:4.2f}"</code>	Pi= 3.14
	<code>f"Pi= {math.pi:12.4f}"</code>	Pi= 3.1416
	<code>f"Pi= {math.pi:<12.4f};"</code>	Pi= 3.1416 ;
	<code>f"Pi= {math.pi:12.8f}"</code>	Pi= 3.14159265
int	<code>f"I= {i:8d}"</code>	I= 18
	<code>f"I= {i:+4d}"</code>	I= +18
	<code>f"I= {i:04d}"</code>	I= 0018

Ejemplos de especificación de tipos

En los formatos `g` y `n` la precisión no indica el número de decimales, sino el número de dígitos significativos

Ejemplos con la constante `math.pi`

Dato	Parámetro del print()	Salida
float	<code>f"Pi= {math.pi:10.2e}"</code>	Pi= 3.14e+00
	<code>f"Pi= {math.pi:10.2E}"</code>	Pi= 3.14E+00
	<code>f"Pi= {math.pi:10.2g}"</code>	Pi= 3.1
	<code>f"Pi= {math.pi:10.4g}"</code>	Pi= 3.142
	<code>f"Pi= {math.pi:10.6g}"</code>	Pi= 3.14159
	<code>f"Pi= {math.pi:10.9g}"</code>	Pi= 3.14159265

Notas:

En los ejemplos con el tipo exponencial podemos ver que es posible controlar si la letra del exponente se pone en minúscula o mayúscula.

En los ejemplos con el tipo general podemos ver que la precisión indica el número de cifras significativas, sumando la parte entera y la decimal.

- Por eso en el primer ejemplo, con precisión .2 solo aparece un decimal y una cifra entera.
- En el último ejemplo se especifica una precisión de 9, que se contabiliza como una cifra entera y 8 decimales.

Uso de formatos locales (español)

Con el formato `n` es posible definir el uso de localismos, como la "*coma*" decimal y el separador de *miles* en español:

```
import locale
locale.setlocale(locale.LC_ALL, "es_ES.UTF-8")
```

Ejemplos de formateo de números en español, con la constante `math.pi` y el valor `j=18000`

Dato	Parámetro del print()	Salida
float	<code>f"Pi= {math.pi:10.4n}"</code>	Pi= 3,142
	<code>f"Pi= {math.pi:10.6n}"</code>	Pi= 3,14159
	<code>f"Pi*1000= {math.pi*1000:10.6n}"</code>	Pi*1000= 3.141,59
int	<code>f"I= {j:6n}"</code>	I= 18.000
	<code>f"I= {j:8n}"</code>	I= 18.000

Notas:

Es posible indicar que se desean utilizar números en formatos locales diferentes del inglés.

- En inglés se usa el "." como separador decimal y la "," para separación de miles. Por ejemplo: 1,234.25
- En español los usamos justo al revés: El ejemplo sería: 1.234,25

El localismo se especifica con la función `locale.setlocale()`, que lleva dos parámetros:

- El primer parámetro es una constante que indica a qué elementos (números, moneda, fechas, etc.) queremos aplicar los localismos.
 - `locale.LC_ALL` indica que a todos.
- El segundo parámetro es un texto que indica el país del formato local y la codificación de caracteres.
 - `"es_ES.UTF-8"` indica español de España y la codificación habitual UTF-8, que permite vocales, acentuadas y ñ.
 - `"en_US.UTF-8"` indica inglés americano y la codificación UTF-8.

7.2 Lectura de números con formato

Habitualmente las funciones de entrada, como `input()`, leen un string y luego lo convertimos a número real o entero con `float()` o `int()`

- Estas conversiones solo funcionan para números en notación inglesa

Para hacer conversión de números en notación española, después de especificar los localismos con `locale.setlocale()`, podemos usar

Función	Descripción
<code>locale.atof(s: str) -> float</code>	Convierte el string <code>s</code> a número real
<code>locale.atoi(s: str) -> int</code>	Convierte el string <code>s</code> a número entero

Ejemplos

Código	Resultado
<code>locale.atof("12.345,56")</code>	12345.56
<code>locale.atoi("10.000")</code>	10000

Notas:

Ejemplo completo para leer un número real y uno entero, usando notación española:

```
import locale

def main():
    """
    Ejemplo que lee del teclado un número entero y uno real
    """
    # Establecer el idioma local
    locale.setlocale(locale.LC_ALL, "es_ES.UTF-8")

    # Leer los números del teclado
    texto_entero: str = input("Introduce un número entero en español: ")
    texto_real: str = input("Introduce un número real en español: ")

    # Convertir los textos a números
    entero: int = locale.atoi(texto_entero)
    real: float = locale.atof(texto_real)

    # Mostrar los resultados
    print(f"|{entero:d}|{real:f}|")
```


7.3. Ficheros

Fichero:

- es una secuencia de bytes en un dispositivo de almacenamiento: disco duro, DVD, memoria USB, ...
- se puede leer y/o escribir
- se identifica mediante un nombre
 - absoluto (*pathname*)
 /home/pepe/documentos/un_fichero
 - o relativo a la carpeta del proyecto
 un_fichero

Tipos de ficheros:

- ***programas***: contienen instrucciones
- ***datos***: contienen información, como números (enteros o reales), secuencias de caracteres, ...

