Programación Funcional, Almacenamiento, Fechas y Expresiones Regulares

python

INDICE

1. Introducción a la programación funcional 5
   1.1 Funciones anónimas(lambda) 5
   1.2 Aplicación a Colecciones: map, filter, y reduce 7
2. Expresiones regulares 11
   2.1 El Módulo re: Funciones Clave 12
   2.2 Sintaxis básica de patrones 13
   2.3 Cuantificadores (Repeticiones) 14
   2.4 Caracteres de posición y comodín 15
   2.5 El Objeto Match y Grupos de Captura 21
   2.6 Reemplazar el texto con expresiones regulares 21
   2.7 Grupos sin captura 23
   2.8 Aserciones de mirada (Lookaheads y Lookbehinds) 24
   2.9 Voracidad 27
3. Fechas 29
   3.1 Introducción: Por qué las Fechas son Complejas 29
   3.2 Casos de uso comunes 29
   3.3 Módulos principales: date, time, datetime 29
   3.4 Crear y Obtener Fechas 30
      3.4.1 Componentes individuales de una fecha 30
   3.5 Formatear Fechas: strftime() 31
      3.5.1 Localización: Fechas en español 32
   3.6 Parsear Fechas: strptime() 32
      3.6.1 Manejo de errores en parseo 33
   3.7 Operaciones con fecha: timedelta 33
   3.8 Comparación de fechas 34
4. Ficheros 36
   4.1 Introducción - ¿ Por qué necesitamos ficheros? 36
   4.2 El objeto Path (pathlib) 37
      4.2.1 Crear objetos Path 37
      4.2.2 Propiedades del objeto Path 38
   4.3 Manejo de rutas relativas y absolutas 39
      4.3.1 Path(__file__) - Rutas relativas al script 40
   4.4 Script y variables reutilizables 42
      4.4.1 if __name__ == '__main__' 42
      4.4.2 Patrón profesional con main() 42
      4.4.3 Otras variables especiales 43

4.5 Buscar archivos 44
4.5.1 glob() - Búsqueda con patrones 44
4.5.2 rglob() - Búsqueda recursiva 44
4.5.3 iterdir() - Listar contenido de un directorio 45
4.6 Lectura de ficheros 45
4.6.1 read_text() - Lectura completa (método simple) 45
4.6.2 splitlines()) – Leer línea por línea 47
4.6.3 read_bytes() - Ficheros binarios 47
4.7 Escritura de ficheros 48
4.7.1 write_text() - Escritura simple 48
4.7.2 write_bytes() - Escritura binaria 49
4.7.3 Modo Append - Añadir sin borrar 49
4.8 Operaciones con archivos y directorios 51
4.8.1 Crear y eliminar archivos 51
4.8.2 Crear y eliminar directorios 52
4.8.3 Renombrar y mover archivos 52
4.9 Módulos os y shutil: Interacción con el Sistema 53
4.9.1 Módulo os – Operaciones básicas 53
4.9.2 El módulo sutil (Utilidades de alto nivel) 57
4.10 Manejo de errores con ficheros 58
4.11 Ficheros CSV – Datos tabulares 59
4.11.1 Lectura de CSV con DictReader (como diccionario) 59
4.11.2 Lectura de CSV con reader (Leer como Listas) 60
4.11.3 Escribir CSV: csv.writer 60
5. Almacenamiento de Datos (JSON) 62
5.1 Introducción a JSON 62
5.1.1 ¿Por qué usar JSON? 62
5.1.2 Casos de uso comunes 63
5.1.3 Ejemplo de JSON 63
5.2 Mapeo: Estructuras Fundamentales de JSON y Python 63
5.2.1 Mapeo de estructuras principales 63
5.2.2 Tabla de mapeo completa 63
5.3 Serialización: de Python a JSON 64
5.3.1 El Método json.dumps() - Dump String 64
5.3.2 El método json.dump() – Guardar en archivo 64
5.4 Deserialización: De JSON a Python (load y loads) 65
5.4.1 El Método json.loads() - Load String 65

5.4.2 El Método json.load() – Cargar desde archivo 66
5.4.3 Manejo profesional de errores 67
5.5 Casos de uso prácticos 67
5.5.1 Archivo de configuración de aplicación 67
5.5.2 Guardar respuesta de API 68
5.5.3 Loggin estructurado 68
5.6 Buenas prácticas y recomendaciones 69
5.6.1 Encoding UTF-8 69
5.6.2 Parámetro ensure_ascii 70
5.6.3 Validación JSON 70
5.6.4 No guardar datos sensibles en JSON plano 71
5.7 Tipos de datos avanzados 71
5.7.1 Problema: JSON no soporta datetime 71

1. Introducción a la programación funcional

La Programación Funcional (PF) es un paradigma de programación que trata la computación como la evaluación de funciones matemáticas. En lugar de dar pasos explícitos sobre cómo cambiar el estado del programa (programación imperativa), la PF se enfoca en qué se debe calcular.
En Python, aunque es un lenguaje multiparadigma (no puramente funcional), adoptamos estas herramientas para escribir código más predecible, conciso y fácil de probar.
Los pilares de la PF que aplicaremos son:
- Funciones Puras: Dadas las mismas entradas, siempre devuelven la misma salida y no tienen efectos secundarios (no modifican nada fuera de ellas, no hacen print, etc.). Hacen el código predecible y fácil de optimizar.
- Inmutabilidad: Una vez creado, un dato o estructura (como una tupla) no puede ser alterado. En lugar de modificar, se crea una nueva versión. Elimina errores de estado y simplifica el manejo de concurrencia.
- Funciones como Ciudadanos de Primera Clase: Las funciones pueden ser tratadas como cualquier otra variable: se asignan, se pasan como argumentos y se devuelven como resultados. Permite el uso de map, filter y sorted (las funciones que reciben otras funciones como argumento).

Las funciones **map**, **filter** y **reduce** son pilares de la programación funcional en Python. Permiten procesar colecciones de datos de manera muy eficiente, concisa y expresiva, a menudo reemplazando bucles for complejos.

1.1 Funciones anónimas(lambda)

Una función lambda en Python es una pequeña función anónima (sin nombre) definida en una sola línea. Se utiliza para crear funciones que solo se necesitan en el lugar donde se usan (en línea), son temporales y generalmente realizan una operación simple.
La sintaxis es estrictamente limitada a una expresión única:

lambda argumentos: expresión

- lambda: La palabra clave que inicia la función.
- argumentos: Uno o más argumentos de entrada, separados por comas.
- expresión: Una única expresión que se evalúa y cuyo resultado es el valor de retorno de la función.

Limitaciones
- Una Sola Expresión: Una lambda no puede contener múltiples sentencias.
- Sin Declaraciones Explícitas: No puede contener bucles (for, while), asignaciones de variables (=), ni la sentencia return (la expresión se devuelve automáticamente).

Casos de uso
- **Uso Básico** (Sustituto de una Función Simple).

Una lambda puede reemplazar cualquier función que contenga solo una sentencia return.
- **Uso con múltiples argumentos**.

Las lambda pueden manejar cualquier número de argumentos.

```
# Lambda para calcular la hipotenusa (c = sqrt(a^2 + b^2))
hipotenusa = lambda a, b: (a**2 + b**2)**0.5

print(f"Hipotenusa de (3, 4): {hipotenusa(3, 4)}") # Salida: 5.0
```

```python
# Lambda para formatear un nombre
formatear_nombre = lambda nombre, apellido: f"{apellido.upper()}, {nombre}"

print(f"Nombre formateado: {formatear_nombre('Luis', 'García')}")
# Salida: GARCÍA, Luis
```

- **Expresión condicional (ternaria)**
Es la única forma de implementar lógica if/else dentro de una lambda.
Sintaxis: lambda argumento:valor_si_True if condición else valor_si_False

```python
# Lambda para asignar estado "Activo" si el saldo es positivo, si no "Inactivo"
verificar_saldo = lambda saldo: "ACTIVO" if saldo > 0 else "INACTIVO"

print(f"Estado con 150€: {verificar_saldo(150)}") # Salida: ACTIVO
print(f"Estado con -5€: {verificar_saldo(-5)}")   # Salida: INACTIVO
```

- **Uso con sorted**() (El Caso más Frecuente)
Este es el caso de uso más común y profesional. Se utiliza lambda como el argumento key de sorted() para especificar un criterio de ordenación complejo. La lambda indica qué valor de cada elemento debe usarse para comparar.

```python
alumnos = [('Ana', 95), ('Carlos', 78), ('Marta', 99)]

# Queremos ordenar por la puntuación (índice 1)
ordenado_por_nota = sorted(alumnos, key=lambda alumno: alumno[1], reverse=True)

print("Ordenado por nota:", ordenado_por_nota)
# Salida: [('Marta', 99), ('Ana', 95), ('Carlos', 78)]
```

```python
# Definición equivalente a: def duplicar(x): return x * 2
duplicar = lambda x: x * 2

print(duplicar(5)) # Salida: 10

puntos = [(1, 5), (3, 2), (2, 8)]

# Ordenar por el segundo elemento de la tupla (índice 1)
puntos_ordenados = sorted(puntos, key=lambda p: p[1])
print(puntos_ordenados) # Salida: [(3, 2), (1, 5), (2, 8)]
```

Ejercicio 1
Se te da una lista de diccionarios, donde cada diccionario representa un registro de un producto.

```python
productos = [
```

```
    {'nombre': 'Laptop', 'precio': 1200, 'stock': 5},
    {'nombre': 'Mouse', 'precio': 25, 'stock': 12},
    {'nombre': 'Monitor', 'precio': 350, 'stock': 2},
    {'nombre': 'Teclado', 'precio': 75, 'stock': 8}
]
```

Ordena la lista original de productos por precio de forma ascendente usando sorted() y una función lambda(). La salida es la siguiente:
Salida:
Mouse: 25€
Teclado: 75€
Monitor: 350€
Laptop: 1200€

1.2 Aplicación a Colecciones: map, filter, y reduce

Estas tres funciones son las herramientas fundamentales de la Programación Funcional para manipular colecciones de datos (listas, tuplas, etc.) sin necesidad de escribir bucles for explícitos.

map(): Aplicar una Función a cada elemento

La función map() aplica una función a cada elemento de un iterable y devuelve un nuevo iterable con los resultados de esas transformaciones. Su objetivo es la transformación uno a uno.
Sintaxis:

map(funcion, iterable)

- función: La función que se aplicará a cada elemento (puede ser una función definida o una lambda).
- iterable: La lista, tupla u otro iterable cuyos elementos serán procesados.

Ejemplo de cálculo de IVA:

```
precios = [100.0, 50.5, 200.0]
IVA = 0.21

# Usamos lambda para calcular el precio final con IVA
precios_finales = list(map(lambda p: p * (1 + IVA), precios))

print(f"Precios Originales: {precios}")
print(f"Precios con IVA: {precios_finales}")
# Salida: [121.0, 61.105, 242.0]
```

Nota: map() devuelve un objeto map (un iterador), por lo que generalmente se convierte a una lista (list()) para visualizar o trabajar con los resultados. El equivalente más parecido en Python es la List Comprehension: [p * (1 + IVA) for p in precios].

Ejemplo: Convertir una lista de temperaturas de Celsius a Fahrenheit.

```
# f(C) = (C * 9/5) + 32
temperaturas_celsius = [0, 10, 25, 30]

# Usamos lambda para definir la transformación
```

```
fahrenheit = list(map(lambda c: (c * 9/5) + 32, temperaturas_celsius))


print(fahrenheit)
# Salida: [32.0, 50.0, 77.0, 86.0]


# Nota: El equivalente puede ser la List Comprehension:
# fahrenheit_comp = [(c * 9/5) + 32 for c in temperaturas_celsius]
```

filter(): Filtra elementos que cumplen una condición

La función filter() construye un nuevo iterable a partir de los elementos del iterable original para los cuales la función de filtrado devuelve True. Es decir, devuelve un objeto filter (un iterador) que contiene solo los elementos para los cuales la función de prueba resultó True.

Sintaxis:

filter(funcion_booleana, iterable)

- función_booleana: La función que debe devolver True o False (la prueba lógica). Solo los elementos para los que devuelve True se incluyen en el resultado.
- iterable: La colección de datos a filtrar.

Ejemplo: Filtrar los salarios por encima de un umbral.

```
salarios = [45000, 75000, 62000, 90000, 48000]
UMBRAL = 70000


# Usamos lambda para verificar si el salario es mayor que el umbral
salarios_altos = list(filter(lambda s: s > UMBRAL, salarios))


print(f"Salarios Totales: {salarios}")
print(f"Salarios Altos (> {UMBRAL}): {salarios_altos}")
# Salida: [75000, 90000]
```

Nota: Al igual que map(), filter() devuelve un objeto iterator que debe ser convertido a una lista si se necesita acceder a todos los elementos. Su equivalente más parecido es la List Comprehension con if: [s for s in salarios if s > UMBRAL].

Ejemplo: Seleccionar solo los números pares de una lista.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8]


# La función lambda devuelve True si el número es par.
pares = list(filter(lambda x: x % 2 == 0, numeros))


print(pares)
# Salida: [2, 4, 6, 8]


# Nota: El equivalente es la List Comprehension con if:
# pares_comp = [x for x in numeros if x % 2 == 0]
```

reduce(): reducción(acumulación)
La función reduce() aplica una función de dos argumentos de forma acumulativa a los elementos de un iterable, de izquierda a derecha, reduciendo el iterable a un único valor. A diferencia de map y filter (que son nativas), reduce() requiere importación desde el módulo functools.
Sintaxis:

```
from functools import reduce
reduce(funcion, iterable, [inicial])
```

- función: Una función que toma dos argumentos (el acumulador y el valor actual).
- iterable: La colección de datos a reducir.
- inicial: (Opcional) Un valor inicial para el acumulador. Si se omite, el primer elemento del iterable es el valor inicial.

Ejemplo: Cálculo del factorial

```python
from functools import reduce

# Cálculo del factorial de 5: 5 * 4 * 3 * 2 * 1
numeros = [5, 4, 3, 2, 1]

# La función lambda toma dos argumentos: 'a' (acumulador) y 'b' (valor actual)
factorial = reduce(lambda a, b: a * b, numeros)

print(f"Números: {numeros}")
print(f"Resultado del Factorial: {factorial}")
# Pasos internos: (((5*4)*3)*2)*1 = 120
```

Ejemplo: Multiplicar todos los elementos de una lista.

```python
from functools import reduce

datos = [2, 3, 5, 4] # El resultado esperado es 2 * 3 * 5 * 4 = 120

# La función lambda (acumulador, elemento_actual)
producto = reduce(lambda acc, x: acc * x, datos)

print(producto)
# Salida: 120

# Secuencia de reduce:
# 1. acc=2, x=3 -> 6
# 2. acc=6, x=5 -> 30
# 3. acc=30, x=4 -> 120
```

Para la suma (la operación de reducción más común), se debe usar la función nativa sum() en lugar de reduce(lambda a, b: a + b, datos).

Ejercicio 2
Tienes una lista de pedidos de clientes, donde cada pedido es una tupla (cantidad, precio_unitario, estado).

- Filtrar (filter): Obtén solo los pedidos que están en estado 'ENVIADO'.
- Mapear (map): Calcula el coste total de cada pedido restante (cantidad × precio_unitario).
- Reducir (reduce): Suma todos los costes totales para obtener el ingreso bruto total de los pedidos enviados.

