



Índice

UD 01: Gestión de información almacenada en ficheros.....	3
1.1. Introducción.....	3
1.2. Gestión de ficheros y directorios con clases.....	3
1.3. Ventajas e inconvenientes de las distintas formas de acceso.....	4
1.3.1. Modos “r”, “w” y “a”.....	5
Modo r (Read - Lectura).....	5
Modo w (Write - Escritura).....	5
Modo a (Append - Añadir).....	5
Resumen:.....	6
1.4. Recuperar/almacenar información de/en ficheros (JSON).....	6
1.5. Recuperar/almacenar información de/en ficheros (CSV).....	8
Ventajas del formato CSV:.....	8
1.6. Conversiones entre Formatos de Ficheros.....	10
1.7. Gestión de Excepciones.....	10
1.8. Pruebas y Documentación.....	11
1.8.1. Introducción a `unittest`.....	11
1.8.2. Estructura básica de `unittest`.....	11
1.8.3. Métodos de Aserción en `unittest`.....	12
1.8.4. Ejemplo Ampliado con `unittest`.....	12
1.8.5. Ventajas de `unittest`:.....	14
1.9. Conclusión.....	14
Anexo 1: El constructor en las clases de Python.....	14
¿Qué es self en Python?.....	15
Anexo 2: La estructura `with ... as` en Python.....	15
Anexo 3: El módulo estándar “shutil” de Python.....	16
Anexo 4: Configurar Visual Studio Code para trabajar con Python.....	17
Anexo 5: ¿Cómo funciona `__name__` en Python?.....	17
1. Ejemplo de archivo ejecutado directamente.....	17
2. Ejemplo de archivo importado como un módulo.....	18

UD 01: Gestión de información almacenada en ficheros

1.1. Introducción

En este tema abordaremos cómo gestionar ficheros y directorios utilizando clases en Python. Exploraremos diferentes formas de acceso, cómo almacenar y recuperar información, cómo convertir entre distintos formatos de ficheros, y cómo manejar excepciones. Además, desarrollaremos ejemplos prácticos para aplicar estos conceptos.

1.2. Gestión de ficheros y directorios con clases

Python ofrece los módulos `os` y `pathlib` para la manipulación de ficheros y directorios. Ambos módulos permiten navegar y manipular el sistema de ficheros. `pathlib` es más moderno y orientado a objetos.

Al implementar clases, encapsulamos funcionalidades y logramos un diseño más limpio y reutilizable. Crear clases nos permite agrupar métodos relacionados con la manipulación de ficheros, lo que facilita su mantenimiento.

Ejemplo Práctico

```
from pathlib import Path

class FileManager:
    def __init__(self, path):
        self.path = Path(path)

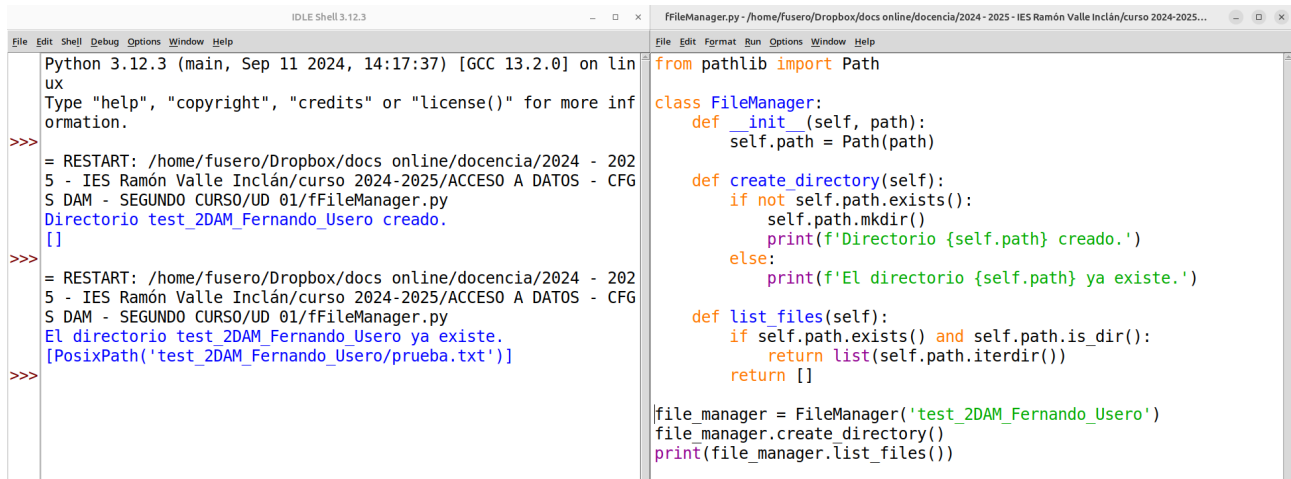
    def create_directory(self):
        if not self.path.exists():
            self.path.mkdir()
            print(f'Directorio {self.path} creado.')
        else:
            print(f'El directorio {self.path} ya existe.')

    def list_files(self):
        if self.path.exists() and self.path.is_dir():
            return list(self.path.iterdir())
        return []

    def delete_directory(self):
        if self.path.exists() and self.path.is_dir():
            self.path.rmdir()
            print(f'Directorio {self.path} eliminado.')

# Uso
file_manager = FileManager('test_directory')
file_manager.create_directory()
```

```
print(file_manager.list_files())
file_manager.delete_directory()
```