Notas:

El fichero es la unidad de almacenamiento de información en la memoria secundaria del computador, es decir en el disco duro y otras unidades como pen-drives, ...

- Las variables del programa se almacenan en la memoria principal del computador, que es rápida pero *volátil*. Es decir, al apagar el computador o finalizarse el programa, su contenido desaparece.
- A diferencia de ellas, la información contenida en ficheros es *persistente* o no volátil: permanece almacenada aunque el programa se acabe o el computador se apague.
- Por ello es tan importante aprender a leer y escribir esta información desde programas Python.

Cada fichero del disco duro recibe un nombre que lo identifica.

- Este nombre puede ser una ruta absoluta, que comienza en la raíz del árbol de directorios y contiene los nombres de todos los directorios en los que está almacenado el fichero, separados por `"/"`.
- O puede ser una ruta relativa, que comienza en el directorio actual o de trabajo. Esto es lo más habitual.

Ficheros de texto y binarios

Tipos de ficheros de datos:

- **de bytes** (binarios): pensados para ser leídos por un programa
- **de caracteres** (de texto): pueden ser leídos y/o creados por una persona

Fichero binario

0	00000000	Un número entero: 14
1	00000000	
2	00000000	
3	00001110	
4	00000000	Otro número entero: 33
5	00000000	
6	00000000	
7	00100001	
...	...	

Un número entero: 14

Un texto: "hola"

Fichero de texto

0	00110001	'1' (código ASCII 0x31)
1	00110100	'4' (código ASCII 0x34)
2	01101000	'h' (código ASCII 0x68)
3	01101111	'o' (código ASCII 0x6F)
4	01101100	'l' (código ASCII 0x6C)
5	01100001	'a' (código ASCII 0x61)
...	...	

Caracteres que vemos

Notas:

Los ficheros se clasifican en dos categorías:

- *Binarios*: pensados para las máquinas. Los humanos no los entendemos bien. Son eficientes porque ocupan solo el espacio justo y el computador los interpreta más rápidamente.
 - Los ficheros binarios contienen información en forma de números binarios (ceros y unos).
- *De texto*: pensados para los humanos. Los podemos ver e incluso editar con un editor de texto. Son menos eficientes que los ficheros binarios.
 - Los ficheros de texto contienen caracteres con símbolos como letras, números y signos de puntuación. Lógicamente los caracteres se guardan en el computador como números, pero siempre según tablas de codificación fáciles de interpretar y representar en una pantalla o en un papel.

En este capítulo veremos los ficheros de texto, ya que suelen resultar más útiles pues los podemos entender.

7.4. Lectura de ficheros de texto

La operación `open` abre un fichero y retorna un objeto para usarlo

```
fich = open("datos.txt", "r")
```

nombre del fichero

modo (read)

Sobre el objeto fichero `fich` disponemos entre otras de las operaciones

- `read()`: lee el fichero completo y lo retorna como string
- `readline()`: lee una línea del fichero y la retorna como string
 - se empieza por la primera línea
 - cada vez que lo invocamos obtenemos la línea siguiente
 - cuando el fichero se acaba obtenemos un string vacío: ""
 - cada línea leída (aunque esté vacía) conserva el salto de línea (carácter `\n`) como último carácter
 - después de leer la línea podemos partirla en trozos con `split()` y convertir alguno de ellos a número con `float()` o `int()`

Notas:

Para poder usar un fichero desde un programa es necesario *abrirlo*.

- Cuando acabemos de usarlo habrá que hacer el proceso contrario: *cerrarlo*.

Para abrir un fichero usamos la función `open()`.

- Para cerrarlo podemos usar la función `close()`, aunque como veremos más adelante es más habitual usar la instrucción **with**, que permitirá que el cierre sea automático.

La función `open()` toma como parámetros el nombre del fichero y el modo en el que se quiere usar (leer, escribir, añadir, ...) y retorna un objeto que representa un fichero abierto.