```
pedidos = [
    (5, 10.0, 'ENVIADO'),
    (10, 2.5, 'PENDIENTE'),
    (2, 50.0, 'ENVIADO'),
    (3, 30.0, 'CANCELADO'),
    (1, 15.0, 'ENVIADO')
]
```

Realizar la alternativa con compresión de lista: En Python moderno, las comprensiones de lista y los generadores son la alternativa más común a map y filter, ya que son más legibles y, a menudo, igual de rápidos. Mientras que la versión map/filter/reduce es puramente funcional, la versión con comprensión es la más aconsejada y se recomienda en la mayoría de los casos de Python.

Si puedo usar list comprehesion en vez de map o filter, ¿que sentido tiene?
El sentido de usar filter o map hoy en día en Python se reduce a tres factores principales: Legibilidad en casos específicos, Programación Funcional, y Compatibilidad/Historia.
Para la mayoría de los desarrolladores Python, especialmente para principiantes e intermedios, la List Comprehension ([expresión for elemento in iterable if condición]) es la opción preferida porque:

- Es más legible y explícita: El código es más directo y fácil de leer para los programadores de Python.
- Devuelve una lista directamente: map() y filter() devuelven un objeto iterator (un generador), y a menudo tienes que forzarlo a ser una lista con list():

```
# List Comprehension (Directo y conciso)
pares = [x for x in numeros if x % 2 == 0]


# Filter (Requiere list() y una lambda)
pares = list(filter(lambda x: x % 2 == 0, numeros))
```

<u>Razones para usar map y filter</u>
A pesar de la preferencia por la List Comprehension, estas funciones siguen siendo herramientas válidas y tienen su lugar.
Programación Funcional Pura (Estilo)

- Composición de Funciones: En la programación funcional, a menudo se encadenan operaciones (compose). Usar map y filter facilita la composición sin la sintaxis de corchetes de la List Comprehension, lo que resulta en un estilo más declarativo.
- Tratamiento de la Función como Objeto: map y filter tratan las funciones como objetos de primera clase, lo que refuerza el paradigma funcional.

Legibilidad con Funciones Preexistentes

Si ya tienes una función compleja definida (no una lambda) y quieres aplicarla a un iterable, map puede ser más claro:

```
# Tienes esta función ya definida
def calcular_impuesto(precio):
    return precio * 0.21

precios = [100, 200, 50]

# Uso de map: Más directo
impuestos_map = list(map(calcular_impuesto, precios))

# Uso de List Comprehension: Requiere reescribir la llamada
impuestos_lc = [calcular_impuesto(p) for p in precios]
```

En este caso, map es una forma muy limpia de decir: "Aplica esta función existente a toda esta colección".

Eficiencia en el Uso de Iteradores (Generadores)

- Retorno Perezoso (Lazy Evaluation): Tanto map como filter devuelven generadores (objetos iterator). Esto significa que no calculan ni almacenan todos los resultados en memoria inmediatamente, solo calculan el siguiente valor cuando se les pide (por ejemplo, en un bucle for o al usar list()).
- Ventaja en Datos Enormes: Si estás trabajando con un conjunto de datos extremadamente grande donde almacenar la lista completa en memoria sería costoso o imposible, usar map o filter sin convertir el resultado a una lista inmediatamente (list(...)) es mucho más eficiente en el uso de memoria que una List Comprehension.

Cuál usar

- Regla de Oro (90% de los casos): Si la operación es simple, se requiere la lista completa inmediatamente, y no tiene sentido definir una función aparte, usa la List Comprehension.
- Caso Funcional o Complejo: Si ya tienes una función compleja definida y quieres aplicarla, o si estás haciendo una composición funcional o buscando eficiencia de memoria, usa map o filter.

2. Expresiones regulares

Una Expresión Regular es una secuencia de caracteres que define un patrón de búsqueda. En lugar de buscar una cadena exacta, buscamos un patrón que puede coincidir con múltiples cadenas.
En Python, todas las operaciones de RegEx se realizan a través del módulo “re”.

2.1 El Módulo re: Funciones Clave

Antes de ver los patrones, es vital conocer las funciones del módulo que actúan sobre el texto:

re.match(patrón, cadena): Intenta coincidir el patrón solo al principio de una cadena de texto. Si el patrón no comienza inmediatamente en el primer carácter de la cadena, re.match() **falla** y devuelve None, sin importar si el patrón existe en otro lugar de la cadena.

re.search(patrón, cadena): Escanea toda la cadena buscando la primera ubicación donde el patrón produce una coincidencia.

re.findall(patrón, cadena): Devuelve una **lista** de todas las subcadenas que coinciden con el patrón.

re.sub(patrón, reemplazo, cadena): Sustituye todas las coincidencias del patrón con la cadena de reemplazo.

re.split(patrón, cadena): Divide la cadena usando el patrón como delimitador.

Diferencia entre match y search:

```
import re

patron = r"Python"
texto1 = "Python es divertido."
texto2 = "Me gusta Python."

# 1. Coincidencia exitosa (el patrón está al inicio)
resultado_match_1 = re.match(patron, texto1)

# 2. Coincidencia fallida (el patrón NO está al inicio)
resultado_match_2 = re.match(patron, texto2)
resultado_search_2 = re.search(patron, texto2) # Usamos search para comparar
```

Ejemplo de uso de split:

```
import re

texto = "manzanas,naranjas;uvas.kiwi"
# Patrón: Busca una coma, un punto y coma O un punto
patron = r",|;|\."

lista_frutas = re.split(patron, texto)

print(lista_frutas)
# Salida: ['manzanas', 'naranjas', 'uvas', 'kiwi']
```

O eliminar uno o más espacios:

```
import re
```

```
texto = "101  202   303 404"
# Patrón: \s+ (uno o más espacios en blanco)
patron = r"\s+"

lista_numeros = re.split(patron, texto)

print(lista_numeros)
# Salida: ['101', '202', '303', '404']
```

2.2 Sintaxis básica de patrones

Para crear nuestra primera expresión regular, hay que crear el patrón, que no es más que otra cadena de texto que describe lo que queremos encontrar:
patron = “Hola”
Luego hay que indicar el texto donde queremos encontrar el patrón:
texto = “Hola mundo”
Y finalmente, usar una función de búsqueda de “re”
resultado = re.search(patron, texto)
Si no encuentra el patrón “resultado” devuelve None, pero si encuentra algo, nos devolverá el objeto Match.

```
import re

patron = "Hola"
texto = "Hola mundo"
resultado = re.search(patron, texto)
if resultado:
    print("¡Coincidencia encontrada!")
else:
    print("No se encontró ninguna coincidencia.")
```

Si una función como re.search() encuentra una coincidencia, devuelve un objeto Match que contiene información detallada:
.group(0): Devuelve la cadena que coincide completamente con el patrón.
.start(): Devuelve la posición del índice inicial de la coincidencia.
.end(): Devuelve la posición del índice final(exclusiva) de la coincidencia.
.groups(): Devuelve una tupla con el texto de todos los grupos de captura ().

Ejercicio 1
Encuentra la primera ocurrencia de la palabra “IA” en el siguiente texto e indica en qué posición empieza y termina la coincidencia. Texto = "Todo el mundo dice que la IA nos va a quitar el trabajo. Pero solo hace falta ver cómo la puede cagar con Las Regex para ir con cuidado"

findall
Si queremos encontrar todas las ocurrencias de un patrón en un texto, usar “findall”.
Ejemplo:

```python
import re

patron = "Python"
texto = "Python es un lenguaje de programación muy popular. Muchas personas aprenden Python para desarrollo web, análisis de datos, inteligencia artificial, y más. Python es mejor que Java"
resultado = re.findall(patron, texto)
print(f"La palabra '{patron}' se encontró {len(resultado)} veces en el texto.")
```

El mostrar las ocurrencias en una lista está bien, pero si queremos todos los datos como se hacía con "search", tendremos que usar "iter", que devuelve un iterador que contiene todos los resultados de la búsqueda.

```python
import re

patron = "Python"
texto = "Python es un lenguaje de programación muy popular. Muchas personas aprenden Python para desarrollo web, análisis de datos, inteligencia artificial, y más. Python es mejor que Java"
resultado = re.finditer(patron, texto)
for coincidencia in resultado:
    print(f"¡Coincidencia encontrada! Empieza en la posición {coincidencia.start()} y termina en la posición {coincidencia.end()}.")
```

La fuerza de RegEx reside en su sintaxis de metacaracteres.

2.3 Cuantificadores (Repeticiones)

Estos metacaracteres indican cuántas veces debe aparecer el elemento anterior en el patrón.
Los metacaracteres son caracteres con un significado especial.

Carácter	Nombre	Significado	Ejemplo de Coincidencia
*	Cero o más	Coincide cero o más repeticiones del elemento anterior.	a* coincide con "", "a", "aa", "aaa", etc.
+	Uno o más	Coincide una o más repeticiones del elemento anterior.	a+ coincide con "a", "aa", "aaa", etc., pero no con "".
?	Cero o uno	Coincide cero o una repetición del elemento anterior (lo hace opcional).	colou?r coincide con "color" y "colour".
{n}	Repeticiones fijas	Coincide exactamente n repeticiones del elemento anterior.	\d{4} coincide con un año: "2025".
{n,m}	Rango	Coincide entre n y m repeticiones del elemento anterior.	\w{3,5} coincide con palabras de 3 a 5 letras.
()	Agrupación	Agrupa partes del patrón para aplicar cuantificadores	(ab)+ coincide con "ab", "abab", "ababab".

		o para capturar el texto coincidente.	

2.4 Caracteres de posición y comodín

^	Inicio	Coincide con el inicio de la cadena.	^A coincide con "Ana" pero no con "Maria".
$	Final	Coincide con el final de la cadena.	z$ coincide con "jazz" pero no con "zebra".
.	Comodín	Coincide con cualquier carácter (excepto salto de línea \n).	a.o coincide con "amo", "a-o", "azo".
|	Opción	Estos metacaracteres indican cuántas veces debe aparecer el elemento anterior en el patrón.	

Puedes ver una referencia rápida y hacer pruebas en: https://regex101.com

Escape de Metacaracteres

Si quieres buscar un carácter que tiene un significado especial (como punto(.), *, +, ?, ^, $, \, |, (), [], {}), debes escaparlo con una barra invertida \ para buscarlo literalmente.

Para buscar un punto literal: \.
Para buscar un signo de dólar literal: \$

Recuerda usar la cadena sin formato r"…"!

```
import re

texto = "El costo es de $5.99."
patron_literal = r"\$5\.99" # Buscamos el '$' y el '.' literales
coincidencia = re.search(patron_literal, texto)
print(f"Búsqueda literal: {coincidencia.group() if coincidencia else 'No encontrado'}")
# Salida: Búsqueda literal: $5.99
```

Ejemplo de uso del punto(.):

```
import re

texto = "casa, cosa, c-s, c1s, c$s, c_s, cuentos, cars."
patron = r"c.s"

# El punto coincide con el segundo carácter, sin importar cuál sea.
coincidencias = re.findall(patron, texto)

print(f"Texto: {texto}")
print(f"Patrón: '{patron}'")
```

```
print(f"Coincidencias: {coincidencias}")
# Salida: ['cas', 'cos', 'c-s', 'c1s', 'c$s', 'c_s']
```

Como puede ver, el patrón tiene una “r” delante, esta “r” se usa para evitar que Python interprete las barras invertidas (\) de una manera no deseada.
En Python, siempre es mejor definir los patrones RegEx usando la notación r'' (raw string), como en r'\d+'. Esto evita que Python interprete las barras invertidas (\) antes de que el motor de RegEx tenga la oportunidad de hacerlo. Por ejemplo, '\n' es un salto de línea para Python, pero r'\n' es la secuencia de dos caracteres \ y n para RegEx.

```
import re

texto = "Mi casa es blanca. La casa de mi amigo es azul. La casa de mi vecino es roja."
patron = "."
coincidencias = re.findall(patron, texto)
print(coincidencias)
```

Aquí el punto tiene un significado especial, que es cualquier carácter, por lo que no funciona como queremos. Esto se soluciona poniendo una “r” delante el patrón y escapando el punto (.), anulando el significado especial del punto(.).

```
import re

texto = "Mi casa es blanca. La casa de mi amigo es azul. La casa de mi vecino es roja."
patron = r"\."
coincidencias = re.findall(patron, texto)
print(coincidencias)
```

Modificadores

Los modificadores en las expresiones regulares son cruciales porque cambian la forma en que el motor de RegEx interpreta ciertos metacaracteres y la cadena de entrada.

En Python, estos modificadores se pasan como un argumento adicional a las funciones del módulo re (como re.search, re.findall, etc.) usando la sintaxis re.FUNCION(patron, cadena, flags=re.MODIFICADOR).

- re.IGNORECASE ó re.I: Este modificador hace que la comparación no distinga entre mayúsculas y minúsculas.

```
import re

patron = "python"
texto = "python es un lenguaje de programación muy popular. Muchas personas aprenden PyThon para desarrollo web, análisis de datos, inteligencia artificial, y más. Python es mejor que Java"
resultado = re.findall(patron, texto, re.IGNORECASE)
print(resultado)
```

- re.MULTILINE o re.M (Modo Multilínea): Este modificador afecta a los anclajes de inicio (^) y final ($). Por defecto, ^ coincide solo con el inicio de toda la cadena, y $ solo con el final de toda la cadena.

Con re.M, estos anclajes también coinciden con el inicio y el final de cada línea dentro de la cadena (justo después o antes de un salto de línea \n).

```
import re

cadena_multi = "Línea uno\nLínea dos\nLínea tres"

# Queremos encontrar líneas que terminan en 'uno' o 'dos' o 'tres'
patron_final = r"dos$"

# Sin Modificador: Busca 'dos' solo al final de la cadena (falla)
resultado_sin_m = re.search(patron_final, cadena_multi)
print(f"Sin re.M: {resultado_sin_m}")
# Salida: None

# Con re.MULTILINE: Busca 'dos' al final de CUALQUIER línea.
resultado_con_m = re.search(patron_final, cadena_multi, flags=re.MULTILINE)
print(f"Con re.M: {resultado_con_m.group(0) if resultado_con_m else None}")
# Salida: dos (Coincide con 'Línea dos\n')
```

- re.DOTALL o re.S (Modo Punto Coincide con Todo): Este modificador cambia el comportamiento del metacaracter punto (.), el comodín universal. Por defecto, “.” coincide con cualquier carácter excepto el salto de línea (\n).

Con re.S, el punto (.) también coincidirá con el carácter de salto de línea (\n), permitiendo que un patrón se extienda a través de múltiples líneas.

```
import re

cadena_multi = "Línea A\nB en Línea"
patron_punto = r"A.B"

# Sin Modificador: El punto NO coincide con \n
resultado_sin_s = re.search(patron_punto, cadena_multi)
print(f"Sin re.S: {resultado_sin_s}")
# Salida: None

# Con re.DOTALL: El punto SÍ coincide con \n
resultado_con_s = re.search(patron_punto, cadena_multi, flags=re.DOTALL)
print(f"Con re.S: {resultado_con_s.group(0) if resultado_con_s else None}")
```

```
# Salida: A\nLínea B (Coincide con 'A' + salto de línea + 'B')
```

Combinación de modificadores

Los modificadores se pueden combinar usando el operador bit a bit OR (|).

```
import re

patron = r"^lÍnea$" # Patrón sensible a mayúsculas y delimitado por anclajes

cadena = "Línea"

# Combinamos I (Ignorecase) y M (Multiline)
resultado = re.search(patron, cadena, flags=re.IGNORECASE | re.MULTILINE)

print(f"Resultado combinado (I|M): {resultado.group(0) if resultado else None}")
# Salida: Línea
```

Conjunto de caracteres

Definen un conjunto de caracteres de los cuales solo uno debe coincidir. Dentro de [], la mayoría de los metacaracteres pierden su significado especial.
[abc]: Coincide con 'a', 'b', o 'c'. [aeiou] coincide con cualquier vocal.
[a-z]: Rango. Coincide con cualquier letra minúscula de la 'a' a la 'z'.
[^abc]:Negación. Coincide con cualquier carácter excepto 'a', 'b', o 'c'.

Secuencias especiales

Son atajos para conjuntos de caracteres comunes.

Secuencia	Significado	Equivalente
\d	Dígito	[0-9]
\D	NO Dígito	[^0-9]
\w	Carácter de palabra	[a-zA-Z0-9_] (letras, números o guion bajo)
\W	NO Carácter de palabra	[^a-zA-Z0-9_]
\s	Espacio en blanco	[\t\n] (espacio, tabulación, salto de línea, etc.)
\S	NO Espacio en blanco	[^ \t\n]
\b	Coincide con el principio o final de una palabra	
|	Coincidir con una opción u otra	

Ejemplo: Queremos extraer un código postal que siempre consta de cinco dígitos consecutivos en una cadena de texto.

```
import re

texto = "Mi numero de teléfono es 06300"
encontrado = re.findall(r'\d\d\d\d\d', texto)
```

```
print(encontrado)
```

Pero es un rollo tener que poner varios “\d”, mejor usar los cuantificadores, el carácter de rango:

```
import re


texto = "Mi numero de teléfono es 06300"
encontrado = re.findall(r'\d{5}', texto)
print(encontrado)
```

El uso de \b

La posición que coincide \b es la transición entre un carácter de palabra (\w: letra, número o guion bajo) y un carácter que no es de palabra (\W: espacio, puntuación, inicio/fin de cadena, etc.), por lo que \b asegura que el patrón se encuentre como una palabra independiente.

\bpalabra\b Busca la secuencia exacta palabra que esté rodeada por límites de palabra (espacios, comas, puntos, etc., o el inicio/fin de la cadena).

\bpalabra Busca palabra al inicio de una palabra (ej. "palabrita").

palabra\b Busca palabra al final de una palabra (ej. "mi-palabra").

Ejemplo:

```
import re


texto = "El ratón y el ratoncito corren rápido. Raton? Si, Raton"
patron = "raton"
resultado = re.findall(patron, texto, re.IGNORECASE)
# Patrón sin anclaje (coincidencia parcial)
print(f"Patrón sin \b: {resultado}")
patron2 = r"\braton\b"
resultado2 = re.findall(patron2, texto, re.IGNORECASE)
# Patrón con anclaje (palabra completa)
print(f"Patrón con \b: {resultado2}")
```

Ejercicio 2

Encontrar si en la cadena hay un número de España.

texto = "Mi número de teléfono es +34 678475673 y el de portugal es +35 656748392"

Ejercicio 3

Validar un nombre de usuario (debe comenzar por algo alfanumérico).

texto = "%pepito_123%"

Si quitamos % del inicio del texto, indicará que el nombre de usuario es válido.

Ejercicio 4

Validar si la cadena acaba con mundo.

texto = "Hola mundo"

Ejercicio 5

Tenemos una lista de archivos, necesitamos saber los nombres de los ficheros con extension .txt
texto = "file1.txt file2.pdf image.png document.docx notes.txt secreto.txt"

Ejercicio 6
En el siguiente texto, filtrar la palabra “casa”.
texto = "casa casado casada"

Ejercicio 7
Nos dan una lista de frutas y filtrar por aguacate o palta.