The screenshot shows a code editor with two windows. The left window, titled 'IDLE Shell 3.12.3', displays the output of a Python script. The right window, titled 'FileManager.py', shows the source code of the script. The code defines a class 'FileManager' with methods for creating, listing, and deleting directories. The output in the shell shows the successful creation and listing of a directory.

```
Python 3.12.3 (main, Sep 11 2024, 14:17:37) [GCC 13.2.0] on lin
ux
Type "help", "copyright", "credits" or "license()" for more inf
ormation.
>>>
= RESTART: /home/fusero/Dropbox/docs online/docencia/2024 - 202
5 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFG
S DAM - SEGUNDO CURSO/UD 01/fFileManager.py
Directorio test_2DAM_Fernando_Usero creado.
[]
>>>
= RESTART: /home/fusero/Dropbox/docs online/docencia/2024 - 202
5 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFG
S DAM - SEGUNDO CURSO/UD 01/fFileManager.py
El directorio test_2DAM_Fernando_Usero ya existe.
[PosixPath('test_2DAM_Fernando_Usero/prueba.txt')]
>>>
```

```
from pathlib import Path

class FileManager:
    def __init__(self, path):
        self.path = Path(path)

    def create_directory(self):
        if not self.path.exists():
            self.path.mkdir()
            print(f'Directorio {self.path} creado.')
        else:
            print(f'El directorio {self.path} ya existe.')

    def list_files(self):
        if self.path.exists() and self.path.is_dir():
            return list(self.path.iterdir())
        return []

file_manager = FileManager('test_2DAM_Fernando_Usero')
file_manager.create_directory()
print(file_manager.list_files())
```

Actividad 1 de clase: añade a la clase anterior un método que sirva para borrar directorios. Pide al usuario que pulse una tecla entre la creación y el borrado del directorio.

Adjunta tres capturas de pantalla:

- (1) Una en la que se vea que la carpeta existe
- (2) Una segunda en la que se vea la pantalla dividida como habitualmente: a la izquierda la ejecución y a la derecha el código
- (3) Una en la que sea vea que la carpeta no existe

1.3. Ventajas e inconvenientes de las distintas formas de acceso

Al igual que en la mayoría de lenguajes de programación, existen diferentes modos para acceder a ficheros:

- Modo texto ('r', 'w', 'a'): Ideal para datos que se pueden representar como texto plano.
- Modo binario ('rb', 'wb'): Utilizado para archivos no textuales como imágenes o ficheros comprimidos.

Ventajas

- Texto: Fácil de manipular y leer. Útil para archivos como '.txt' o '.csv'.
- Binario: Mayor flexibilidad al trabajar con datos no textuales, pero más complejo de manipular.

Inconvenientes

- Texto: Limitado a caracteres legibles, problemas de codificación.

- Binario: Más propenso a errores si no se entiende el formato.

1.3.1. Modos “r”, “w” y “a”

Estos modos definen cómo vamos a interactuar con el fichero una vez abierto, ya sea para leer, escribir o añadir información. Aquí les explico en qué consiste cada uno:

Modo r (Read - Lectura)

Este es el modo por defecto para abrir un fichero en Python. El modo **r** se utiliza para **leer** el contenido del fichero.

- **Características:**
 - El fichero **debe existir**; si intentas abrir un fichero que no existe en este modo, Python lanzará un error (`FileNotFoundError`).
 - Solo se permite **leer** el contenido del fichero, no modificarlo.

Ejemplo:

```
with open('archivo.txt', 'r') as file:
    contenido = file.read() # Lee todo el contenido del fichero
```

Modo w (Write - Escritura)

El modo **w** se utiliza para **escribir** en un fichero.

- **Características:**
 - Si el fichero **no existe**, Python lo creará automáticamente.
 - Si el fichero **ya existe**, todo su contenido será **sobrescrito**, es decir, se borrará lo que estaba antes y se empezará desde cero.

Ejemplo:

```
with open('archivo.txt', 'w') as file:
    file.write('Hola, mundo!') # Escribe en el fichero (sobrescribiendo cualquier contenido previo)
```

Modo a (Append - Añadir)

El modo **a** se utiliza para **añadir** información al final de un fichero ya existente, sin eliminar su contenido actual.

- **Características:**
 - Si el fichero **no existe**, también se creará.
 - Si el fichero **ya existe**, el nuevo contenido se añadirá **al final** del fichero sin eliminar lo que ya había.

Ejemplo:

```
with open('archivo.txt', 'a') as file:
    file.write('\nAñadiendo una nueva línea al fichero.')
```

Resumen:

- **r**: Solo lectura. El fichero debe existir.
- **w**: Escritura. Si el fichero existe, se sobrescribe; si no, se crea uno nuevo.
- **a**: Añadir al final. Si el fichero existe, se añade contenido al final; si no, se crea.

Es importante que recuerden usar el contexto `with` para abrir ficheros, ya que asegura que el fichero se cierre automáticamente cuando terminan de trabajar con él, evitando posibles errores o pérdidas de datos.