- Sobre el objeto fichero retornado podremos invocar funciones de lectura tales como `read()` o `readline()` que retornan un string.

Una vez leída una línea o el texto completo del string, podemos manipularlos con las funciones de manipulación de strings vistas en capítulos anteriores.

Ejemplos de lectura

Leer todo el fichero y mostrarlo

```
fich = open("datos.txt", "r")  
contenido: str = fich.read()  
print(contenido)
```

Leer cada línea del fichero y mostrarla. Lo haremos iterando en un bucle

```
fich = open("datos.txt", "r")  
for linea in fich:  
    print(linea, end="") # línea ya tiene el \n
```

Notas:

Se muestran dos ejemplos de lectura de ficheros:

- En el primer ejemplo se lee el contenido completo del fichero, con `read()`. Con ello se obtiene un string, que luego mostramos en pantalla a modo de prueba.
- En el segundo ejemplo se lee el contenido completo del fichero, pero línea a línea. Como vemos, es posible iterar con un bucle `for` sobre el fichero. La variable de control del bucle, `línea` en este caso, es un string que va tomando sucesivamente los valores de cada una de las líneas del fichero, como si se hubiesen leído con `readline()`. Estos strings contienen al final el salto de línea que aparecía en el fichero al final de la línea. Por ello, al mostrarlos en pantalla con un `print` lo hacemos con la opción `end=""`, para que no se añada un segundo salto de línea.
- Por ejemplo, si el fichero contuviese las líneas:

```
Esta es la primera línea
Esta es la segunda
Y esta es la tercera
```

- La variable `línea` valdría:
 - En la primera iteración del bucle: `"Esta es la primera línea\n"`
 - En la segunda iteración del bucle: `"Esta es la segunda\n"`
 - En la tercera iteración del bucle: `"Y esta es la tercera\n"`

Cerrar el fichero

Una vez acabado de usar debemos cerrar el fichero

- `close()`: cierra el fichero y libera los recursos que usa
- ```
fich = open("datos.txt", "r")
for linea in fich:
 print(linea, end="") # línea ya tiene el \n
fich.close()
```

Como esto es muy importante hay una orden especial (`with`) para hacer el `close()` automático, incluso aunque salte una excepción:

```
with open("datos.txt", "r") as fich:
 for linea in fich:
 print(linea, end="")
```

Ya no ponemos el `close()`, pues se hace automático

## Notas:

Como comentábamos antes, en lugar de usar la función `close()` para cerrar un fichero es más habitual y seguro utilizar la instrucción **with**.

Esta instrucción se usa combinadamente con la función `open()` al abrir el fichero. Al final de la instrucción, tras la palabra **as**, se pone el nombre del objeto que representará el fichero abierto.

- Cuando se acabe la instrucción **with** el fichero se cerrará automáticamente, incluso aunque haya ocurrido alguna excepción.
- Recordemos que el final de la instrucción **with**, al igual que para cualquier otra instrucción, se marca mediante el sangrado.

En definitiva, recomendamos siempre abrir el fichero con la instrucción **with**, al ser el funcionamiento mucho más fiable.

- Además, tal como se explica a continuación, incluiremos siempre la instrucción **with** dentro de un bloque **try-catch** para tratar las excepciones que puedan producirse.

# Excepciones y ficheros

---

Las operaciones con ficheros pueden fallar con una excepción del tipo `IOError`

- por ejemplo, si abrimos un fichero que no existe para leerlo
  - esto ocurre típicamente cuando nos equivocamos al dar el nombre del fichero
- otro ejemplo: si abrimos un fichero para escribirlo y está protegido frente a escritura

Por tanto hay que protegerse de esta excepción

En el siguiente ejemplo mostramos cómo hacerlo

# Ejemplo: leer un fichero con palabras seguidas de números

---

Para el fichero (`mis_datos.txt`):

```
azul 1.0 3.5 7.7
rojo 2
verde 10.0 11.1
```

- Se desea obtener la siguiente salida por consola:

```
Palabra: azul
Número: 1.0
Número: 3.5
Número: 7.7
Palabra: rojo
Número: 2.0
Palabra: verde
Número: 10.0
Número: 11.1
```

## Notas:

En este ejemplo disponemos de un fichero con varias líneas. Cada línea contiene una palabra y uno o varios números reales o enteros.

Pretendemos que el programa lea estas líneas y las interprete, separando las palabras y los números. Además debe convertir cada texto que contenga un número a su representación como número real, con el que se podrían hacer operaciones aritméticas.

Para comprobar el correcto funcionamiento mostraremos en pantalla los datos obtenidos (palabras y números).