```
texto = "platano, manzana, palta, aguacate, pera, aguacate"
```

Set y corchetes
[ ] significa que coincida con cualquier carácter dentro de los corchetes.
Patron = r"[\w._%+-]+": significa que aparezca una palabra, un punto, subrayado bajo, %, + o - 1 o más veces.
Ejemplo: encontrar todas las vocales que aparecen en el texto “Hola mundo”.

```
import re


texto = "Hola mundo"

patron = r"[aeiou]"

resultado = re.findall(patron, texto)

print("Vocales encontradas:", resultado)
```

También puedo usar los corchetes para buscar unas palabras que comiencen por determinados caracteres.

```
import re


texto = "casa tera masa clasa pasa "

patron = r"[cmp]asa"

resultado = re.findall(patron, texto)

print("Palabras encontradas:", resultado)
```

También podemos indicar que los números estén en cierto rango. Coger los teléfonos que tengan 9 caracteres y que empiecen entre 6 y 7.

```
import re


texto = "657463732 63617361 74657374 2063617361 6d617261"

patron = r"\b[6-7][0-9]{8}\b"

resultado = re.findall(patron, texto)

print("Palabras encontradas:", resultado)
```

Coincidencia negada

[^] Coincide con cualquier carácter que no esté dentro. Por ejemplo, el ejemplo anterior de buscar las vocales, poniendo ^ podemos indicar que me encuentre lo que no sea vocal.

```
import re


texto = "Hola mundo"
patron = r"[^aeiou]"
resultado = re.findall(patron, texto)
print("Consonantes encontradas:", resultado)
```

2.5 El Objeto Match y Grupos de Captura

Cuando re.match() o re.search() encuentran una coincidencia, devuelven un objeto match. Este objeto permite acceder a la información de la coincidencia.

Grupos de Captura ()

Los paréntesis () no solo agrupan partes del patrón, sino que también capturan el texto que coincide con esa parte específica.

match.group(0): Devuelve la coincidencia completa (por defecto).
match.group(1): Devuelve el texto capturado por el primer grupo().
match.groups(): Devuelve una tupla con todos los grupos capturados.

```
import re
cadena = "Mi email es user@dominio.com"


# Capturamos el usuario (grupo 1) y el dominio (grupo 2)
patron_email = r"(\w+)@(\w+\.com)"


coincidencia = re.search(patron_email, cadena)


if coincidencia:
    print(f"Coincidencia completa: {coincidencia.group(0)}")
    print(f"Usuario: {coincidencia.group(1)}")
    print(f"Dominio: {coincidencia.group(2)}")
```

2.6 Reemplazar el texto con expresiones regulares

La función re.sub() (de substitute) busca todas las subcadenas que coinciden con un patrón dado y las reemplaza con una cadena de reemplazo especificada.

Su sintaxis es:

re.sub(patrón, reemplazo, cadena, [count=0], [flags=0])

- patrón: La expresión regular a buscar.
- reemplazo: La cadena que se usará para sustituir cada coincidencia. Puede incluir referencias a grupos capturados.
- cadena: El texto original donde se realizará la búsqueda y el reemplazo.

- count (opcional): El número máximo de reemplazos a realizar. Por defecto, 0 (reemplaza todas las ocurrencias).
- flags (opcional): Modificadores como re.IGNORECASE o re.MULTILINE.

Reemplazo simple

El caso más sencillo es reemplazar una coincidencia con una cadena fija.
Ejemplo: reemplazar todos los IDs de usuario por la palabra USUARIO_ANÓNIMO.

```
import re


texto_log = "El ID 123456 realizó la acción. El ID 987654 falló."
patron_id = r"\d{6}" # Patrón: Seis dígitos


# Reemplazamos cualquier ID de 6 dígitos por una cadena fija
texto_modificado = re.sub(patron_id, "USUARIO_ANÓNIMO", texto_log)


print(f"Original: {texto_log}")
print(f"Modificado: {texto_modificado}")
# Salida: El ID USUARIO_ANÓNIMO realizó la acción. El ID USUARIO_ANÓNIMO falló.
```

Reemplazo con referencias a grupos capturados

La verdadera potencia de re.sub() se revela cuando se usan grupos de captura (()) en el patrón y se hace referencia a ellos en la cadena de reemplazo mediante la notación \1, \2, etc.

- \1 hace referencia al texto capturado por el primer grupo.
- \2 hace referencia al texto capturado por el segundo grupo.

Ejemplo: Si tienes nombres en formato "Apellido, Nombre" y quieres cambiarlos a "Nombre Apellido".

```
import re


nombres = ["Garcia, Luis", "Perez, Ana", "Sanchez, Elena"]
# Patrón: Captura (Grupo 1) + coma y espacio + Captura (Grupo 2)
patron_reordenar = r"(\w+),\s*(\w+)"


# Reemplazo: \2 (Nombre) + espacio + \1 (Apellido)
reemplazo = r"\2 \1"


print("--- Ejemplo 2: Reordenación ---")
for nombre in nombres:
    nombre_modificado = re.sub(patron_reordenar, reemplazo, nombre)
    print(f"'{nombre}' -> '{nombre_modificado}'")
# Salida: 'Garcia, Luis' -> 'Luis Garcia'
```

Ejemplo: Actualizar un protocolo sin modificar el resto del dominio o ruta.

```
import re

url = "Visita http://ejemplo.com/pagina y no http://otra.net"
# Captura el protocolo 'http' (Grupo 1) y el resto de la URL (Grupo 2)
patron_url = r"(http)://(\w+.\w+)"

# Reemplazo: Añade 's' a 'http' (\1) y luego el resto de la URL (\2)
reemplazo = r"\1s://\2"

url_modificada = re.sub(patron_url, reemplazo, url)

print(f"Original: {url}")
print(f"Modificada: {url_modificada}")
# Salida: Visita https://ejemplo.com y no https://otra.net
```

Ejercicio 8

Extraer todas las fechas que sigan el formato DD/MM/AAAA de una cadena de texto. Usa el patrón: \d{2}/\d{2}/\d{4}. Utiliza re.findall().

```
texto_agenda = "La reunión es el 05/11/2025. La fecha límite era el 30/10/2025, no el 1/1/2025."
```

Ejercicio 9

Valida si una cadena sigue el patrón de una matrícula española moderna: 4 dígitos seguidos de 3 letras mayúsculas.
- El patrón debe cubrir toda la cadena (usa ^ y $).
- Usa \d para los dígitos y [A-Z] para las letras.
- Usa re.match() o re.search() con los anclajes (^...$) para validar la cadena completa.

```
matriculas = ["1234ABC", "123AB", "A1234BC"]
```

Ejercicio 10

Reemplazar todos los caracteres que no sean letras o espacios por un guion bajo (_).
- Usa el conjunto de caracteres negado [^...] para capturar todo lo que NO sea letra o espacio.
- Usa re.sub().

```
cadena_sucia = "Archivo_Final-V1.0.txt (Copia)"
```

Ejercicio 11

Extraer el código de área, el código de central y el número final de un teléfono, separados por guiones opcionales.
- Usa los cuantificadores ? para hacer el guion opcional (-?).
- Usa grupos de captura () para las tres partes numéricas.
- Usa re.search() y match.groups().

```
telefono = "Mi teléfono es (555) 123-4567, pero el de la oficina es 987654321."
```

2.7 Grupos sin captura

Como se ha visto, los paréntesis () normalmente agrupan y capturan. Si solo quieres usar los paréntesis para agrupar elementos y aplicarles un cuantificador (por ejemplo, (ab)+), pero sin que se guarde el texto en el objeto Match, usas la sintaxis (?:...).

Esto es útil por dos razones:
- Optimización: Evita que el motor de RegEx capture y almacene información que no necesitas, lo que mejora la velocidad.
- Simplificación: Mantiene limpio el índice de grupos capturados.

Sintaxis: (?:patrón)
Ejemplo: Queremos encontrar las repeticiones de la palabra "batman" o "robin", y contar cuántas veces se repite el patrón completo, sin capturar individualmente "batman" o "robin".

```python
import re


texto = "batmanrobin batman batmanrobin"
# Patrón con grupo de captura: (\w+) (captura 'batman' o 'robin'), lo que complica el indexado.
patron_captura = r"(\w+)(robin|batman)"
coincidencias_captura = re.findall(patron_captura, texto)
# Salida: [('batman', 'robin'), ('batman', 'batman')] (Hay duplicados)
print(f"Resultado con captura: {coincidencias_captura}")


# Patrón sin captura: (?:...) - Agrupamos 'batman|robin' para poder cuantificarlo con el '+'
patron_sin_captura = r"(?:batman|robin)+"
coincidencias_sin_captura = re.findall(patron_sin_captura, texto)
# Salida: ['batmanrobin', 'batman', 'batmanrobin']
print(f"Resultado sin captura: {coincidencias_sin_captura}")
```

patron_captura busca una palabra, seguida de la palabra robin o batman y findall devuelve una tupla con los grupos que encuentra, en este caso batman y robin.

2.8 Aserciones de mirada (Lookaheads y Lookbehinds)

Las aserciones de mirada son una de las herramientas más potentes. Permiten crear condiciones para que una coincidencia ocurra, pero sin incluir la condición en la coincidencia final (la que devuelve .group(0) o re.findall()).
Estas aserciones son de "cero-ancho" (zero-width), es decir, consumen cero caracteres, solo comprueban si la condición se cumple en esa posición.

Lookahead (Mirada Adelante)
Comprueba si el patrón que le sigue coincide (positivo) o no coincide (negativo).

Tipo	Descripción	Sintaxis
Positivo	patrón(?=condición)	Coincide con patrón solo si es seguido por condición.
Negativo	patrón(?!condición)	Coincide con patrón solo si NO es seguido por condición.

Ejemplo de Lookahead Positivo (?=...)
Queremos encontrar todos los números, pero solo aquellos que son seguidos por el símbolo de Euro (€). Queremos que el símbolo no se incluya en el resultado.

```
import re


texto = "Precio: 100€, Coste: 50$, Tasa: 25€."
# Patrón: \d+ (uno o más dígitos) seguido de (?=€) (la condición de que haya un '€' a continuación)
patron = r"\d+(?=€)"
coincidencias = re.findall(patron, texto)
print(f"Precios en Euro: {coincidencias}")
# Salida: Precios en Euro: ['100', '25']
```

Ejemplo de Lookahead Negativo (?!...)
Queremos encontrar la palabra "gato" pero solo cuando NO está seguida inmediatamente de la letra 's' (para excluir plurales como 'gatos').

```
import re


texto = "El gato es negro. Me gustan los gatos."
# Patrón: gato seguido de (?!s) (la condición de que NO haya una 's' a continuación)
patron = r"gato(?!s)"
coincidencias = re.findall(patron, texto)
print(f"Coincidencias de 'gato' singular: {coincidencias}")
# Salida: Coincidencias de 'gato' singular: ['gato']
```

Lookbehind (Mirada Atrás)
Comprueba si el patrón que le precede coincide (positivo) o no coincide (negativo).
Nota: En Python, los patrones dentro del lookbehind deben ser de ancho fijo (longitud conocida), a diferencia de otros motores de RegEx.

Tipo	Sintaxis	Descripción
Positivo	(?<=condición)patrón	Coincide con patrón solo si es precedido por condición (ancho fijo).
Negativo	(?<!condición)patrón	Coincide con patrón solo si NO es precedido por condición (ancho fijo).

Ejemplo de Lookbehind Positivo (?<=...)
Queremos encontrar un nombre de archivo (por ejemplo, .pdf) solo si está precedido por data_. Queremos que data_ no esté en la coincidencia.

```
import re


texto = "archivo_final.txt data_reporte.pdf otra.pdf"
# Patrón: (?<=data_) (la condición de que haya 'data_' a la izquierda) seguido de \w+\.pdf
patron = r"(?<=data_)\w+\.pdf"
coincidencias = re.findall(patron, texto)
print(f"Archivos PDF de datos: {coincidencias}")
```

```
# Salida: Archivos PDF de datos: ['reporte.pdf']
```

Ejemplo de Lookbehind Negativo (?<=!...)
Sustituir la palabra "color" por "colour" (para un público británico), pero solo si NO está seguida por la letra 'i' (para no sustituir "colorido"). Usa re.sub().

```
import re


texto = "El color de la pintura es bonito, pero el colorido del cuadro es excesivo."
# Patrón de sustitución: color(?!i)
patron_avanzado = r"color(?!i)"
reemplazo = r"colour"


texto_modificado = re.sub(patron_avanzado, reemplazo, texto)
print(f"Texto modificado: {texto_modificado}")
```

A primera vista, puede parecer que las Aserciones de Mirada (Lookahead y Lookbehind) son innecesarias porque casi siempre puedes encontrar una manera de lograr una coincidencia similar con grupos de captura normales.

1. Simplificación de Resultados (Aislamiento)
Escenario sin Lookahead (Método tradicional con Captura)

Si quieres extraer un precio sin el símbolo del euro (€), tienes que usar un grupo de captura.
Patrón: (\d+)€
Texto: "100€"
Resultado de re.findall(): Te devuelve una lista de los grupos capturados, en este caso, ['100'].

Problema: Tienes que usar re.findall() (que devuelve solo los grupos) o .group(1) si usas re.search().

Escenario con Lookahead (Método de Aislamiento)
Usando un Lookahead positivo (?=...), aislas el patrón de la condición.
Patrón: \d+(?=€)
Texto: "100€"
Resultado de re.findall(): Te devuelve ['100'].

Ventaja: El patrón se lee de forma más intuitiva ("dígitos seguidos de €"). Además, utilizas la función re.findall() en su modo más simple, sin depender de los grupos de captura. El código es más limpio porque la condición (€) no está en el resultado.

2. Flexibilidad y Búsquedas Negativas
Búsqueda Negativa con Lookahead Negativo (?!...)

Las búsquedas negativas son mucho más difíciles o, a veces, imposibles de replicar con la sintaxis básica.

Ejemplo: Encontrar la palabra "gato" cuando no está seguida inmediatamente de una "s" (para excluir "gatos").

Patrón: gato(?!s)

Resultado: Coincide con "gato" en "El gato" pero ignora "gatos".

Sin Lookahead: El patrón alternativo podría ser gato[^s\b], pero esto tiene un problema serio: el [^s] consume el siguiente carácter, lo que significa que no podrás encontrar otra coincidencia inmediatamente después, o podrías coincidir con una 'o' de otra palabra. El Lookahead negativo (?!s) no consume ningún carácter, simplemente comprueba la condición en ese punto.

3. Solapamiento de Coincidencias
Debido a su naturaleza de "cero-ancho" (no consumen caracteres), los Lookaheads y Lookbehinds permiten encontrar patrones que se solapan.

Ejemplo: Encontrar todos los pares de dígitos repetidos en una secuencia larga (por ejemplo, 12123434).

* Patrón de Solapamiento (requiere Lookahead): (?=(\d\d))
    - Coincide con un grupo de 2 dígitos, pero como es un lookahead, el motor de RegEx no avanza su posición de búsqueda.
    - Resultado para "1212": ['12', '21', '12']

* Patrón Normal (no solapa): (\d\d)
    - Resultado para "1212": ['12', '12'] (Solo encuentra el primer '12', luego avanza, ignorando el '21' central).

En resumen, el Lookahead y Lookbehind no son solo una alternativa, son una herramienta esencial para:
- Limpieza del Código: Mantener la coincidencia (el resultado) limpia de las condiciones (el contexto).
- Búsqueda Condicional: Implementar condiciones negativas (e.g., "busca X, pero solo si no le sigue Y").
- Solapamiento: Encontrar patrones que se superponen en el texto.

2.9 Voracidad

El uso de (.*?) es fundamental en las expresiones regulares de Python (y en muchos otros lenguajes) porque soluciona el problema de la "Voracidad" de los cuantificadores. El patrón (.*?) es la combinación de tres metacaracteres con los siguientes significados:

Carácter	Significado	Función
.	Comodín	Coincide con cualquier carácter (excepto salto de línea, a menos que uses re.S).
*	Cuantificador Voraz	Coincide con cero o más repeticiones del elemento anterior (.).
?	Modificador No Voraz	Modifica al * para que sea No Voraz.
()	Grupo de Captura	Almacena la coincidencia para poder usarla en re.findall o re.sub.

En resumen, (.*?) significa: "Captura cualquier carácter, cero o más veces, pero detente en la primera oportunidad que tengas."

El problema de la voracidad

Por defecto, los cuantificadores como * (cero o más) y + (uno o más) son voraces. Esto significa que intentarán coincidir con la cadena más larga posible.

Ejemplo con Cuantificador Voraz (.*)

Imagina que tienes etiquetas HTML y quieres extraer el contenido dentro de la primera etiqueta ....

Texto: Texto A y Texto B

Patrón Voraz: .*

1. El motor RegEx encuentra la primera .
2. El .* dice: "Coincide con todo lo que puedas...".
3. El motor avanza hasta encontrar la última de la cadena para satisfacer el patrón.

Resultado de la Coincidencia Voraz: Texto A y Texto B (Incorrecto: capturó el contenido intermedio).

La Solución No Voraz (Non-Greedy)

Al añadir el signo de interrogación ? al cuantificador (* → *?), le indicas al motor que debe buscar la cadena más corta posible.

Ejemplo con Coincidencia No Voraz (.*?)

Texto: Texto A y Texto B

Patrón No Voraz: .*?

1. El motor RegEx encuentra la primera .
2. El .*? dice: "Coincide con la cantidad mínima de caracteres hasta que puedas satisfacer el resto del patrón."
3. El motor se detiene en la primera que encuentra.

Resultado de la Coincidencia No Voraz (Primera Coincidencia): Texto A (Correcto: Captura solo la primera etiqueta)

Ejercicio 12

Transformar el siguiente texto en formato Markdown (como si lo devolviera una API de OpenAI) a su equivalente en HTML.