Ejemplo

[Nota: si tienes dudas al respecto de la estructura `with...as` en Python, consulta la sección Anexos de este tema]

```
class FileHandler:
    def read_file(self, file_path, mode='r'):
        try:
            with open(file_path, mode) as f:
                content = f.read()
            return content
        except Exception as e:
            print(f"Error leyendo el archivo: {e}")

    def write_file(self, file_path, content, mode='w'):
        try:
            with open(file_path, mode) as f:
                f.write(content)
        except Exception as e:
            print(f"Error escribiendo en el archivo: {e}")
```

Actividad 2 de clase: Adjunta la captura de pantalla habitual de un programa en Python que haga uso de esta clase. El programa debe usar un archivo que se llame “12345678.txt”, siendo el nombre del fichero tu DNI. El programa debe escribir en dicho archivo una cadena de texto que sea tu fecha de nacimiento y debe leerla luego del fichero. Por último el programa ha de mostrar por pantalla lo que haya leído del fichero.

Adjunta además una **captura de pantalla adicional** que se corresponda con una terminal que muestre el contenido del fichero de texto.

1.4. Recuperar/almacenar información de/en ficheros (JSON)

El acceso y recuperación de datos almacenados puede ser gestionado mediante clases que encapsulan las operaciones necesarias, como leer líneas, parsear datos en formatos específicos (JSON, CSV, etc.).

- Ficheros estructurados (JSON, CSV): Los ficheros JSON y CSV son comunes para almacenar datos estructurados.
- Ficheros binarios: Para datos no estructurados como imágenes o ficheros comprimidos.

Ejemplo Práctico: Leer un archivo JSON

```
import json

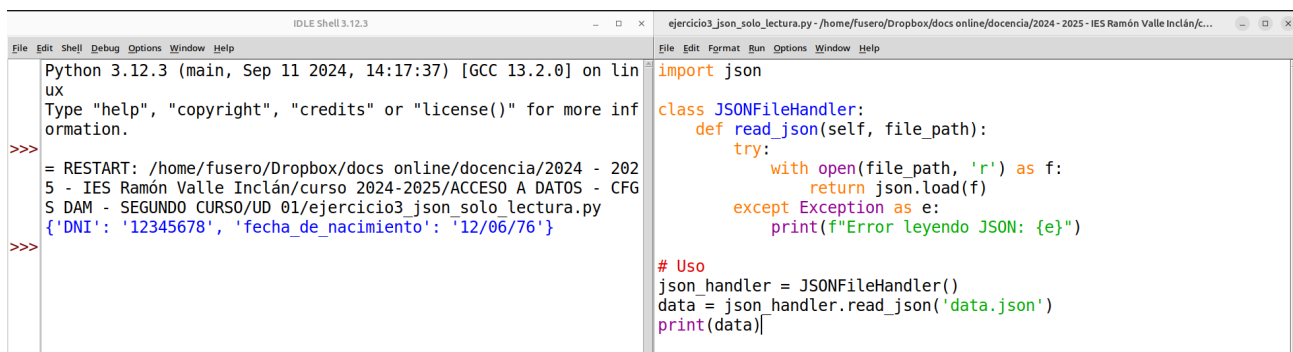
class JSONFileHandler:
    def read_json(self, file_path):
        try:
            with open(file_path, 'r') as f:
                return json.load(f)
        except Exception as e:
            print(f"Error leyendo JSON: {e}")

# Uso
json_handler = JSONFileHandler()
data = json_handler.read_json('data.json')
print(data)
```

Creamos un archivo denominado “data.json” con este contenido:

```
{
  "DNI": "12345678",
  "fecha_de_nacimiento": "12/06/76"
}
```

y si lo ejecutamos obtenemos esto:



```
Python 3.12.3 (main, Sep 11 2024, 14:17:37) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: /home/fusero/Dropbox/docs online/docencia/2024 - 2025 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFGS DAM - SEGUNDO CURSO/UD 01/ejercicio3 json solo lectura.py
>>> {'DNI': '12345678', 'fecha_de_nacimiento': '12/06/76'}
```

```
import json

class JSONFileHandler:
    def read_json(self, file_path):
        try:
            with open(file_path, 'r') as f:
                return json.load(f)
        except Exception as e:
            print(f"Error leyendo JSON: {e}")

# Uso
json_handler = JSONFileHandler()
data = json_handler.read_json('data.json')
print(data)
```

Actividad 3 de clase: amplía la clase anterior para escribir un JSON en el fichero de data.json. Dicho JSON ha de tener dos claves: una para tu DNI (por ejemplo 12345678) y otra para tu fecha

de nacimiento (por ejemplo 12/06/76). Usa en Python un dato tipo diccionario para construir el JSON.

1.5. Recuperar/almacenar información de/en ficheros (CSV)

Los ficheros CSV (Comma-Separated Values) son un formato simple y ampliamente utilizado para almacenar datos tabulares, como hojas de cálculo o bases de datos, en un archivo de texto. Cada línea del archivo CSV representa una fila de la tabla, y los valores de cada columna están separados por comas (aunque también se pueden utilizar otros delimitadores como punto y coma).

Este formato es muy útil para intercambiar datos entre diferentes aplicaciones, ya que es legible tanto por humanos como por máquinas. Es compatible con muchas herramientas de software, como Microsoft Excel, bases de datos y lenguajes de programación, incluyendo Python.