# Ejemplo (cont.)

---

```
nombre_fichero: str = "mis_datos.txt"
try:
 with open(nombre_fichero, "r") as fich:
 for linea in fich:
 # separamos por los espacios en blanco
 palabras=linea.split(" ")
 # Mostramos la palabra
 print(f"Palabra: {palabras[0]}")
 # Mostramos los números, de uno en uno
 for pal in palabras[1:]:
 num: float = float(pal)
 print(f"Número: {num}")
except IOError:
 print(f"El fichero {nombre_fichero} no existe")
```

# Notas:

Se muestra solo una secuencia de instrucciones, que lógicamente habría que incluir dentro de un `main()` o de otra función.

Podemos observar el patrón de uso de la instrucción `with` metida dentro de un bloque `try-catch`, tal como habíamos visto anteriormente.

- Tras abrir el fichero recorreremos sus líneas una a una, con la variable de control de bucle `linea`.
- Separamos la línea en sus palabras usando la función `split()`. Con ello obtenemos una lista de strings. Por ejemplo, para la primera línea obtendríamos:

```
palabras = ["azul", "1.0", "3.5", "7.7"]
```

- La casilla cero de esta lista es la palabra por la que comienza la línea.
- Para tratar el resto de las palabras (los números) escribimos un segundo bucle que recorre todos los elementos de la lista `palabras`, excepto el primero. Convertimos al tipo `float` cada una de estas palabras, para obtener el correspondiente número real.

Como tratamiento de la excepción `IOError` ponemos un mensaje de error en pantalla.

# 7.5 Escritura de ficheros de texto

La operación `open` abre un fichero y retorna un objeto para usarlo

```
fich = open("datos.txt", "w")
```

nombre del fichero

modo (write)

Estos son los principales modos para ficheros de texto:

| Modo |           | Descripción                                             |
|------|-----------|---------------------------------------------------------|
| "r"  | read      | leer (valor por defecto)                                |
| "w"  | write     | escribir, creando el fichero aunque exista              |
| "a"  | append    | escribe añadiendo al final, aunque lo crea si no existe |
| "x"  | exclusive | escribir creando el fichero, pero falla si ya existe    |

Sobre el objeto fichero `fich` disponemos entre otras de la operación

- `write()`: escribe un string en el fichero



# Notas:

Para escribir ficheros de texto debemos abrirlos con `open()`, pero usando uno de estos tres modos:

- `w` (*write*): escribir. En este modo, si el fichero no existe se crea nuevo. Si ya existía se borra y se parte de uno nuevo vacío.
- `a` (*append*): añadir. En este modo si el fichero no existe se crea uno nuevo vacío. Pero si ya existía, en lugar de borrarlo como antes, los datos que se escriban se añadirán al final del fichero.
- `x` (*xclusive*): exclusivamente crear. En este modo si el fichero existe se produce un fallo. Si no existe, se crea nuevo vacío.

Una vez abierto el fichero para escritura podemos utilizar la función `write()`, que escribe un string al final del fichero.

- Si deseamos escribir datos con formato podremos usar *f-strings* con especificaciones de formato, igual que hacíamos con la función `print()`.

# Ejemplo: función que escribe tres datos en un fichero de texto

---

```
def escribe_fichero(nombre_fichero: str, palabra: str,
 i: int, x_0: float):
 """
 Escribe un string y dos números en un fichero de texto
 """

 try:
 with open(nombre_fichero, "w") as fich:
 fich.write(f"Palabra: {palabra}\n")
 fich.write(f"Entero: {i}, Real: {x_0}\n")
 except IOError as err:
 print(f"Problema con el fichero {nombre_fichero}.", err)
```

# Ejemplo: función que escribe en un fichero de texto (cont.)

---

Ejemplo de fichero generado con

```
escribe_fichero("datos.txt", "Pedro", 12, 34.56)
```

datos.txt:

|                                           |
|-------------------------------------------|
| Palabra: Pedro<br>Entero: 12, Real: 34.56 |
|-------------------------------------------|

## Notas:

En el ejemplo anterior se crea una función para escribir en un fichero cuyo nombre se pasa como primer parámetro. Queremos escribir un string, un número entero y un número real, que también se pasan como parámetros.

- El fichero se abre en modo "w" (escritura) usando el patrón habitual con la instrucción **with** metida en un bloque **try-catch**.
  - Si ocurre la excepción `IOError` se pone un mensaje en pantalla.
- Una vez abierto el fichero, se escriben en él *f-strings* usando la función `write()`.

Finalmente se invoca a esta función con datos concretos para escribir en un fichero llamado "datos.txt".