```
# Resumen de la IA

La Inteligencia Artificial (IA) está transformando el mundo.


## Características Clave

La IA se basa en algoritmos.


* **Aprendizaje Automático (ML):** El núcleo de la IA.
* Procesamiento del Lenguaje Natural (PLN).
* Visión por Computadora.


Para más información, visita [Google](https://www.google.com).
```

Debes implementar cinco reglas de sustitución (re.sub) para convertir las siguientes sintaxis de Markdown a sus etiquetas HTML/CSS equivalentes:

1. Encabezados (H1): # Título → <h1>Título</h1>
2. Encabezados (H2): ## Subtítulo → <h2>Subtítulo</h2>
3. Negritas: **palabra** → palabra
4. Listas Desordenadas: * elemento → elemento (Esto requiere un paso extra, envolver el resultado final con <ul>...)
5. Enlaces: [Texto](URL) → Texto

3. Fechas

3.1 Introducción: Por qué las Fechas son Complejas

El manejo de fechas y horas es uno de los problemas más complejos en programación. Lo que parece simple ("sumar un día a una fecha") se complica por:

- Zonas horarias: 18:00 en Madrid ≠ 18:00 en Nueva York
- Horario de verano: Algunos días tienen 23 o 25 horas
- Años bisiestos: Febrero puede tener 28 o 29 días
- Diferentes calendarios: Gregoriano, Juliano, islámico, etc.
- Formatos variados: DD/MM/YYYY vs MM/DD/YYYY vs YYYY-MM-DD
- Localización: "Monday" en inglés, "Lunes" en español
- Precisión: Segundos, microsegundos, nanosegundos

Python proporciona el módulo datetime para manejar fechas y horas de forma robusta. Es parte de la librería estándar, no necesita instalación.

3.2 Casos de uso comunes

- Aplicaciones web: Timestamps de creación/modificación de registros.
- APIs: Logs con fecha/hora, expiración de tokens.
- Reportes: Filtrar datos por rango de fechas.
- Validación: Verificar que un usuario es mayor de edad.
- Scheduling: Tareas programadas, recordatorios.
- Analytics: Calcular tiempo transcurrido, patrones temporales.

3.3 Módulos principales: date, time, datetime

El módulo datetime contiene varias clases para trabajar con fechas y horas:

Clase	Contiene	Uso típico
date	Solo fecha (año, mes, día)	Fechas de nacimiento, vencimientos
time	Solo hora (hora, minuto, segundo)	Horarios, duraciones cortas
datetime	Fecha + hora completa	Timestamps, logs, registros
timedelta	Diferencia de tiempo	Sumar/restar días, horas
timezone	Información de zona horaria	Conversión UTC ↔ local

Importación

Para trabajar con fechas, importamos las clases que necesitemos:

```
# Importar clases específicas (recomendado)
from datetime import datetime, date, time, timedelta
# Importar todo el módulo (menos común)
import datetime
# Usar con prefijo
fecha = datetime.datetime.now()
```

Importante: Es recomendable importar clases específicas (from datetime import datetime) en lugar de todo el módulo, para evitar confusión entre el módulo y la clase.

3.4 Crear y Obtener Fechas

Fecha y hora actual

Para dar la fecha y hora actual:

```
from datetime import datetime


print("Fecha y hora actual:", datetime.now())
# Solo la fecha de hoy
from datetime import date
hoy = date.today()
print("Solo fecha:", hoy)
```

Crear fecha y hora específica

Para crear una fecha específica, pasamos año, mes y día (en ese orden):

```
from datetime import datetime


fecha_especifica = datetime(2026, 1, 6)
print("Fecha específica:", fecha_especifica)
from datetime import datetime
# Fecha con hora completa
fecha_hora = datetime(2026, 1, 6, 15, 30, 0)
print("Fecha y hora:", fecha_hora)
# Con microsegundos
fecha_precisa = datetime(2026, 1, 6, 15, 30, 0, 123456)
print("Fecha precisa:", fecha_precisa)
```

Importante: En Python el mes de enero es 1 (no 0 como en JavaScript). Los parámetros son: datetime(año, mes, día, hora, minuto, segundo, microsegundo).

3.4.1 Componentes individuales de una fecha

Podemos acceder a cada componente de la fecha:

```
from datetime import datetime


fecha = datetime(2025, 12, 8, 14, 30, 45)


# Acceder a componentes individuales
print(f"Año: {fecha.year}")
print(f"Mes: {fecha.month}")
print(f"Día: {fecha.day}")
```

```
print(f"Hora: {fecha.hour}")
print(f"Minuto: {fecha.minute}")
print(f"Segundo: {fecha.second}")

# Día de la semana (0=lunes, 6=domingo)
print(f"Día semana: {fecha.weekday()}")  # 0-6
print(f"Día semana ISO: {fecha.isoweekday()}")  # 1-7 (1=lunes)

# Número de día del año
print(f"Día del año: {fecha.timetuple().tm_yday}")
"""
**Salida esperada:**

Año: 2025
Mes: 12
Día: 8
Hora: 14
Minuto: 30
Segundo: 45
Día semana: 0
Día semana ISO: 1
Día del año: 342
"""
```

Nota:
weekday() devuelve 0=lunes, 6=domingo.
isoweekday() devuelve 1=lunes, 7=domingo.
Usa isoweekday() para más claridad (1-7 es más intuitivo que 0-6).

3.5 Formatear Fechas: strftime()

El método strftime() (string format time) convierte un objeto datetime a string con el formato que especifiquemos, puedes ver el formato en la documentación de Python aquí.

```
from datetime import datetime
fecha = datetime(2026, 1, 6, 15, 30, 0) # Formato europeo
print(fecha.strftime("%d/%m/%Y")) # 06/01/2026 # Formato con hora
print(fecha.strftime("%d/%m/%Y %H:%M:%S")) # 06/01/2026 15:30:00 # Formato ISO 8601 (estándar internacional)
print(fecha.strftime("%Y-%m-%d %H:%M:%S")) # 2026-01-06 15:30:00 # Formato personalizado
print(fecha.strftime("Hoy es %A, %d de %B de %Y")) # Hoy es Tuesday, 06 de January de 2026
```

Códigos de formato principales

Código	Significado	Ejemplo
%Y	Año con siglo	2026
%y	Año sin siglo	26
%m	Mes (01-12)	01
%d	Día del mes (01-31)	06
%H	Hora 24h (00-23)	15
%I	Hora 12h (01-12)	03
%M	Minuto (00-59)	30
%S	Segundo (00-59)	45
%p	AM/PM	PM
%A	Nombre día completo	Tuesday
%a	Nombre día abreviado	Tue
%B	Nombre mes completo	January
%b	Nombre mes abreviado	Jan

3.5.1 Localización: Fechas en español

Por defecto, los nombres de días y meses aparecen en inglés. Para cambiarlos a español:

```
import locale
from datetime import datetime
# Configurar localización en español
locale.setlocale(locale.LC_TIME, 'es_ES.UTF-8')
fecha = datetime.now()
print(fecha.strftime("%A, %d de %B de %Y"))
print(fecha.strftime("%A %d/%m/%Y a las %H:%M"))
```

3.6 Parsear Fechas: strptime()

El método strptime() (string parse time) es la operación inversa a strftime(): convierte un string a objeto datetime:

```
from datetime import datetime
# String con fecha
fecha_string = "06/01/2026 15:30:00"
# Convertir a datetime
fecha_objeto = datetime.strptime(fecha_string, "%d/%m/%Y %H:%M:%S")
print(type(fecha_objeto)) # <class 'datetime.datetime'>
print(fecha_objeto) # 2026-01-06 15:30:00
# Ahora podemos hacer operaciones con la fecha
print(fecha_objeto.year) # 2026
print(fecha_objeto.month) # 1
```

Casos comunes de parseo de fechas

```
import locale
from datetime import datetime
```

```
# Formato europeo
locale.setlocale(locale.LC_TIME, 'es_ES.UTF-8')
fecha1 = datetime.strptime("15/03/2025", "%d/%m/%Y")
# Formato americano
fecha2 = datetime.strptime("03/15/2025", "%m/%d/%Y")
# Formato ISO 8601
fecha3 = datetime.strptime("2025-03-15", "%Y-%m-%d") # Con hora
fecha4 = datetime.strptime("2025-03-15 14:30", "%Y-%m-%d %H:%M")
# Formato largo
fecha5 = datetime.strptime("15 de marzo de 2025", "%d de %B de %Y")
```

Importante: El formato en strptime() DEBE coincidir EXACTAMENTE con el string. Si el string es '15/03/2025' pero usas '%Y-%m-%d', lanzará ValueError.

3.6.1 Manejo de errores en parseo

```
from datetime import datetime
def parsear_fecha_segura(fecha_string, formato):
    """
    Parsea fecha con manejo de errores
    """
    try:
        return datetime.strptime(fecha_string, formato)
    except ValueError as e:
        print(f"Error: formato incorrecto - {e}")
        return None
# Uso
fecha = parsear_fecha_segura("15/13/2025", "%d/%m/%Y") # Mes 13 inválido
if fecha:
    (f"Fecha válida: {fecha}")
else: print("Fecha no pudo ser parseada")
```

3.7 Operaciones con fecha: timedelta

timedelta representa una diferencia de tiempo. Se usa para sumar o restar tiempo a fechas.
Puedo obtener la fecha de ayer usando timedelta:

```
from datetime import datetime, timedelta


ayer = datetime.now() - timedelta(1)
print(ayer.strftime("%d/%m/%Y %H:%M:%S"))
```

También puedo quitarle días, horas, minutos…

```
ayer = datetime.now() - timedelta(days=7, hours=3)
```

```
print(ayer.strftime("%d/%m/%Y %H:%M:%S"))
```

Ejemplo:

```
from datetime import datetime, timedelta
# Fecha actual
ahora = datetime.now()
print("Ahora:", ahora.strftime("%d/%m/%Y %H:%M"))
# Ayer (restar 1 día)
ayer = ahora - timedelta(days=1)
print("Ayer:", ayer.strftime("%d/%m/%Y %H:%M"))
#Mañana (sumar 1 día)
manana = ahora + timedelta(days=1)
print("Mañana:", manana.strftime("%d/%m/%Y %H:%M")) # Hace una semana
semana_pasada = ahora - timedelta(weeks=1)
print("Hace una semana:", semana_pasada.strftime("%d/%m/%Y"))
```

3.8 Comparación de fechas

Las fechas se pueden comparar directamente con operadores de comparación:

```
from datetime import datetime
fecha1 = datetime(2025, 1, 15)
fecha2 = datetime(2025, 6, 30)
fecha3 = datetime(2025, 1, 15)
print(fecha1 < fecha2) # True
print(fecha1 > fecha2) # False
print(fecha1 == fecha3) # True
print(fecha1 != fecha2) # True
# Verificar si una fecha está en un rango
fecha_actual = datetime.now()
inicio_verano = datetime(2025, 6, 21)
fin_verano = datetime(2025,9, 21)
if inicio_verano <= fecha_actual <= fin_verano:
    print("Estamos en verano")
else:
    print("No estamos en verano")
```

Al restar dos fechas devuelve un objeto timedelta. Solamente hay que restar las instancias de 2 objetos fecha.

```
fecha1 = datetime.now()
fecha2 = datetime(2025, 6, 1)
diferencia = fecha2 - fecha1
print("Días restantes para el 1 de junio de 2025:", diferencia.days)
```

```
print("Segundos totales:", diferencia.total_seconds())
```

Ejemplo: comprobar si es mayor de edad

```
from datetime import datetime
# Calcular edad
def calcular_edad(fecha_nacimiento):
    hoy = datetime.now()
    edad = hoy.year - fecha_nacimiento.year # Ajustar si aún no cumplió años este año
    if hoy.month < fecha_nacimiento.month or (hoy.month == fecha_nacimiento.month and hoy.day <
fecha_nacimiento.day):
        edad -= 1
    return edad

nacimiento = datetime(1990, 5, 15)
edad = calcular_edad(nacimiento)
print(f"Edad: {edad} años") # Verificar si es mayor de edad
if edad >= 18:
    print("Usuario mayor de edad")
```

Ejercicio 1

Crea una función es_fin_de_semana que reciba una fecha y devuelva true si es fin de semana. Otra función es_dia_laborable que reciba una fecha y devuelva true si es día laborable.

```
#Prueba
hoy = datetime.now()
if es_fin_de_semana(hoy):
    print("¡Es fin de semana!")
else:
    print("Es día laborable")
```

Ejercicio 2

Crea una función llamada contar_dias_laborales, con parámetros de entrada fecha_inicio y fecha_fin que cuente los días laborales (no fin de semana) entre dos fechas.

```
#Prueba
inicio = datetime(2025, 12, 1)
fin = datetime(2025, 12, 31)
dias_laborables = contar_dias_laborables(inicio, fin)
print(f"Días laborables en diciembre: {dias_laborables}")
```

Ejercicio 3 - Calcular Edad

Escribe una función calcular_edad(fecha_nacimiento) que:

1. Reciba una fecha de nacimiento como string en formato DD/MM/YYYY
2. La convierta a objeto datetime
3. Calcule la edad actual en años
4. Devuelva un mensaje: "Tienes X años"
5. Indique si es mayor o menor de edad

```
# Pruebas
print(calcular_edad("15/05/1990"))
print(calcular_edad("01/01/2010"))
print(calcular_edad("20/12/2024"))  # Recién nacido casi
```

Ejercicio 4 – Dias hasta Enero
Crea un programa que:
1. Pida al usuario una fecha de evento (DD/MM/YYYY)
2. Calcule cuántos días faltan para ese evento
3. Si el evento ya pasó, mostrar "El evento ya ocurrió hace X días"
4. Si el evento es hoy, mostrar "¡El evento es hoy!"
5. Si el evento es en el futuro, mostrar "Faltan X días para el evento"

Ejercicio 5 – Validador de reservas
Crea un sistema de validación de reservas de hotel:
1. Función validar_reserva(fecha_entrada, fecha_salida)
2. Verificar que fecha_entrada sea futura (al menos mañana)
3. Verificar que fecha_salida sea posterior a fecha_entrada
4. Verificar que la estancia sea mínimo 1 noche, máximo 30 noches
5. Calcular número de noches y precio (70€/noche entre semana, 100€/noche fin de semana)
6. Devolver diccionario con: válido (bool), mensaje (str), noches (int), precio_total (float)

```
# Pruebas
print(validar_reserva("07/12/2025", "10/12/2025"))  # Válida
print(validar_reserva("05/12/2025", "06/12/2025"))  # Entrada debe ser mañana
print(validar_reserva("10/12/2025", "09/12/2025"))  # Salida antes de entrada
print(validar_reserva("15/12/2025", "20/01/2026"))  # Más de 30 noches
```

4. Ficheros

4.1 Introducción - ¿ Por qué necesitamos ficheros?

Cuando ejecutamos un programa Python, toda la información que manejamos (variables, listas, diccionarios) se almacena en la memoria RAM. El problema es que la RAM es volátil: cuando el programa termina o apagamos el ordenador, todos los datos se pierden.

Concepto clave: persistencia de datos:
Los ficheros nos permiten guardar información de forma permanente en el disco duro, de manera que podamos recuperarla más tarde, incluso después de reiniciar el sistema.

Ejemplos reales de uso de ficheros:
• Configuración: Guardar preferencias del usuario (idioma, tema, conexiones BD).

• Logs: Registrar eventos de la aplicación para depuración.
• Datos de usuario: Almacenar información (perfiles, historial, caché).
• Exportación: Generar informes CSV, JSON para otras aplicaciones.
• Backups: Copiar datos importantes para recuperación.

Nota: En aplicaciones web profesionales, aunque uses bases de datos para datos estructurados, SIEMPRE necesitarás ficheros para logs, configuración, archivos subidos por usuarios, exportaciones, etc.

4.2 El objeto Path (pathlib)

Antiguamente en Python se trabajaba con rutas como strings ('datos/archivo.txt'). El problema es que esto era propenso a errores: barras invertidas en Windows vs Linux, concatenación manual, verificar existencia con funciones separadas, etc.

Concepto clave: pathlib - El estándar moderno:
Desde Python 3.4, el módulo pathlib permite trabajar con rutas como OBJETOS. Un objeto Path tiene métodos y propiedades que hacen el código más legible, seguro y multiplataforma…

4.2.1 Crear objetos Path

Para crear un objeto Path, simplemente pasamos la ruta como string al constructor. Podemos crear rutas relativas (desde el directorio actual) o absolutas (desde la raíz del sistema).