Ventajas del formato CSV:

- **Simplicidad:** Es un formato de texto fácil de leer y escribir.
- **Ligero:** Ocupa poco espacio en comparación con otros formatos, como Excel o JSON.
- **Compatibilidad:** Es reconocido por una gran variedad de herramientas y lenguajes.

En Python, se pueden manipular archivos CSV de manera eficiente utilizando el módulo incorporado CSV, que permite tanto la lectura como la escritura de estos archivos de forma sencilla.

Ejemplo Práctico: Escribir /almacenar en un archivo CSV

```
import csv

class CSVFileHandler:
    def read_csv(self, file_path):
        try:
            with open(file_path, mode='r', newline='') as f:
                reader = csv.DictReader(f) # Creamos el objeto reader para leer las filas como
diccionarios
                rows = [] # Lista vacía para almacenar las filas
                for row in reader: # Recorremos cada fila en el archivo CSV
                    rows.append(row) # Añadimos cada fila (un diccionario) a la lista
                return rows # Devolvemos la lista con todas las filas
        except Exception as e:
            print(f"Error leyendo el archivo CSV: {e}")

    def write_csv(self, file_path, data, fieldnames):
        try:
            with open(file_path, mode='w', newline='') as f:
                writer = csv.DictWriter(f, fieldnames=fieldnames) # Creamos un objeto writer para
escribir en CSV
                writer.writeheader() # Escribimos la primera fila que contiene los nombres de las
columnas (encabezado)
                writer.writerow(data) # Escribimos los datos en una fila en el archivo CSV
        except Exception as e:
            print(f"Error escribiendo en el archivo CSV: {e}")
```



```
# Ejemplo de uso
csv_handler = CSVFileHandler()

# Definir la ruta del archivo CSV
file_path = 'data.csv'

# Datos para escribir en el archivo CSV (día, mes, año, DNI)
data = {
    "Día": "12",
    "Mes": "Junio",
    "Año": "1976",
    "DNI": "12345678"
}

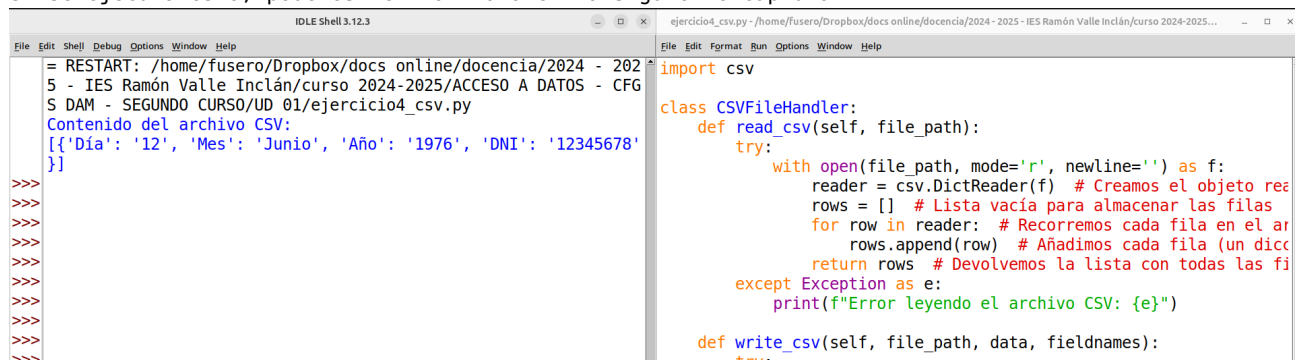
# Definir los nombres de las columnas
fieldnames = ["Día", "Mes", "Año", "DNI"]

# Escribir los datos en el archivo CSV
csv_handler.write_csv(file_path, data, fieldnames)

# Leer los datos del archivo CSV
contenido_csv = csv_handler.read_csv(file_path)

# Mostrar el contenido del archivo CSV por pantalla
print("Contenido del archivo CSV:")
print(contenido_csv)
```

Si se ejecuta esto, podemos ver la traza en la siguiente captura:



The screenshot shows two windows. The left window is a terminal titled 'IDLE Shell 3.12.3' showing the execution of a Python script. The output is: 'RESTART: /home/fusero/Dropbox/docs online/docencia/2024 - 2025 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFG S DAM - SEGUNDO CURSO/UD 01/ejercicio4_csv.py', 'Contenido del archivo CSV:', and a list of dictionaries: [{'Día': '12', 'Mes': 'Junio', 'Año': '1976', 'DNI': '12345678'}]. The right window is a code editor titled 'ejercicio4_csv.py' showing the Python code for the CSVFileHandler class, including methods for reading and writing CSV files.

y si vemos el contenido del fichero csv, vemos que la información efectivamente se ha guardado en dicho formato.



1.6. Conversiones entre Formatos de Ficheros

Una funcionalidad importante es la conversión de formatos, como transformar un archivo CSV a JSON.

Ejemplo Práctico: Convertir CSV a JSON

```
import csv
import json

class FileConverter:
    def csv_to_json(self, csv_file, json_file):
        try:
            with open(csv_file, 'r') as f:
                reader = csv.DictReader(f)
                rows = list(reader)
            with open(json_file, 'w') as f:
                json.dump(rows, f)
            print(f'Conversión de {csv_file} a {json_file} completada.')
        except Exception as e:
            print(f"Error en la conversión: {e}")

# Uso
converter = FileConverter()
converter.csv_to_json('data.csv', 'data.json')
```

Actividad 4 de clase: extiende la clase anterior para construir un método inverso, que pase de formato json a formato csv. Adjunta tres capturas de pantalla:

- (1) La clásica con división a la izquierda (traza) y a la derecha (código).
- (2) Contenido del fichero JSON
- (3) Contenido del fichero CSV

El fichero JSON de origen ha de ser como el de la actividad 3. El fichero csv resultante ha de tener una primera fila de cabecera con título el nombre de las columnas: DNI y fecha de nacimiento.

1.7. Gestión de Excepciones

Es importante prever y gestionar los errores que pueden surgir durante el manejo de ficheros, como ficheros inexistentes o falta de permisos.

Bloques “try-except”: Permiten capturar y manejar excepciones, evitando que la aplicación se detenga abruptamente.

Ejemplo Práctico

```
class SafeFileHandler:
    def safe_read(self, file_path):
        try:
            with open(file_path, 'r') as f:
                return f.read()
        except FileNotFoundError:
            print("El archivo no existe.")
        except PermissionError:
            print("No tienes permisos para leer este archivo.")
        except Exception as e:
            print(f"Error: {e}")
```

1.8. Pruebas y Documentación

Las pruebas aseguran el correcto funcionamiento de las aplicaciones. Es recomendable usar módulos como `unittest` para estructurar y automatizar las pruebas.

1.8.1. Introducción a `unittest`

El módulo `unittest` es el framework estándar para realizar pruebas unitarias en Python. Este módulo proporciona una manera estructurada y automatizada de verificar que el código funcione correctamente, permitiendo que los desarrolladores creen pruebas específicas para funciones, métodos y clases, y puedan ejecutarlas de manera eficiente.

Características de `unittest`:

- Automatización de pruebas: Puedes ejecutar múltiples pruebas de forma automática.
- Agrupación de pruebas: Las pruebas se agrupan en clases y métodos, lo que facilita su organización.

- Aserciones: Proporciona varios métodos de aserción para verificar que las salidas de los métodos sean las esperadas.
- Configuración y limpieza: Métodos como ``setUp`` y ``tearDown`` permiten ejecutar acciones antes y después de cada prueba.

1.8.2. Estructura básica de ``unittest``

Para crear un test usando ``unittest``, hay que seguir los siguientes pasos:

1. Importar el módulo ``unittest``.
2. Definir una clase que herede de ``unittest.TestCase``.
3. Definir métodos de prueba, cuyos nombres deben empezar con ``test_``.
4. Utilizar métodos de aserción para verificar los resultados.

1.8.3. Métodos de Aserción en ``unittest``

Estos métodos se usan para comparar los resultados obtenidos con los resultados esperados. Algunos de los métodos más utilizados son:

- ``assertEqual(a, b)``: Verifica que ``a == b``.
- ``assertNotEqual(a, b)``: Verifica que ``a != b``.
- ``assertTrue(x)``: Verifica que ``x`` sea verdadero.
- ``assertFalse(x)``: Verifica que ``x`` sea falso.
- ``assertIsNone(x)``: Verifica que ``x`` sea ``None``.
- ``assertRaises(exception, callable, *args, **kwargs)``: Verifica que una excepción específica sea lanzada.

1.8.4. Ejemplo Ampliado con ``unittest``

Veamos cómo podríamos probar la clase ``FileManager`` que creamos anteriormente usando ``unittest``.

```
import unittest
import os

class FileManager:
    def __init__(self, file_path):
        self.file_path = file_path

    def write_file(self, content):
        """Escribe contenido en el archivo. Si no existe, lo crea."""
        with open(self.file_path, 'w') as file:
            file.write(content)

    def read_file(self):
        """Lee y retorna el contenido del archivo."""
        with open(self.file_path, 'r') as file:
```

```
        return file.read()

    def append_to_file(self, content):
        """Agrega contenido al final del archivo."""
        with open(self.file_path, 'a') as file:
            file.write(content)

    def delete_file(self):
        """Elimina el archivo si existe."""
        if os.path.exists(self.file_path):
            os.remove(self.file_path)

# Clase de prueba
class TestFileManager(unittest.TestCase):

    def setUp(self):
        """Se ejecuta antes de cada prueba. Crea un archivo temporal para las pruebas."""
        self.file_manager = FileManager('test_file.txt')
        self.file_manager.write_file('Contenido inicial.\n')

    def tearDown(self):
        """Se ejecuta después de cada prueba. Elimina el archivo de prueba."""
        self.file_manager.delete_file()

    def test_write_file(self):
        """Prueba que el método write_file escribe correctamente."""
        self.file_manager.write_file('Nuevo contenido.')
        contenido = self.file_manager.read_file()
        self.assertEqual(contenido, 'Nuevo contenido.')

    def test_read_file(self):
        """Prueba que read_file devuelve el contenido correcto."""
        contenido = self.file_manager.read_file()
        self.assertEqual(contenido, 'Contenido inicial.\n')

    def test_append_to_file(self):
        """Prueba que append_to_file añade correctamente contenido al final del archivo."""
        self.file_manager.append_to_file('Contenido adicional.\n')
        contenido = self.file_manager.read_file()
        self.assertEqual(contenido, 'Contenido inicial.\nContenido adicional.\n')

    def test_delete_file(self):
        """Prueba que delete_file elimina el archivo correctamente."""
        self.file_manager.delete_file()
        self.assertFalse(os.path.exists('test_file.txt'))

# Ejecución de las pruebas
if __name__ == '__main__':
    unittest.main()
``
```

Explicación del Ejemplo:

1. **setUp**: Este método se ejecuta ****antes**** de cada prueba individual. En este caso, crea un archivo de prueba con contenido inicial para que cada prueba comience en el mismo estado.
2. **tearDown()**: Este método se ejecuta ****después**** de cada prueba individual. Aquí, se elimina el archivo temporal utilizado en la prueba, para que no quede rastro después de la ejecución.