```
from pathlib import Path
# Ruta relativa (desde donde se ejecuta el script)
fichero = Path("datos.txt")
print(fichero) # datos.txt
# Ruta relativa con subdirectorios
informe = Path("reportes/2025/enero.pdf")
print(informe) # reportes/2025/enero.pdf
# Ruta absoluta (ruta completa desde raíz)
config = Path("/etc/mi_app/config.json")
print(config) # /etc/mi_app/config.json
```

¿Por qué es mejor usar el operador / ?
Path permite "dividir" rutas con el operador /. Esto es MUCHO más legible que concatenar strings y funciona en Windows, Linux y Mac automáticamente.

```
# Forma antigua (strings) - ERROR en Windows
ruta_mala = "datos" + "/" + "usuarios" + "/" + "perfil.json"
# Forma moderna (Path) - Funciona en todos los OS
directorio = Path("datos")
subdirectorio = directorio / "usuarios"
archivo = subdirectorio / "perfil.json"
```

```
print(archivo) # datos/usuarios/perfil.json
# También se puede encadenar
ruta_completa = Path("datos") / "usuarios" / "perfil.json"
print(ruta_completa) # datos/usuarios/perfil.json
```

Nota: El operador / detecta automáticamente el separador correcto: / en Linux/Mac, \\ en Windows. Tu código funcionará en cualquier sistema operativo sin cambios.

Actividad

Crea un archivo pi_digitos.txt con este contenido:

3.1415926535
8979323846
2643383279

El primer paso es importar la clase Path e instanciar un objeto que represente la ruta del fichero con el que queremos trabajar.

```
from pathlib import Path

# Ejemplo: Instanciamos un objeto Path. Puede ser ruta relativa o absoluta.
fichero = Path("pi_digitos.txt")

# Manejo de rutas multiplataforma:
# pathlib corrige automáticamente las barras.
ruta = Path("datos/informes/archivo.txt")
# En Windows, internamente será 'datos\\informes\\archivo.txt'
```

Instanciamos un objeto Path. La ruta puede ser relativa (como aquí) o absoluta.
El operador / es la forma recomendada para unir rutas en pathlib:

ruta_unida = Path('datos') / 'informes' / 'archivo.txt'

4.2.2 Propiedades del objeto Path

Un objeto Path no es solo un string: tiene propiedades y métodos que nos dan información sobre el archivo o directorio. Veamos las más importantes:

.exists(): Devuelve True si el fichero o directorio existe.
.is_file(): Devuelve True si la ruta apunta a un fichero.
.is_dir(): Devuelve True si la ruta apunta a un directorio.
.name: Devuelve el nombre del fichero con extensión (e.g., 'pi_digitos.txt').
.stem: Devuelve el nombre del fichero sin extensión (e.g., 'pi_digitos').
.suffix: Devuelve la extensión (e.g., '.txt').
.parent: Devuelve el Path(ruta) del directorio padre.
.absolute(): Devuelve la ruta absoluta del fichero.

Ejemplo:

```
fichero = Path("documentos/informe_2025.pdf")
print(f"Nombre: {fichero.name}") #informe_2025.pdf
print(f"Sin ext: {fichero.stem}") # informe_2025
```

```
print(f"Extensión: {fichero.suffix}") # .pdf
print(f"Directorio: {fichero.parent}") # documentos
print(f"Existe: {fichero.exists()}") # True/False
print(f"¿Es archivo?: {fichero.is_file()}")
print(f"¿Es directorio?: {fichero.is_dir()}")
```

Importante: Los métodos is_file() e is_dir() solo devuelven True si el archivo/directorio EXISTE físicamente. Si el Path apunta a algo que no existe, ambos devolverán False.

4.3 Manejo de rutas relativas y absolutas

Entender cómo funcionan las rutas es fundamental para escribir código que funcione en cualquier entorno o sistema operativo.

Ruta absoluta

Una ruta absoluta es la ruta completa de un fichero o directorio que comienza desde el directorio raíz del sistema (el origen). Es única e independiente de dónde se ejecute el script.

Ejemplos:

- En Linux/macOS: /home/usuario/documentos/reporte.pdf
- En Windows: C:\Users\Usuario\Documentos\reporte.pdf

Uso con pathlib:

- Path.cwd(): Devuelve la ruta absoluta del Directorio de Trabajo Actual (CWD).
- Path('fichero.txt').resolve(): Convierte una ruta relativa en su equivalente absoluto, pero tieniendo en cuenta ..(directorio padre) y .(directorio actual).

```
from pathlib import Path

# Estructura de carpetas:
# /proyecto
#   ├── src
#   │   └── main.py       <- Estamos aquí
#   └── data
#       └── datos.json

# Desde main.py, quiero acceder a datos.json
archivo = Path("../data/datos.json")
# .. = sube a /proyecto
# data/datos.json = baja a /proyecto/data/datos.json

print(archivo.absolute())
# MAL /proyecto/src/../data/datos.json  (con ..)

print(archivo.resolve())
# /proyecto/data/datos.json  (sin .., limpio)
```

Ruta relativa

Una ruta relativa es la ruta de un fichero o directorio definida en relación al Directorio de Trabajo Actual (CWD) desde donde se lanzó el script de Python.

Concepto clave - CWD - Current Working Directory:
El CWD es el directorio desde donde ejecutas el script. Si ejecutas 'python app/main.py' desde /home/usuario/, el CWD es /home/usuario/, NO /home/usuario/app/.

Ejemplos:
./datos/informe.csv: Busca la carpeta datos dentro del CWD.
../logs/errores.txt: Sube un nivel de directorio y busca la carpeta logs.

Buenas prácticas

La buena práctica profesional dicta evitar las rutas absolutas, ya que rompen la portabilidad.
La mejor estrategia es:
- Usar rutas relativas para ficheros dentro del proyecto.
- Usar la variable mágica Path(__file__).parent para obtener la ruta absoluta del directorio que contiene el script actual, y construir rutas relativas a partir de ahí.

```python
from pathlib import Path


# Obtener la ruta del directorio donde está ESTE script
directorio_base = Path(__file__).parent


# Construir una ruta absoluta para el recurso 'data.csv' dentro de la carpeta 'recursos'
ruta_recurso = directorio_base / 'recursos' / 'data.csv'


print(f"Ruta Absoluta del Recurso: {ruta_recurso}")
```

4.3.1 Path(__file__) - Rutas relativas al script

El problema con rutas relativas simples es que dependen de DÓNDE ejecutas el script (no donde está el propio .py). La variable __file__ contiene la ruta del archivo .py actual, lo que permite crear rutas relativas AL SCRIPT, no al CWD.

```
# Estructura:
/home/usuario/
  ├── app/
  │   ├── main.py
  │   └── datos/
  │       └── config.json
  └── otros_archivos/


# Si ejecutas desde /home/usuario/:
cd /home/usuario
python app/main.py
```

```
# CWD = /home/usuario/  (donde ejecutaste)
# Script = /home/usuario/app/main.py  (donde está el script)
```

Si en main.py haces:

```
# INCORRECTO
Path("datos/config.json")  # Busca en /home/usuario/datos/config.json (NO existe)
```

Queremos:

```
# CORRECTO
# Buscar en /home/usuario/app/datos/config.json (SÍ existe)
```

La solución es usar __file__ que contiene la ruta del script SIEMPRE.

```
from pathlib import Path

# Obtener el directorio donde está main.py
SCRIPT_DIR = Path(__file__).parent
print(f"Script está en: {SCRIPT_DIR}")
# /home/usuario/app/

# Ahora construir rutas relativas al script
archivo_config = SCRIPT_DIR / "datos" / "config.json"
print(f"Config en: {archivo_config}")
# /home/usuario/app/datos/config.json

# Guardar archivo en app/datos/
with open(archivo_config, 'w') as f:
    f.write("datos")
```

Normalmente se definen las rutas importantes al inicio del script:

```
from pathlib import Path

# Al inicio del script, definir todas las rutas importantes
BASE_DIR = Path(__file__).parent  # /home/usuario/app/
DATA_DIR = BASE_DIR / "datos"             # /home/usuario/app/datos/
CONFIG_FILE = DATA_DIR / "config.json"    # /home/usuario/app/datos/config.json

# Ahora usar en el resto del código
def guardar_datos():
    with open(CONFIG_FILE, 'w') as f:
        f.write("datos")
    print(f"Guardado en: {CONFIG_FILE}")

def cargar_datos():
```

```
    with open(CONFIG_FILE, 'r') as f:
        return f.read()
```

Importante: NUNCA uses rutas absolutas escritas a mano tipo '/home/miguel/proyecto/config.json'. Tu código no funcionará en otros ordenadores. Usa SIEMPRE Path(__file__).parent.

4.4 Script y variables reutilizables

4.4.1 if __name__ == '__main__'

En Python, un mismo archivo .py puede funcionar de DOS formas:
1. Como SCRIPT ejecutable: python mi_archivo.py
2. Como MÓDULO importable: import mi_archivo (desde otro script)
La variable especial __name__ permite distinguir entre ambos casos.

Concepto clave - ¿Qué es __name__?:
__name__ es una variable que Python asigna automáticamente a cada archivo .py:
• Si EJECUTAS el archivo directamente: __name__ = '__main__'
• Si IMPORTAS el archivo desde otro: __name__ = 'nombre_del_archivo'

```
# archivo: utilidades.py
print(f"Valor de __name__: {__name__}")
def procesar_datos():
    print("Procesando datos...")
    return "Resultado"
# Este código SOLO se ejecuta si ejecutas el archivo directamente
if __name__ == "__main__":
    print("Ejecutando como script principal")
    resultado = procesar_datos()
    print(f"Resultado: {resultado}")
```

Caso 1 – Ejecutar directamente
Salida:

```
Valor de __name__: __main__
Ejecutando como script principal
Procesando datos...
Resultado: Resultado
```

Caso 2 – Importar desde otro archivo
Salida:

```
Valor de __name__: utilidades
Procesando datos...
```

Nota: ¿Ves la diferencia? Al importar, NO se ejecutó el bloque if __name__ == '__main__'. Solo se ejecutó al correr el archivo directamente.

4.4.2 Patrón profesional con main()

La forma MÁS PROFESIONAL es definir una función main() que contiene la lógica principal, y llamarla solo si el archivo se ejecuta directamente. Esto hace el código más limpio y testeable.

```
from pathlib import Path
def procesar_logs(directorio):
    """Función reutilizable - se puede importar"""
    archivos = list(Path(directorio).glob("*.log"))
    print(f"Encontrados {len(archivos)} archivos log")
    return archivos
def limpiar_cache():
    """Otra función reutilizable"""
    caches = list(Path(".").glob("cache_*.json"))
    for cache in caches:
        cache.unlink()
        print(f"Eliminado: {cache.name}")
def main():
    """Función principal del programa"""
    print("=== Iniciando aplicación ===")
    procesar_logs("logs")
    limpiar_cache()
    print("=== Finalizado ===")
# Solo ejecuta main() si el archivo se corre directamente
if __name__ == "__main__":
    main()
```

Nota: Este patrón te permite: (1) Importar las funciones desde tests, (2) Reutilizar funciones en otros scripts, (3) Incluir ejemplos de uso sin que se ejecuten al importar.

4.4.3 Otras variables especiales

Python tiene más variables especiales que empiezan y terminan con __ (double underscore). Son CONVENCIONES profesionales para documentar módulos.

```
# Variables especiales del módulo
__version__ = "1.0.0"
__author__ = "Miguel García"
__all__ = ["procesar_datos", "validar_email"] # Qué exportar con import *
def procesar_datos():
    """Procesa datos de entrada"""
    pass
def validar_email(email):
    """Valida formato de email"""
```

```
    return "@" in email
def _funcion_interna():
    """Función privada (por convención, no se exporta)"""
    pass
# __doc__ contiene el docstring
print(__doc__) # None (si no hay docstring al inicio del archivo)
print(validar_email.__doc__) # Valida formato de email
# __file__ contiene la ruta del archivo
print(__file__) # /home/usuario/proyecto/utils.py
```

4.5 Buscar archivos

Una tarea muy común es buscar archivos que cumplan cierto criterio: todos los PDFs, archivos que empiecen por "log_", JSONs de un año concreto, etc. Path proporciona tres métodos esenciales para esto.

4.5.1 glob() - Búsqueda con patrones

Concepto clave - Patrones glob:
Los patrones glob son como 'comodines' para buscar archivos:
- * = cualquier cosa (cero o más caracteres)
- ? = exactamente un carácter
- [abc] = uno de estos caracteres
- ** = cualquier nivel de subdirectorios (solo con rglob)

El método glob(patrón) busca archivos en el directorio actual que coincidan con el patrón. Devuelve un generador (lo convertimos a lista con list()).

```
from pathlib import Path
# Buscar TODOS los archivos .txt en el directorio actual
archivos_txt = list(Path(".").glob("*.txt"))
print(f"Encontrados {len(archivos_txt)} archivos .txt")
for archivo in archivos_txt:
print(f" → {archivo.name}")
# Buscar archivos con patrón específico
informes_2025 = list(Path("reportes").glob("informe_2025_*.pdf"))
# Buscar en subdirectorio específico
json_datos = list(Path("datos").glob("*.json"))
# Patrón con un carácter específico
logs_enero = list(Path("logs").glob("log_2025_01_??.txt"))
# Encuentra: log_2025_01_01.txt, log_2025_01_15.txt, etc.
```

Importante: glob() solo busca en el directorio especificado, NO en subdirectorios. Para búsqueda recursiva usa rglob().

4.5.2 rglob() - Búsqueda recursiva

Cuando necesitas buscar en TODOS los subdirectorios (recursivamente), usa rglob(). Es perfecta para buscar un archivo en todo el proyecto sin saber exactamente dónde está.

```
from pathlib import Path
# Buscar TODOS los .py en el proyecto completo (recursivo)
archivos_python = list(Path(".").rglob("*.py"))
print(f"Total de archivos Python: {len(archivos_python)}")
# Buscar todos los JSON sin importar dónde estén
todos_json = list(Path(".").rglob("*.json"))
print(f"\nArchivos JSON encontrados:")
for json_file in todos_json:
print(f" → {json_file}")
# Buscar archivos de caché en cualquier ubicación
caches = list(Path(".").rglob("cache_*.json"))
print(f"\nCachés encontrados: {len(caches)}")
```

Nota: rglob() es muy potente pero puede ser lenta en directorios muy grandes (ej: node_modules). Úsala solo cuando realmente necesites buscar recursivamente.

4.5.3 iterdir() - Listar contenido de un directorio

A veces no necesitas buscar por patrón, sino simplemente listar todo lo que hay en un directorio específico. Para eso está iterdir().

```
from pathlib import Path
# Listar TODO lo que hay en un directorio
directorio = Path("proyectos")
print("Contenido de 'proyectos':")
for item in directorio.iterdir():
    if item.is_file():
        print(f" Archivo: {item.name}")
    elif item.is_dir():
        print(f" Carpeta: {item.name}")
# Separar archivos y carpetas en listas
archivos = [f for f in directorio.iterdir() if f.is_file()]
carpetas = [d for d in directorio.iterdir() if d.is_dir()]
print(f"\nTotal: {len(archivos)} archivos, {len(carpetas)} carpetas")
```

4.6 Lectura de ficheros

La lectura de ficheros es la operación más común. Python ofrece varias formas de leer archivos dependiendo de tus necesidades: leer todo de golpe, leer línea por línea, leer binarios, etc.

4.6.1 read_text() - Lectura completa (método simple)

El método read_text() es la forma MÁS SIMPLE de leer un archivo de texto. Lee TODO el contenido del archivo y lo devuelve como un string.