3. Métodos de prueba:

- `test_write_file`**: Verifica que el método `write_file()` escribe correctamente en el archivo.
- `test_read_file`**: Comprueba que `read_file()` retorna el contenido correcto.
- `test_append_to_file`**: Verifica que `append_to_file()` añade contenido al final del archivo.
- `test_delete_file`: Verifica que el archivo es eliminado correctamente con `delete_file()`.

4. `if __name__ == '__main__':`

Esta línea es una comprobación que se utiliza para determinar si el script está siendo ejecutado directamente o si está siendo importado como un módulo en otro script.

- Cuando el archivo se ejecuta directamente (es decir, corres el script desde la línea de comandos o un entorno de ejecución), el valor de la variable especial `__name__` es `'__main__'`.
- Si el archivo es importado como un módulo, el valor de `__name__` será el nombre del archivo (sin extensión).

En este caso, como el archivo se ejecuta directamente, el bloque de código dentro de esta comprobación será ejecutado.

5. `unittest.main()`

Esta función es el motor principal del módulo `unittest` de Python. Hace lo siguiente:

- Descubre y ejecuta automáticamente todas las pruebas que encuentren en las clases que heredan de `unittest.TestCase`.
- En este caso, ejecutará todas las pruebas definidas en la clase `TestFileManager`.
- Al final, muestra un resumen de los resultados, indicando si todas las pruebas pasaron o si hubo errores o fallos.

Resultados de las Pruebas:

Al ejecutar este código, cada prueba será ejecutada de forma automática y se mostrará un informe indicando si todas las pruebas han pasado, o si alguna ha fallado. Un reporte típico en la consola puede verse así:

```
-----  
Ran 4 tests in 0.001s
```

```
OK
```

1.8.5. Ventajas de `unittest`:

- Permite detectar errores rápidamente en el desarrollo.
- Automatiza las pruebas, lo que ahorra tiempo a largo plazo.
- Facilita el mantenimiento del código, ya que cuando se cambian funciones o métodos, puedes verificar fácilmente si todo sigue funcionando correctamente.
- Ofrece reportes detallados sobre los resultados de las pruebas, permitiendo identificar fallos fácilmente.

1.9. Conclusión

Este tema abarca desde la gestión de ficheros y directorios hasta la conversión entre formatos de ficheros, la gestión de excepciones, y la prueba de las aplicaciones desarrolladas. Con Python y las clases, podemos estructurar el acceso a datos de una manera clara y mantenible.

Anexo 1: El constructor en las clases de Python

En Python, como en otros lenguajes orientados a objetos, el método `__init__()` es lo que usamos para inicializar una instancia de una clase, y es equivalente al constructor en Java. La principal diferencia es que, en Python, no es necesario declarar explícitamente el tipo de datos de los atributos o parámetros, lo que hace que el constructor sea más flexible.

Cuando creas una instancia de una clase, el método `__init__()` se ejecuta automáticamente, y en este método puedes establecer el estado inicial de los atributos del objeto. Un aspecto importante en Python es el parámetro **`self`**, que siempre debe ser el primer argumento de cualquier método de instancia dentro de una clase, incluyendo el constructor.

¿Qué es `self` en Python?

El parámetro `self` en Python es análogo a `this` en Java. Se utiliza para referirse a la instancia actual de la clase, permitiendo acceder a sus atributos y métodos. Cada vez que creas un método dentro de una clase, `self` te permite trabajar con los atributos de esa instancia en particular, lo que es crucial para manejar datos de forma dinámica en objetos diferentes.

Por ejemplo, cuando declaras algo como:

```
class Coche:
    def __init__(self, color, marca):
        self.color = color
        self.marca = marca
```

Aquí, `self.color` y `self.marca` son atributos que pertenecen a la instancia del objeto que estás creando. Cuando llamas al constructor pasando argumentos, `self` permite que esos valores se asignen correctamente a esa instancia específica. A diferencia de Java, en Python no necesitas pasar explícitamente `self` al llamar a los métodos; Python lo hace por ti automáticamente.

En resumen, aunque la estructura en Python es muy similar a la de Java, el uso de `self` es un pequeño detalle que asegura que siempre trabajamos con la instancia correcta. Esto permite que Python mantenga su filosofía de ser claro y fácil de leer sin perder las capacidades de programación orientada a objetos que ya conoces de Java.

Anexo 2: La estructura `with ... as` en Python

La estructura `with ... as` en Python se utiliza para gestionar contextos, permitiendo realizar tareas que requieren configuración previa y limpieza posterior, como abrir archivos o gestionar recursos. Esta estructura garantiza que los recursos, como archivos o conexiones, se gestionen adecuadamente, liberando memoria o cerrando el archivo una vez que ya no son necesarios, incluso si ocurre un error durante la ejecución.

Sintaxis básica:

```
```python
with expresión_de_contexto as variable:
 # Bloque de código donde se utiliza 'variable'
```
```

Ejemplo con archivos:

```
```python
with open('archivo.txt', 'r') as archivo:
 contenido = archivo.read()
 print(contenido)
```
```

En este ejemplo:

- `open('archivo.txt', 'r')` abre el archivo en modo lectura.
- El archivo abierto se asigna a la variable `archivo` gracias al `as`.
- Cuando el bloque de código dentro de `with` termina, Python cierra automáticamente el archivo, sin necesidad de llamar manualmente a `archivo.close()`, lo que previene errores como olvidar cerrarlo.

Ventajas del uso de `with`:

1. **Gestión automática de recursos**: El recurso se cierra o libera automáticamente al salir del bloque.
2. **Código más limpio y legible**: No es necesario manejar manualmente la liberación de recursos (como cerrar archivos o conexiones).
3. **Manejo de excepciones**: Si ocurre una excepción dentro del bloque `with`, el recurso sigue liberándose de manera segura.

Este mecanismo es posible gracias a los **managers de contexto** que definen dos métodos: `__enter__()` (que se ejecuta al entrar en el bloque) y `__exit__()` (que se ejecuta al salir del bloque, liberando recursos).

Anexo 3: El módulo estándar “shutil” de Python

`shutil` es útil cuando necesitas realizar operaciones de alto nivel en ficheros y directorios, como copiar, mover o eliminar ficheros.

Ejemplo:

```
import shutil
shutil.copy('archivo.txt', 'copia.txt') # Copia un archivo
```

Anexo 4: Configurar Visual Studio Code para trabajar con Python

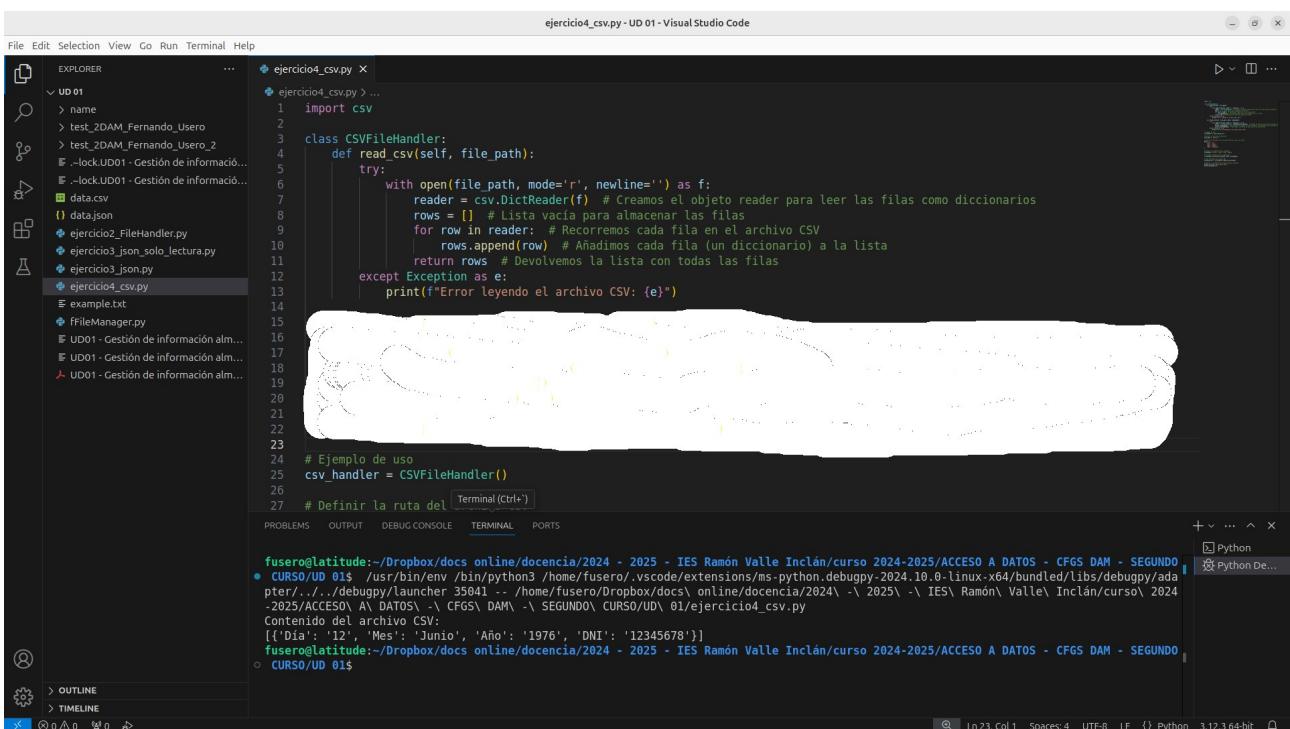
1. Instalar la Extensión de Python

- La **extensión de Python de Microsoft** es esencial. Ofrece características como IntelliSense, autocompletado, depuración y mucho más.
 - Ve a Extensiones (Ctrl + Shift + X), busca "Python" de Microsoft e instálala.