```
from pathlib import Path


fichero = Path("pi_digitos.txt")
contenido = fichero.read_text()
print(contenido)
```

A la hora de leer el fichero, es conveniente (VSCode seguramente te lo esté diciendo, subrayando fichero.read_text() con una línea en naranja) poner la codificación del fichero.

```
contenido = fichero.read_text(encoding="utf-8")
```

Si en el fichero ponemos caracteres en blanco después de los números, se nos mostrarán dichos caracteres al hacer el print, esto lo podemos eliminar usando “rstrip”, que también elimina los \n de fin de línea:

```
contenido = fichero.read_text(encoding="utf-8")
print(contenido.rstrip())
```

Ejemplo completo:

```
from pathlib import Path
fichero = Path("pi_digitos.txt")
# SIEMPRE especificar la codificación, siendo UTF-8 el estándar.
try:
    contenido = fichero.read_text(encoding="utf-8")
    # .rstrip() elimina los saltos de línea (\n) o espacios al final de la cadena
    print(contenido.rstrip())
except FileNotFoundError:
    print(f"Error: El fichero {fichero.name} no existe.")
```

Otro ejemplo completo:

```
from pathlib import Path
fichero = Path("datos/mensaje.txt")
try:
    # Leer TODO el contenido
    contenido = fichero.read_text(encoding="utf-8")
    print("Contenido del archivo:")
    print(contenido)
    # Limpiar espacios/saltos de línea finales
    contenido_limpio = contenido.rstrip()
    print(f"\nLongitud: {len(contenido_limpio)} caracteres")
except FileNotFoundError:
    print(f"■ Error: El archivo '{fichero.name}' no existe")
```

Importante: SIEMPRE especifica encoding='utf-8'. Si no lo pones, Python usa el encoding por defecto del sistema, que puede variar entre Windows/Linux y causar errores con tildes y símbolos.

4.6.2 splitlines()) – Leer línea por línea

Cuando se trabaja con un fichero, a menudo queremos examinar cada línea del fichero, ya sea para buscar algo o modificar las líneas que cumplan cierta condición, por ejemplo, al trabajar con un fichero de datos del tiempo, si solo queremos las líneas donde aparezca la palabra "soleado", habrá que recorrer todas las líneas y quedarnos solo con las que tengan la palabra "soleado". Esto se hace usando "splitlines", que transforma las líneas de un fichero en una lista de líneas, donde cada valor de la lista es una línea del fichero, por lo que podemos usar un "for in" para examinar dichas líneas.

```
fichero =  Path("pi_digitos.txt")
contenido = fichero.read_text(encoding="utf-8").rstrip()
lineas = contenido.splitlines()
for linea in lineas:
    print(linea + "#")
```

Si queremos también el número de las líneas, utilizar "enumerate".

```
fichero =  Path("pi_digitos.txt")
contenido = fichero.read_text(encoding="utf-8").rstrip()
lineas = contenido.splitlines()
for num, linea in enumerate(lineas):
    print(f"Línea {num + 1}: {linea}")
print(f"Número total de líneas: {len(lineas)}")
```

Importante: Python lee TODO como strings. Si el archivo contiene números y quieres usarlos en cálculos, debes convertirlos: numero = int(linea) o precio = float(linea).

4.6.3 read_bytes() - Ficheros binarios

No todos los archivos son texto. Las imágenes, PDFs, ejecutables, etc. son archivos binarios. Para leerlos usa read_bytes(), que devuelve los datos como bytes en lugar de string.

```
from pathlib import Path
# Leer archivo binario (imagen)
imagen = Path("fotos/perfil.jpg")
if imagen.exists():
    datos_binarios = imagen.read_bytes()
    print(f"Tamaño del archivo: {len(datos_binarios)} bytes")
    print(f"Primeros 10 bytes: {datos_binarios[:10]}")
    print(f"Tipo de datos: {type(datos_binarios)}")
else:
    print("La imagen no existe")
```

Nota: Los archivos binarios no se pueden leer con read_text(). Si intentas hacerlo, obtendrás un UnicodeDecodeError. Usa read_bytes() para imágenes, PDFs, ZIP, etc.

Ejercicio 1
El fichero que hemos creado más arriba (pi_digitos.txt) tiene los dígitos de pi en varias líneas, utiliza print para mostrar los dígitos en 1 sola línea.

4.7 Escritura de ficheros

Escribir en ficheros es tan importante como leer. Podemos guardar resultados de cálculos, configuraciones, logs, exportaciones, etc.

4.7.1 write_text() - Escritura simple

El método write_text() es la forma más simple de escribir en un archivo.
IMPORTANTE: Si el archivo existe, lo SOBRESCRIBE completamente. Si no existe, lo crea.
Igual que read_text, primero hay que especificar la ruta del fichero. Ejemplo:

```
from pathlib import Path


fichero = Path("nuevo_fichero.txt")
fichero.write_text("Nueva línea")
```

Ahora si queremos sobrescribir dicho fichero con nuevos datos y comprobar su existencia.

```
from pathlib import Path


fichero = Path("nuevo_fichero.txt")
texto_a_escribir = """
    Este es un nuevo fichero creado para el ejercicio.
    Contiene varias líneas de texto.
    Cada línea será escrita en el fichero.
    """
fichero.write_text(texto_a_escribir, encoding="utf-8")
print(f"Fichero '{fichero.name}' creado y texto escrito correctamente.")


#Comprobar existencia del fichero
if fichero.exists():
    print(f"El fichero existe: {fichero.exists()}")
    # Verificar el contenido
    verificacion = fichero.read_text(encoding="utf-8")
        print(f"\nContenido guardado:\n{verificacion}")
```

Importante: write_text() BORRA todo el contenido anterior. Si quieres AÑADIR al final sin borrar, usa el modo 'append' con open('a').

4.7.2 write_bytes() - Escritura binaria

Para escribir datos binarios (no texto), usa write_bytes(). Esto es útil cuando copias archivos binarios o generas datos binarios.

```
from pathlib import Path
# Crear datos binarios (ejemplo)
datos_binarios = b'\x48\x65\x6c\x6c\x6f' # "Hello" en bytes
# Escribir archivo binario
archivo_bin = Path("datos.bin")
archivo_bin.write_bytes(datos_binarios)
print(f"■ Archivo binario creado: {len(datos_binarios)} bytes")
```

4.7.3 Modo Append - Añadir sin borrar

A veces NO quieres sobrescribir, sino añadir contenido al final del archivo existente. Para esto necesitas usar open() en modo 'a' (append). Esto es ESENCIAL para logs. Aunque se puede añadir contenido a un fichero concatenando el contenido nuevo al contenido del fichero de esta forma:

```
from pathlib import Path

# 1. Definir la ruta del fichero
fichero = Path('nuevo_fichero.txt')

# 2. Contenido inicial (Aseguramos que el fichero existe)
fichero.write_text("Registro inicial de la prueba.\n", encoding='utf-8')
print(f"Contenido inicial:\n{fichero.read_text(encoding='utf-8')}")

# Añadir nuevo contenido
nuevo_contenido = "Datos adicionales sin usar. \n"

# 1. Leer el contenido actual del fichero.
# Se lee todo el texto existente del fichero.
contenido_actual = fichero.read_text(encoding='utf-8')

# 2. Concatenar el contenido nuevo.
# Concatenamos el texto existente con el nuevo.
contenido_combinado = contenido_actual + nuevo_contenido

# 3. Sobrescribir (escribir de nuevo) el fichero completo.
# .write_text() sobrescribe el fichero, incluyendo ahora el texto antiguo + el nuevo.
fichero.write_text(contenido_combinado, encoding='utf-8')
```

```
print("---")
print(f"Contenido después de anexar:\n{fichero.read_text(encoding='utf-8')}")
```

Aunque .read_text() y .write_text() son convenientes para textos pequeños, el bloque with open() es el mecanismo fundamental y más eficiente de Python para el manejo de ficheros, especialmente para:

1. Añadir contenido ('a', append) sin cargar todo el fichero en memoria.
2. Lectura/Escritura eficiente línea por línea (para ficheros grandes).
3. Asegurar el cierre del recurso, incluso si ocurre un error.

Concepto clave - Gestor de contexto (with):
La palabra clave 'with' garantiza que el archivo se cierre automáticamente al salir del bloque, incluso si hay un error. Esto previene fugas de recursos y datos corruptos.

Modos de apertura (mode)

Al usar open(), debes especificar el modo de apertura. Cada modo determina qué puedes hacer con el archivo y qué pasa si existe o no existe.

Modo	Nombre	Descripción	Si Existe	Si No Existe
'r'	Read	Solo lectura (por defecto)	Lee desde el inicio.	Error (FileNotFoundError).
'w'	Write	Solo escritura	Borra el contenido.	Crea el fichero.
'a'	Append	Añadir	Escribe al final (sin borrar).	Crea el fichero.
'r+'	Read + Write	Lectura y Escritura	Permite leer y escribir.	Error.
'wb'	Write Binary	Escritura de binarios	Borra el contenido.	Crea el fichero.

Importante: Modo 'w' es PELIGROSO: borra TODO el contenido del archivo en cuanto lo abres, incluso si no escribes nada. Usa 'a' para añadir sin borrar.

Añadir contenido("a")

El modo 'a' (append) permite añadir contenido al final de un fichero existente sin borrar el contenido anterior. Esta es la forma más eficiente de añadir datos, ya que no carga todo el archivo en memoria.

```
from pathlib import Path
fichero = Path('registro_append.txt')

# Aseguramos un contenido inicial para la prueba
fichero.write_text("--- Inicio de Registro ---\n", encoding='utf-8')

# Utilizamos .open() con el modo 'a' (append)
with fichero.open('a', encoding='utf-8') as f:
```

```
    # f es un objeto archivo tradicional
    f.write("Esta línea se añade al final (1).\n")
    f.write("Otra línea añadida (2).\n")

print("Contenido añadido exitosamente usando 'a'.")
# Verificar resultado
contenido_final = fichero.read_text(encoding="utf-8")
print("Contenido final del archivo:")
print(contenido_final)
```

Nota: El modo 'a' (append) es MUCHO más eficiente que leer todo, concatenar y volver a escribir. Además, evita perder datos si hay un error durante la escritura.

Lectura Eficiente Línea por Línea (Iteración)

Para archivos MUY grandes (varios GB), NO uses read_text() porque carga todo en memoria. Usa un bucle for directamente sobre el archivo abierto: procesa línea por línea bajo demanda.

```
from pathlib import Path
fichero = Path('nuevo_fichero.txt')

print("\n-- Lectura Eficiente por Iteración ---")
# Abrimos en modo lectura 'r'.
with fichero.open('r', encoding='utf-8') as f:
    # La iteración carga cada línea bajo demanda.
    for linea in f:
        # El método .strip() es esencial para remover el salto de línea (\n)
        # que queda al final de cada línea leída por iteración.
        print(f"--> {linea.strip()}")
```

Nota: Este método es ESENCIAL para procesar logs de producción que pueden tener millones de líneas. Solo carga en memoria la línea actual, no todo el archivo.¡

4.8 Operaciones con archivos y directorios

Además de leer y escribir, necesitas poder crear, eliminar, renombrar y mover archivos y directorios. Path proporciona métodos para todas estas operaciones.

4.8.1 Crear y eliminar archivos

El método touch() crea un archivo vacío. El método unlink() elimina un archivo. Ambos son útiles para inicializar archivos o limpiar temporales.

```
from pathlib import Path
# Crear archivo vacío
```

```
archivo = Path("nuevo.txt")
archivo.touch() # Crea si no existe, no hace nada si ya existe
print(f"Archivo creado: {archivo.exists()}")
# Eliminar archivo
archivo.unlink() # Error si no existe
print("Archivo eliminado")
# Eliminar solo si existe (forma segura)
if archivo.exists():
    archivo.unlink()
    print("Archivo eliminado")
# O usar parámetro (Python 3.8+)
archivo.unlink(missing_ok=True) # No da error si no existe
print("Operación completada")
```

Importante: unlink() sin missing_ok=True lanza FileNotFoundError si el archivo no existe. Usa missing_ok=True para evitar errores.

4.8.2 Crear y eliminar directorios

El método mkdir() crea directorios. Puede crear directorios anidados con el parámetro parents=True. El método rmdir() elimina directorios VACÍOS.

```
from pathlib import Path
# Crear directorio simple
directorio = Path("nueva_carpeta")
directorio.mkdir() # Error si ya existe
# No error si ya existe
directorio.mkdir(exist_ok=True)
print("Directorio creado")
# Crear directorios anidados (crea todos los niveles)
Path("datos/2025/enero").mkdir(parents=True, exist_ok=True)
print("Estructura completa creada")
# Eliminar directorio VACÍO
directorio.rmdir() # Error si no está vacío
print("Directorio eliminado")
```

Importante: rmdir() solo elimina directorios VACÍOS. Si tiene archivos o subdirectorios, lanza OSError. Para eliminar directorios con contenido usa shutil.rmtree(), que veremos más adelante.

4.8.3 Renombrar y mover archivos

El método rename() cambia el nombre de un archivo o lo mueve a otra ubicación. El método replace() hace lo mismo pero sobrescribe si el destino existe.

```
from pathlib import Path
```

```
# Renombrar archivo en el mismo directorio
old = Path("antiguo.txt")
new = Path("nuevo.txt")
old.rename(new)
print("Archivo renombrado")
# Mover a otro directorio
archivo = Path("datos.txt")
destino = Path("backup/datos.txt")
archivo.rename(destino)
print("Archivo movido")
# replace() sobrescribe si el destino existe
archivo = Path("temp.txt")
destino = Path("final.txt")
archivo.replace(destino) # Si final.txt existe, lo sobrescribe
print("Archivo reemplazado")
```

Nota: La diferencia entre rename() y replace(): rename() da error si el destino existe, replace() lo sobrescribe sin preguntar.

4.9 Módulos os y shutil: Interacción con el Sistema

Mientras que pathlib se enfoca en manipular las rutas y el contenido de ficheros individuales, los módulos os (Operating System) y shutil (Shell Utilities) se utilizan para realizar acciones de la línea de comandos a nivel del sistema de archivos, como crear directorios recursivamente, copiar árboles de carpetas o renombrar archivos.

4.9.1 Módulo os – Operaciones básicas

El módulo "os" provee una interfaz para interactuar con el sistema operativo. Es esencial para tareas de bajo nivel como crear, eliminar, listar directorios, variables de entorno, permisos…
En Python moderno (2025), **pathlib es la forma recomendada** para trabajar con rutas y archivos. Sin embargo, el módulo os todavía tiene **métodos útiles y necesarios** que pathlib NO tiene.

¿Cuándo usar os y cuándo pathlib?
Usa pathlib para:

- Crear rutas.
- Manipular rutas (unir, obtener partes).
- Crear archivos y directorios.
- Leer/escribir archivos.
- Eliminar archivos y directorios vacíos.

Usa os para:

- Variables de entorno.
- Información del sistema.
- Cambiar directorio de trabajo.
- Ejecutar comandos del sistema.
- Permisos y propietarios (Unix/Linux).

- Información de procesos.

Importante: NO uses os.path, os.mkdir(), os.remove(), etc. Usa pathlib para eso.

Variables de entorno

Las variables de entorno son configuraciones del sistema operativo que los programas pueden leer.

os.getenv() - Obtener variable con valor por defecto

```
import os


# MEJOR: con valor por defecto
api_key = os.getenv('API_KEY', 'no_configurada')
print(f"API Key: {api_key}")


# PROBLEMA: lanza KeyError si no existe
try:
    api_key = os.environ['API_KEY']
except KeyError:
    print("API_KEY no está configurada")


# RECOMENDADO: usar getenv()
api_key = os.getenv('API_KEY')
if api_key is None:
    print("API_KEY no está configurada")
```

os.environ - Establecer variables

```
import os


# Modificar os.environ
os.environ['MI_VARIABLE'] = 'valor'
print(os.getenv('MI_VARIABLE'))  # "valor"


os.environ['PUERTO'] = '8080'
```

Importante: Los cambios en os.environ solo afectan al proceso actual y sus hijos, NO al sistema completo. Cuando se acaba de ejecutar el archivo .py las variables se eliminan.

Uso práctico – Carga de un fichero de configuración

Tengo un archivo .env con la configuración de mi base de datos.

```
DB_HOST=localhost
DB_PORT=5432
DB_NAME=miapp
```

```
DEBUG=True
SECRET_KEY=mi_secreto_super_seguro
```

Importante:
* Una variable por línea.
* Formato: NOMBRE=valor.
* Sin espacios alrededor del =.
* Sin comillas (aunque puedes usarlas si el valor tiene espacios).

Sería mucho más fácil poder tener estas variables en os.environ para poder acceder a ellas con os.getenv. Para ello hay que instalar Python-dotenv. Recuerda activar antes el entorno virtual:

source venv/bin/activate
python -m pip install python-dotenv

Puede que tengas que seleccionar el intérprete en nuestro caso el venv (porque el import de dotenv te aparece en rojo), haciendo click en la rueda de configuración de abajo a la izquierda > paleta de comandos. Ahí busca seleccionar intérprete y busca la venv/bin/python como intérprete.

Importante: Debes tener cargada en VSCode la carpeta donde esté el directorio venv. Si tienes la carpeta Ejercicios y dentro Tema 4 y solo cargas la carpeta Tema 4 en VSCode, no se quitará el rojo en la importación, debes cargar la carpeta Ejercicios.

Cargar el .env en Python

```python
from pathlib import Path
from dotenv import load_dotenv
import os

BASE_DIR = Path(__file__).parent
env_path = BASE_DIR / ".env"

# Cargar las variables del archivo .env
load_dotenv(env_path)

# Ahora acceder con os.getenv()
api_key = os.getenv('API_KEY')
nombre_bd = os.getenv('DB_NAME')
clave_bd = os.getenv('SECRET_KEY')

print(f"API Key: {api_key}")
print(f"Nombre Base de Datos: {nombre_bd}")
print(f"Clave Base de Datos: {clave_bd}")
```

Información del sistema operativo
os.name - Nombre del sistema operativo

```
import os

print(f"Sistema operativo: {os.name}")

# Valores posibles:
# - 'posix' -> Linux, Mac, Unix
# - 'nt'    -> Windows

# Detectar Windows
if os.name == 'nt':
    print("Estás en Windows")
else:
    print("Estás en Linux/Mac/Unix")
```

platform - Información detallada

```
import platform

print(f"Sistema: {platform.system()}")       # Windows, Linux, Darwin (Mac)
print(f"Versión: {platform.version()}")      # Versión del OS
print(f"Release: {platform.release()}")      # Release del OS
print(f"Arquitectura: {platform.machine()}")  # x86_64, arm64, etc.
print(f"Procesador: {platform.processor()}")  # Información del CPU
print(f"Python: {platform.python_version()}") # Versión de Python

# Información completa
print(f"Completo: {platform.platform()}")
# Windows-10-10.0.19041-SP0
# Linux-5.10.0-21-amd64-x86_64-with-glibc2.31
```

Ejecutar comandos del Sistema
subprocess - La forma MODERNA (RECOMENDADO, mejor que os.system())

```
import subprocess

# MODERNO: subprocess.run()
resultado = subprocess.run(
    ['ls', '-la'],
    capture_output=True,
    text=True
)
```

```
print("Salida:")
print(resultado.stdout)

print("Código de retorno:", resultado.returncode)
```

Permisos de archivos

Importante: Estos métodos solo funcionan en sistemas Unix (Linux, Mac). En Windows se ignoran o dan error.

os.chmod() - Cambiar permisos

```
import os
from pathlib import Path

# Crear archivo
archivo = Path("script.sh")
archivo.write_text("#!/bin/bash\necho 'Hola'")

# Cambiar permisos a ejecutable (rwxr-xr-x = 755)
os.chmod(archivo, 0o755)

# También funciona en pathlib (Python 3.10+)
archivo.chmod(0o755)

print(f"{archivo} ahora es ejecutable")
```

4.9.2 El módulo sutil (Utilidades de alto nivel)

El módulo shutil es para operaciones de alto nivel y manipulación de archivos que normalmente harías en una shell (línea de comandos). Es más robusto que las funciones básicas de os para tareas como copiar directorios completos. Permite copiar, mover, eliminar directorios completos, crear archivos ZIP…

Operaciones clave de archivos:

Función	Descripción	Ejemplo
shutil.copy(src, dst)	Copia el fichero de origen (src) al destino (dst).	shutil.copy('info.txt', 'backup/info.txt')
shutil.copy2(src, dst)	Copia + metadatos (preserva la fecha original)	shutil.copy2('info.txt', 'backup/info.txt')
shutil.copytree(src, dst)	Copia recursivamente todo el contenido de un directorio (src) a otro nuevo (dst).	shutil.copytree('recursos', 'copia_recursos')
shutil.move(src, dst)	Mueve (o renombra, si el destino está en el mismo directorio) un fichero o directorio.	shutil.move('temp/file.log', 'logs/file.log')
shutil.rmtree(path)	Elimina recursivamente un directorio y todo su	shutil.rmtree('dir_antiguo')

	contenido. ¡Usar con precaución!	

```python
import shutil
from pathlib import Path
# Copiar archivo simple
shutil.copy('datos.txt', 'backup/datos.txt')
print("Archivo copiado")
# Copiar preservando metadatos (fecha, permisos)
shutil.copy2('config.json', 'backup/config.json')
print("Archivo copiado con metadatos")
# Copiar directorio completo
shutil.copytree('proyecto', 'proyecto_backup')
print("Directorio copiado")
# Mover archivo
shutil.move('temp/log.txt', 'logs/log.txt')
print("Archivo movido")
# Eliminar directorio completo (¡PELIGROSO!)
if Path('directorio_antiguo').exists():
    shutil.rmtree('directorio_antiguo')
    print("Directorio eliminado completamente")
```

Importante: shutil.rmtree() ELIMINA TODO sin preguntar: archivos, subdirectorios, etc. NO hay papelera de reciclaje, NO hay confirmación. Usa con EXTREMO cuidado.

4.10 Manejo de errores con ficheros

Cuando se intenta acceder a un fichero que no existe, Python lanza la excepción FileNotFoundError. Es crucial manejarla con bloques **try...except** para que el programa no falle inesperadamente. También es recomendable manejar PermissionError y ValueError (al convertir datos de texto).

```python
from pathlib import Path
fichero_inexistente = Path('archivo_fantasma.txt')

try:
    # Intenta leer un fichero que puede no existir
    contenido = fichero_inexistente.read_text(encoding='utf-8')
    print(contenido)

except FileNotFoundError:
    # Captura específica para ficheros no encontrados.
    print(f"\n[ERROR] No se encontró el fichero: {fichero_inexistente.name}")
    print("El programa ha manejado la ausencia del archivo sin fallar.")
```

```
except PermissionError:
    # Captura si el sistema operativo deniega la operación.
    print("[ERROR] Acceso denegado. Verifique permisos.")

except ValueError:
    # Captura errores de codificación o conversión de datos
    print(f"\n Error de formato o codificación")
```

Nota: Capturar errores específicos te da MÁS INFORMACIÓN al usuario sobre qué salió mal y cómo puede solucionarlo. Es mucho mejor que un 'Error desconocido'.

4.11 Ficheros CSV – Datos tabulares

CSV (Comma-Separated Values) es el formato estándar para datos tabulares (filas y columnas). Lo usa Excel, Google Sheets, Bases de Datos, etc. Python tiene el módulo csv que maneja correctamente casos complicados: comas dentro de campos, saltos de línea, diferentes delimitadores.

Concepto clave - ¿Por qué usar el módulo csv?:
Podrías leer CSV con split(','), pero fallaría con: 'Madrid, España' (coma dentro del campo), campos con comillas, diferentes delimitadores (;, tab), saltos de línea (pueden ser distintos según el Sistema Operativo). El módulo csv maneja TODO esto correctamente.

4.11.1 Lectura de CSV con DictReader (como diccionario)

DictReader es la forma más profesional de leer CSV. Cada fila es un diccionario donde las claves son los nombres de las columnas. Es más legible que trabajar con listas de números.

```
import csv


fichero = Path('datos.csv')


with fichero.open('r', newline='', encoding='utf-8') as f:
    # La cabecera se usa como clave automáticamente
    lector = csv.DictReader(f)

    for fila in lector:
        # Acceso por nombre de columna (más legible y menos propenso a errores)
        print(f"Nombre: {fila['Nombre']}, Salario: {fila['Salario']}")


# Cargar todo en una lista (si necesitas procesar varias veces)
with fichero.open('r', newline='', encoding='utf-8') as f:
    datos = list(csv.DictReader(f))
    print(f"\nTotal empleados: {len(datos)}")
```

Importante: El parámetro newline='' controla cómo Python maneja los saltos de línea. Windows usa \n\n Unix/Linux/Mac usan \n. Siempre úsalo cuando trabajes con CSV.

4.11.2 Lectura de CSV con reader (Leer como Listas)

Si tu CSV NO tiene cabecera o prefieres trabajar con listas en lugar de diccionarios, usa csv.reader. Cada fila es una lista de strings.

```
import csv


fichero = Path('datos.csv')


with fichero.open('r', newline='', encoding='utf-8') as f:
    lector = csv.reader(f)


    # Imprime la cabecera (primera lista)
    cabecera = next(lector)
    print(f"Cabecera: {cabecera}")


    for fila in lector:
        print(f"Fila: {fila}")
```

Nota: reader devuelve TODO como strings. Si necesitas números, convierte manualmente: edad = int(fila[1]) o precio = float(fila[2]).

4.11.3 Escribir CSV: csv.writer

Para escribir CSV, usa csv.writer (para listas) o csv.DictWriter (para diccionarios). DictWriter es más claro porque especificas explícitamente los nombres de las columnas.

```
import csv
# MÉTODO 1: Escribir desde listas
datos = [
    ['Nombre', 'Edad', 'Ciudad'], # Cabecera
    ['Ana', 30, 'Madrid'],
    ['Luis', 25, 'Barcelona']
]
with open('empleados.csv', 'w', newline='', encoding='utf-8') as f:
    escritor = csv.writer(f)
    escritor.writerows(datos) # Escribe todas las filas
print("CSV creado con writer")


# MÉTODO 2: Escribir desde diccionarios (MÁS CLARO)
empleados = [
    {'nombre': 'Ana', 'edad': 30, 'ciudad': 'Madrid'},
```

```
    {'nombre': 'Luis', 'edad': 25, 'ciudad': 'Barcelona'}
]
with open('emp.csv', 'w', newline='', encoding='utf-8') as f:
    # Especificar nombres de columnas
    campos = ['nombre', 'edad', 'ciudad']
    escritor = csv.DictWriter(f, fieldnames=campos)
    escritor.writeheader() # Escribir cabecera
    escritor.writerows(empleados) # Escribir datos
print("CSV creado con DictWriter")
```

Nota: DictWriter te protege de errores: si intentas escribir un campo que no está en fieldnames, lanza un error. Con writer normal podrías escribir filas de diferentes longitudes accidentalmente.

Ejercicio 2

Utiliza pathlib para crear un fichero llamado reporte_ventas.csv (el formato CSV es texto plano). El fichero debe contener tres líneas de datos de ventas, usando la coma(,) como separador.

1. Abre y escribe el fichero en modo de escritura (write_text).
2. Escribe la cabecera: Region,Ventas_2024,Ventas_2025
3. Escribe dos líneas de datos:
    - Norte,10500.50,12300.00
    - Sur,8900.25,9500.75

Instala la extensión Rainbow CSV para que los .csv se vean más bonitos.

Ejercicio 3

Escribe un programa que:

1. Lea el fichero reporte_ventas.csv usando .read_text().
2. Calcule la suma total de Ventas_2025 de las dos regiones.
3. Añade (usando .open('a')) una línea con el total calculado al final del fichero, en el formato: Total,0,SUMA_TOTAL.

Ten en cuenta que el .csv puede no existir o que la conversión a float falle (ValueError). No debes coger la primera línea, que es la cabecera (usa slicing de listas para coger del primer valor de la lista hasta el final)

Ejercicio 4

Crea un fichero llamado log_acceso.txt con el siguiente contenido(desde Python, no a mano – write-text):

[2024-11-20 10:05:01] ERROR: Fichero no encontrado en /data/a/fichero.txt
[2024-11-20 10:05:30] INFO: Usuario 'user_001' ha iniciado sesion.
[2024-11-20 10:06:15] WARN: Disco al 80%.
[2024-11-20 10:06:40] INFO: Usuario 'user_002' ha iniciado sesion.
[2024-11-20 10:07:05] ERROR: Conexión perdida con base de datos.

Escribe un programa que:

1. Lea el fichero log_acceso.txt de forma eficiente (línea por línea).
2. Cree una nueva lista que contenga solo las líneas de ERROR o WARN.

3. Escriba estas líneas filtradas en un nuevo fichero llamado log_errores.txt.

Ejercicio 5
Tienes un fichero CSV con datos de empleados. Crea un fichero llamado empleados.csv con el siguiente contenido:
ID,Nombre,Departamento,Salario
101,Ana García,Ventas,45000
102,Luis Pérez,IT,60000
103,Elena Sánchez,Ventas,48000
Escribe un programa que:

1. Lea el fichero empleados.csv utilizando el módulo csv.DictReader.
2. Calcule el salario total de los empleados del departamento de "Ventas".
3. Imprima el resultado.

Ejercicio 6 - Uso de pathlib y os.makedirs
Quieres crear una estructura de carpetas anidadas para guardar informes de ventas de diferentes años, asegurándote de que el programa no falle si ya existen.

1. Define la ruta base a crear: data/informes/2026/ventas.
2. Utiliza os.makedirs() con el argumento exist_ok=True para crear toda la estructura.
3. Una vez creada la estructura, utiliza el operador / de pathlib para crear la ruta completa de un fichero dentro del directorio final: informe_final.txt.
4. Escribe el texto "Informe Creado" en dicho fichero.

Ejercicio 7 – Sistema de logs con fechas
Un sistema profesional de logs necesita:
1. Crear directorio de logs.
2. Un archivo por día.
3. Añadir timestamps a cada entrada. (nombre: app_2025-12-09.log)
4. Modo append para no perder logs anteriores.
5. Puedes limpiar logs antiguos buscando archivos con glob() y eliminando los que tengan más de X días(usa el nombre del fichero sin extensión(stem) y reemplaza app_ por “”).

5. Almacenamiento de Datos (JSON)

5.1 Introducción a JSON

El formato JSON (JavaScript Object Notation) es un formato de texto ligero y legible por humanos, diseñado para el intercambio de datos. Se ha convertido en el estándar de facto para la comunicación entre servidores y aplicaciones web. En Python, el módulo integrado json permite serializar (convertir objetos Python a JSON) y deserializar (convertir JSON a objetos Python) con facilidad.

5.1.1 ¿Por qué usar JSON?

• Legible: Fácil de leer y escribir para humanos.
• Ligero: Menos verboso que XML, ideal para transferencia de datos.
• Universal: Soportado por todos los lenguajes de programación modernos.
• Texto plano: No requiere librerías especiales para visualización.
• Estándar web: Formato nativo de JavaScript, perfecto para APIs REST.

5.1.2 Casos de uso comunes

• APIs REST: Comunicación entre frontend y backend (ej: acceso a API del tiempo).
• Archivos de configuración: Almacenar settings de aplicaciones.
• Persistencia ligera: Guardar datos sin necesidad de base de datos.
• Logging estructurado: Registros de eventos en formato estructurado.
• Intercambio de datos: Entre servicios, microservicios o sistemas diferentes.
• NoSQL databases: MongoDB usa JSON para almacenar documentos.

5.1.3 Ejemplo de JSON

Este es un ejemplo típico de respuesta JSON de una API:

```
{
"usuario": "miguel.rodriguez",
"edad": 35,
"activo": true,
"roles": ["profesor", "desarrollador"],
"direccion": { "ciudad": "Madrid",
        "codigo_postal": "28001" },
"cursos_impartidos": 12,
"ultima_conexion": null
}
```

Observa que JSON usa:
• Comillas dobles " " obligatorias para strings (nunca comillas simples).
• Llaves { } para objetos (diccionarios).
• Corchetes [ ] para arrays (listas).
• Coma , entre elementos (sin coma final).
• Dos puntos : entre clave y valor.
• Valores: string, number, boolean (true/false), null, objeto, array.

Importante: JSON es más estricto que Python: SIEMPRE usa comillas dobles, no permite comas finales, y usa true/false/null en minúsculas.

5.2 Mapeo: Estructuras Fundamentales de JSON y Python

Para que Python pueda guardar o leer datos JSON, necesita saber cómo traducir los símbolos de JSON a sus propios tipos de datos. El módulo json hace esta traducción automática, que se conoce como mapeo.

5.2.1 Mapeo de estructuras principales

En JSON los objetos se definen entre {} y como en Python los diccionarios también se definen con {}, es fácil convertir objetos de JSON a Diccionarios en Python en formato clave-valor.
En JSON los arrays se definen entre [] y esto hace que la conversión a listas de Python sea instantánea.

5.2.2 Tabla de mapeo completa

Tipo JSON	Tipo Python	Ejemplo JSON	Ejemplo Pyton
object	dict	{"a": 1}	{'a': 1}
array	list	[1, 2, 3]	[1, 2, 3]
string	str	"hola"	'hola'

number (int)	int	42	42
number (float)	float	3.14	3.14
true	True	true	True
false	False	false	False
null	None	null	None

En esencia, el módulo json actúa como un traductor: toma los símbolos de la estructura de datos que recibe y los mapea a la estructura nativa de Python que utiliza los mismos símbolos o representa la misma lógica (clave-valor u orden).

Importante: Python acepta comillas simples y dobles, pero JSON SOLO acepta comillas dobles. El módulo json se encarga de esta conversión automáticamente.

5.3 Serialización: de Python a JSON

La serialización, también conocida como codificación, es el proceso de tomar una estructura de datos de Python (como un diccionario o una lista) y convertirla en una cadena de texto en formato JSON.

5.3.1 El Método json.dumps() - Dump String

Este método se utiliza cuando queremos convertir el objeto Python a una cadena JSON (str) para manipularlo o enviarlo, sin escribirlo directamente en un fichero.