2. Configurar el Intérprete de Python

- Después de instalar la extensión de Python, debes configurar el intérprete de Python.
 - Usa la paleta de comandos (**Ctrl + Shift + P**) y busca **Python: Select Interpreter**. Elige el entorno de Python correcto (por ejemplo, virtualenv, conda o Python del sistema).

Actividad 5 de clase: con estas instrucciones, ejecuta la actividad 4 de clase con Visual Studio Code y proporciona una captura de pantalla, en la que se vea el código de la actividad 4 de clase y la traza de ejecución. Se adjunta captura de pantalla a modo de ejemplo.



Anexo 5: ¿Cómo funciona `__name__` en Python?

Veamos un ejemplo sencillo para que entiendas cómo funciona `__name__` dependiendo de si un archivo es ejecutado directamente o importado como un módulo.

1. Ejemplo de archivo ejecutado directamente

Crea un archivo llamado `mi_modulo.py` con el siguiente código:

```
def funcion_modulo():
    print("La función 'funcion_modulo' fue llamada.")

if __name__ == '__main__':
    print("Este archivo se está ejecutando directamente.")
    funcion_modulo()
else:
    print("Este archivo está siendo importado como un módulo.")
```

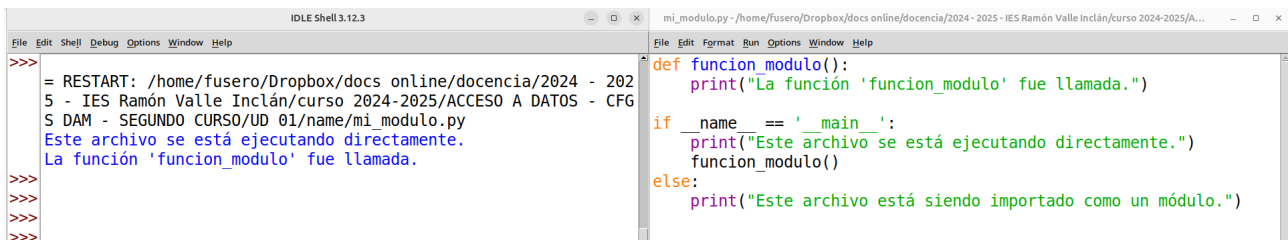
Ahora, si ejecutas `mi_modulo.py` directamente desde la terminal o tu entorno de desarrollo:

```
python mi_modulo.py
```

El resultado será:

Este archivo se está ejecutando directamente.

La función 'función_modulo' fue llamada.



2. Ejemplo de archivo importado como un módulo

Ahora crea otro archivo en la misma carpeta llamado `main.py` con el siguiente código:

```
import mi_modulo

print("Estoy en el archivo 'main.py'.")

mi_modulo.funcion_modulo()
```

Cuando ejecutas `main.py`, importará el módulo `mi_modulo` y el resultado será:

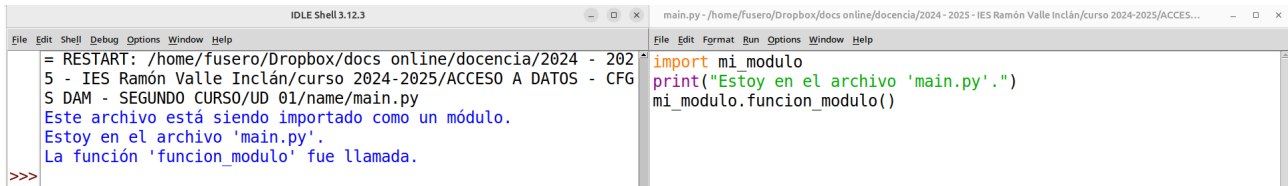
`python main.py`

El resultado será:

Este archivo está siendo importado como un módulo.

Estoy en el archivo 'main.py'.

La función 'funcion_modulo' fue llamada.



The screenshot shows two windows from the IDLE 3.12.3 environment. The left window, titled 'IDLE Shell 3.12.3', displays the output of running 'python main.py'. The output text is: 'RESTART: /home/fusero/Dropbox/docs online/docencia/2024 - 2025 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFG - SEGUNDO CURSO/UD 01/name/main.py', 'Este archivo está siendo importado como un módulo.', 'Estoy en el archivo 'main.py'.', and 'La función 'funcion_modulo' fue llamada.' followed by a prompt '>>>'. The right window, titled 'main.py - /home/fusero/Dropbox/docs online/docencia/2024 - 2025 - IES Ramón Valle Inclán/curso 2024-2025/ACCESO A DATOS - CFG - SEGUNDO CURSO/UD 01/name/main.py', shows the source code: 'import mi_modulo', 'print("Estoy en el archivo 'main.py'.")', and 'mi_modulo.funcion_modulo()'.

¿Qué pasa aquí?

- Cuando ejecutas ``mi_modulo.py`` directamente, el valor de ``__name__`` es ``'__main__'``, por lo que el bloque ``if __name__ == '__main__':`` se ejecuta.
- Cuando importas ``mi_modulo.py`` desde otro archivo, como ``main.py``, el valor de ``__name__`` es ``'mi_modulo'``, y no se ejecuta el bloque que está bajo ``if __name__ == '__main__':``.

Con este comportamiento, puedes controlar qué código se ejecuta cuando un archivo se usa como módulo o se ejecuta directamente.