```
import json


datos_python = {
    'nombre': 'Análisis Ventas Q3',
    'productos': ['Software', 'Hardware', 'Servicios'],
    'total_q3': 45000.75,
    'activo': True
}


# Usamos dumps() para obtener la representación JSON como una CADENA de texto.
json_string = json.dumps(datos_python)


print(f"Tipo original: {type(datos_python)}")
print(f"Tipo JSON (string): {type(json_string)}")
print(json_string)
```

Observa: Python usa True (mayúscula) pero JSON usa true (minúscula). El módulo json hace la conversión automáticamente.

5.3.2 El método json.dump() – Guardar en archivo

El método json.dump() es el método clave para la persistencia en ficheros. Escribe la estructura de datos directamente a un objeto archivo (file object) abierto.

Importante: No confundir json.dump() con json.dumps(). El primero escribe en archivo, el segundo devuelve string. La 's' final significa 'string'.

```
from pathlib import Path
import json
```

```python
fichero_json = Path('reporte.json')
datos_a_guardar = {
    'curso': 'Especialización Python',
    'version': 3.12,
    'modulos': 15
}

# 1. USAMOS json.dump() para escribir directamente en el fichero abierto.
# Esto es más eficiente que serializar primero a string (dumps) y luego escribir.
try:
    with fichero_json.open('w', encoding='utf-8') as f: # Abrimos el fichero
        # 'dump' necesita el objeto Python y el objeto fichero abierto 'f'
        json.dump(datos_a_guardar, f, indent=4)

    print(f"\nDatos serializados y guardados con indentación en '{fichero_json.name}'.")

except IOError:
    print("Error al escribir en el disco.")
```

Primero elige el nombre del fichero donde voy a guardar la estructura de datos, luego se crea dicha estructura. Es aconsejable que el fichero tenga la extensión .json.
Luego usamos json.dump para generar una cadena que contiene la representación JSON de la estructura de datos. Una vez que tenemos dicha estructura, la guardamos normalmente en el fichero.
Si queremos que la estructura del json en el fichero sea "bonita"(legible), usar "indent": contenido = json.dump(datos_a_guardar, fichero, indent=4)

```
# Sin indent (una sola línea, difícil de leer) json.dump(datos, f)
{"curso": "Python", "version": 3.12, "modulos": 15}
# Con indent=4 (formato legible) json.dump(datos, f, indent=4)
{
"curso": "Python",
"version": 3.12,
"modulos": 15
 }
```

Recomendación profesional: Usa indent=4 en desarrollo y archivos de configuración. En producción (APIs), el JSON compacto ahorra ancho de banda.

5.4 Deserialización: De JSON a Python (load y loads)

La deserialización o decodificación, es el proceso inverso: tomar texto en formato JSON (ya sea de una cadena o de un fichero) y reconstruir el objeto Python original (diccionario o lista).

5.4.1 El Método json.loads() - Load String

Este método se usa cuando recibimos el JSON como una **cadena de texto** (típico en respuestas de APIs) y queremos convertirlo en un objeto Python manipulable.

```
import json

json_recibido = '{"cliente": "A. Smith", "pedido": 1001, "pagado": false}'

# Usamos loads() para convertir la cadena JSON en un diccionario de Python.
objeto_python = json.loads(json_recibido)

print(f"\nTipo JSON: {type(json_recibido)}")
print(f"Tipo Python (dict): {type(objeto_python)}")
print(f"Estado del pago: {objeto_python['pagado']}")
```

5.4.2 El Método json.load() – Cargar desde archivo

Este es el método fundamental para cargar datos persistentes. Lee y decodifica directamente el JSON desde un objeto archivo abierto.

Importante: No confundir json.load() con json.loads(). El primero lee desde archivo, el segundo lee desde string. La 's' final significa 'string'.

Por defecto, "open" abre el fichero en modo lectura, por lo que no se puede modificar.

```
from pathlib import Path
import json

fichero_json = Path('reporte.json')
# Asegúrate de que el fichero existe antes de intentar abrirlo,
# o usa try/except más amplio.
try:
    with fichero_json.open('r', encoding='utf-8') as f: # Usar with open()
        datos_cargados = json.load(f)

        print("\n--- Datos Cargados ---")
        print(f'Curso: {datos_cargados['curso']}')

except FileNotFoundError:
    print(f'Error: No se encontró el fichero {fichero_json.name}.')

except json.JSONDecodeError:
    # Esta excepción es clave: el fichero existe, pero el contenido es JSON inválido.
    print(f'Error: El fichero {fichero_json.name} contiene JSON inválido.')

except Exception as e:
    print(f'Ocurrió un error inesperado: {e}')
```

5.4.3 *Manejo profesional de errores*

Al deserializar (cargar JSON), el error más común no es que el fichero no exista (FileNotFoundError), sino que el contenido del fichero sea JSON inválido (ej., una coma fuera de lugar, o comillas simples en lugar de dobles). Es fundamental manejar esta excepción profesionalmente.

Errores comunes en JSON:

- Comillas simples: {'nombre': 'Juan'} → Debe ser {"nombre": "Juan"}
- Coma final: {"a": 1,} → JSON no permite comas finales
- Sin comillas en claves: {nombre: "Juan"} → Las claves deben tener comillas
- Comentarios: JSON no soporta comentarios // esto o /* esto */
- True/False en mayúsculas: Debe ser true/false (minúsculas)

Ejercicio 1

Define una lista de diccionarios, donde cada diccionario representa una tarea con las claves título, estado (True/False) y prioridad (número). Serializa esta lista y guárdala en un fichero llamado tareas.json. Asegúrate de usar indent=2 para que el JSON sea legible.

Estructura de la lista:

```
tareas = [
    {'titulo': 'Revisar tema JSON', 'estado': True, 'prioridad': 1},
    # ... otras tareas
]
```

Ejercicio 2

Escribe un programa que:

- Cargue el fichero tareas.json del ejercicio anterior.
- Cuente cuántas tareas tienen el estado igual a False (tareas pendientes).
- Muestre los títulos de las tareas pendientes.
- Añada una nueva tarea a la lista ('titulo': 'Planificar examen', 'estado': False, 'prioridad': 3).
- Sobrescriba el fichero tareas.json con la lista actualizada.

5.5 Casos de uso prácticos

5.5.1 *Archivo de configuración de aplicación*

Es muy común usar JSON para almacenar la configuración de una aplicación.

Archivo config.json:

```
{
    "database": {
        "host": "localhost",
        "port": 5432,
        "nombre": "mi_bd",
        "usuario": "admin"
    },
    "api_keys": {
        "weather": "abc123def456",
        "maps": "xyz789uvw012"
```

```
    },
    "debug": true,
    "max_conexiones": 100
}
```

Archivo que carga la configuración guardada:

```
import json
from pathlib import Path
def cargar_configuracion():
    """
    Cargar configuración en la aplicación
    """
    config_file = Path('config.json')
    try:
        with config_file.open('r', encoding='utf-8') as f:
            config = json.load(f)
            return config
    except FileNotFoundError:
        print("Error: Archivo de configuración no encontrado")
        return None
    except json.JSONDecodeError:
        print("Error: Configuración JSON inválida")
        return None # Usar configuración
config = cargar_configuracion()
if config:
    print(f"Conectando a BD: {config['database']['host']}")
    if config['debug']:
        print("Modo DEBUG activado")
```

5.5.2 Guardar respuesta de API

También es común guardar la respuesta de una API externa, permitiendo:

- Reducir las llamadas a la API (ahorrando cuota y dinero).
- Respuesta instantánea para datos recientes.
- Funciona offline si hay caché válida.
- Reduce latencia de ~200ms a ~2ms.

5.5.3 Loggin estructurado

JSON es excelente para logs que luego necesitas procesar automáticamente:

Archivo para crear eventos de log:

```
import json
from datetime import datetime
from pathlib import Path
```

```python
def log_evento(nivel, mensaje, **kwargs):
    """Registra evento en formato JSON"""
    log_file = Path('app.log')
    evento = { 'timestamp': datetime.now().isoformat(), 'nivel': nivel, 'mensaje': mensaje, **kwargs}
    # Append mode: añadir al final sin sobrescribir
    with log_file.open('a', encoding='utf-8') as f:
        json.dump(evento, f, ensure_ascii=False)
        f.write('\n') # Un evento por línea

# Ejemplos de uso
log_evento('INFO', 'Usuario iniciado sesión', usuario='miguel', ip='192.168.1.100')
log_evento('ERROR', 'Fallo en API externa', api='weather', codigo=500)
log_evento('WARNING', 'Límite de API cercano', llamadas_restantes=50)
```

Archivo de log existente, app.log:

```
{"timestamp": "2025-01-15T10:30:45","nivel": "INFO","mensaje": "Usuario iniciado sesión","usuario":
"miguel","ip": "192.168.1.100"}
{"timestamp": "2025-01-15T10:31:02","nivel": "ERROR","mensaje": "Fallo en API externa","api":
"weather","codigo": 500}
{"timestamp": "2025-01-15T10:31:15","nivel": "WARNING","mensaje": "Límite de API
cercano","llamadas_restantes": 50}
```

Archivo para analizar los logs:

```python
import json
from pathlib import Path

def analizar_logs():
    log_file = Path('app.log')
    errores = []
    with log_file.open('r', encoding='utf-8') as f:
        for linea in f:
            evento = json.loads(linea)
            if evento['nivel'] == 'ERROR':
                errores.append(evento)
    print(f"Total errores: {len(errores)}")
    for error in errores:
        print(f" - {error['timestamp']}: {error['mensaje']}")
analizar_logs()
```

5.6 Buenas prácticas y recomendaciones

5.6.1 Encoding UTF-8

SIEMPRE especifica encoding='utf-8' al abrir archivos JSON:

```
# CORRECTO - con encoding
with fichero.open('w', encoding='utf-8') as f:
    json.dump(datos, f)
# INCORRECTO - sin encoding (puede fallar con acentos)
with fichero.open('w') as f:
    json.dump(datos, f)
```

Importante: Sin UTF-8, caracteres como 'ñ', 'á', 'ü' pueden causar errores UnicodeEncodeError. JSON es texto UTF-8 por estándar.

5.6.2 Parámetro ensure_ascii

Por defecto, json.dumps() escapa caracteres no-ASCII. Para mantener acentos:

```
datos = {'nombre': 'José', 'ciudad': 'Málaga'}
# Por defecto (ensure_ascii=True)
print(json.dumps(datos))
# {"nombre": "Jos\u00e9", "ciudad": "M\u00e1laga"}
# Con ensure_ascii=False
print(json.dumps(datos, ensure_ascii=False))
# {"nombre": "José", "ciudad": "Málaga"}
```

Recomendación: Usa ensure_ascii=False con encoding='utf-8' para mantener caracteres legibles.

5.6.3 Validación JSON

Siempre valida que los datos esperados existan antes de usarlos:

```
def procesar_usuario(json_data):
    try:
        datos = json.loads(json_data)
        # Validar que existan las claves necesarias
        if 'nombre' not in datos or 'email' not in datos:
            print("Error: JSON incompleto, faltan campos obligatorios")
            return None
        # Validar tipos
        if not isinstance(datos['nombre'], str):
            print("Error: 'nombre' debe ser string")
            return None
        # Validar valores
        if '@' not in datos['email']:
            print("Error: email inválido")
            return None
        return datos
    except json.JSONDecodeError:
```

```
    print("Error: JSON mal formado")
    return None
```

5.6.4 No guardar datos sensibles en JSON plano

Importante: NUNCA guardes contraseñas, tokens de API, o datos sensibles en archivos JSON sin encriptar. Usa variables de entorno o servicios de secrets management.

```
# MAL - API key en JSON
{ "api_key": "sk-1234567890abcdef" }
# BIEN - Usar variable de entorno
import os
api_key = os.getenv('WEATHER_API_KEY')
# BIEN - Archivo .env (no subir a git)
WEATHER_API_KEY=sk-1234567890abcdef
```

5.7 Tipos de datos avanzados

5.7.1 Problema: JSON no soporta datetime

JSON solo soporta tipos básicos. Para fechas, hay que convertirlas a string:

```
from datetime import datetime
import json
datos = { 'evento': 'Conferencia Python', 'fecha': datetime.now()}
# Esto causará TypeError
json.dumps(datos)# TypeError: Object of type datetime is not JSON serializable
```

Solución: convertir manualmente

```
from datetime import datetime
import json
datos = { 'evento': 'Conferencia Python', 'fecha': datetime.now().isoformat() }
# Convertir a string ISO 8601
json_string = json.dumps(datos)
print(json_string) # {"evento": "Conferencia Python", "fecha": "2025-01-15T10:30:45.123456"}
# Al cargar, reconvertir a datetime
datos_cargados = json.loads(json_string)
fecha_objeto = datetime.fromisoformat(datos_cargados['fecha'])
print(type(fecha_objeto))
#<class 'datetime.datetime'>
```

Ejercicios

Ejercicio 3 – Configuración de aplicación

Crea un archivo config.json con la siguiente estructura:

```
{
```

```
{
  "app_name": "Mi Aplicación Web",
  "version": "1.0.0",
  "database": {
    "host": "localhost",
    "port": 5432,
    "nombre": "mi_bd"
  },
  "debug": true
}
```

Escribe un programa que:

1. Cargue la configuración desde config.json (crea función cargar_configuración() que devuelva la configuración.
2. Muestre el nombre de la aplicación y versión (recibe la configuración como parámetro de entrada). También la configuración de la BD.
3. Si debug es true, muestre "Modo DEBUG activado".
4. Modifique debug a false (crea una función que reciba la configuración y el nuevo valor como parámetros de entrada, retorna la nueva configuración).
5. Guarde la configuración actualizada (crea función guardar_configuración que recibe la configuración como parámetro de entrada). Retorna True si todo va bien y False en caso contrario.

Ejercicio 4 - Caché de API

Implementa un sistema de caché simple para una API ficticia:

1. Crea una función guardar_en_cache(ciudad, datos) que guarde datos en cache_{ciudad}.json. Retorna True si todo va bien y False en caso contrario.
2. Crea una función cargar_desde_cache(ciudad) que cargue datos si existen. Si el archivo no existe, devolver None. Debe retornar los datos de la caché.
3. Crea la función mostrar_datos_ciudad, que reciba los datos de caché como parámetro de entrada y muestre los datos meteorológicos de la ciudad
4. Prueba guardando datos de Madrid y Barcelona
5. Verifica que puedes cargar ambos caches

```
# Pista: Estructura de datos
datos_madrid = { 'ciudad': 'Madrid', 'temperatura': 18.5, 'descripcion': 'Soleado' }
guardar_en_cache('madrid', datos_madrid)
cache = cargar_desde_cache('madrid')
print(cache['temperatura']) # 18.5
print("\n3. Mostrando datos de Madrid:")
mostrar_datos_ciudad(cache_madrid)
```

Ejercicio 5 - Validación de JSON

Escribe una función validar_usuario(json_string) que:

1. Intente deserializar el JSON
2. Verificar que sea un diccionario
3. Verifique que existan las claves: nombre, email, edad
4. Verifique que edad sea un número entre 18 y 100
5. Verifique que email contenga el carácter '@'

6. Devuelva True si es válido, False si no
7. Maneje json.JSONDecodeError apropiadamente

```
# Casos de prueba
print(validar_usuario('{"nombre": "Ana", "email": "ana@mail.com", "edad": 25}')) # True
print(validar_usuario('{"nombre": "Luis", "edad": 30}')) # False (falta email)
print(validar_usuario('{"nombre": "Eva", "email": "eva.com", "edad": 25}')) # False (email sin @)
print(validar_usuario('{nombre: "Juan"}')) # False (JSON inválido)
```