

# EL LIBRO DE PYTHON



+50  
EJEMPLOS  
Y EJERCICIOS

+100  
ERRORES  
COMUNES

ÁLVARO REVUELTA

# El Libro De Python

Álvaro Revuelta

16 de diciembre de 2024

Versión: v2024-1

<https://www.ellibrodepython.com>

---

Gracias por comprar El Libro de Python. Este libro es una guía completa para aprender Python. Su metodología gira en torno a dos pilares:

- La regla de Pareto. Aplicada en este contexto, dice que con el 20% del lenguaje puedes resolver el 80% de los problemas. Eliminamos todo lo no necesario.
- Aprender haciendo. De nada sirve consumir contenido técnico sin ponerlo en práctica. Mostramos ejemplos prácticos y pequeños proyectos y ejercicios que te ayudarán a entender los conceptos.

El libro está destinado a cualquier persona que tenga unas nociones básicas de programación. No son necesarios conocimientos previos de Python.

 Ilustraciones: @ionutsebastian

 Portada: @lunadesigncm

 Agradecimientos: Elena, Elpidio, Lola e Irene. Por sus sugerencias y correcciones.

### **⚠️ Nota de Copyright ⚠️**

Este libro es para tu uso personal. Si no lo has comprado, te agradeceríamos que lo hicieras en [ellibrodepython.com](http://ellibrodepython.com). Parte de sus beneficios son destinados al blog. También ofrecemos licencias para empresas, centros educativos y universidades.

 [ellibrodepython@gmail.com](mailto:ellibrodepython@gmail.com)

© El Libro de Python. 2024.

---

# Índice

<b>Introducción</b>	<b>1</b>
<b>1 Presentación El Libro De Python</b>	<b>10</b>
1.1 ¿Qué es El Libro de Python? . . . . .	11
1.2 ¿Qué no es El Libro de Python? . . . . .	12
1.3 ¿Cómo leer El Libro de Python? . . . . .	13
1.4 Más allá de Python . . . . .	14
1.5 Pros/Contras de Python . . . . .	15
1.6 Paquetes en Python . . . . .	16
1.7 El Código Pythonic . . . . .	17
1.8 Python, ChatGPT y LLMs . . . . .	19
1.9 Primeros pasos con Python . . . . .	22
<b>2 Elige Tipo y Estructura de Datos</b>	<b>25</b>
2.1 Tipos de Datos . . . . .	26
2.2 Conversión Entre Tipos . . . . .	35
2.3 Estructuras de Datos . . . . .	37
2.4 Estructuras en NumPy . . . . .	48
2.5 Estructuras en Pandas . . . . .	52
2.6 Eligiendo Estructuras y Tipos . . . . .	55
<b>3 Control con Bucles y Condicionales</b>	<b>62</b>
3.1 Condicionales con if/else . . . . .	63
3.2 Switch usando Match . . . . .	67
3.3 Iterando Con Bucles For . . . . .	68
3.4 Iterando Con List Comprehension . . . . .	73
3.5 Iterando Con Bucles While . . . . .	75
<b>4 Reutiliza Código con Funciones</b>	<b>78</b>
4.1 Introducción a Funciones . . . . .	79

---

4.2	Paso por valor y referencia . . . . .	88
4.3	Documentando Funciones . . . . .	90
4.4	Anotaciones En Funciones . . . . .	93
4.5	Recursividad Con Funciones . . . . .	96
4.6	Decorando Funciones . . . . .	98
4.7	Yield y Generadores . . . . .	102
4.8	Lambdas y Programación Funcional . . . . .	105
4.9	Funciones asíncronas . . . . .	110
<b>5</b>	<b>Programación Orientada a Objetos</b>	<b>114</b>
5.1	Introducción y contexto . . . . .	115
5.2	Crea tus clases y objetos . . . . .	117
5.3	Crea tu constructor . . . . .	119
5.4	Crea tus atributos . . . . .	121
5.5	Crea tus métodos . . . . .	123
5.6	Decorador setter . . . . .	125
5.7	Decorador property . . . . .	128
5.8	Herencia de clases . . . . .	130
5.9	Polimorfismo y Duck Typing . . . . .	133
5.10	Métodos dunder . . . . .	134
5.11	Uso de dataclass . . . . .	140
5.12	Monkey patching . . . . .	141
<b>6</b>	<b>Gestiona Errores con Excepciones</b>	<b>143</b>
6.1	Gestión de errores . . . . .	144
6.2	Excepciones en Python . . . . .	145
6.3	Lanza Excepciones . . . . .	149
6.4	Uso de Context Managers . . . . .	152
6.5	Dos filosofías para manejar errores . . . . .	156
<b>7</b>	<b>Testea tu Código</b>	<b>159</b>
7.1	Introducción al Testing . . . . .	160

---

---

7.2	Testing con pytest . . . . .	164
7.3	Múltiples tests con parametrize . . . . .	171
7.4	Usando fixture en pytest . . . . .	173
7.5	Otras funcionalidades de pytest . . . . .	176
7.6	Coverage con pytest-cov . . . . .	179
7.7	Fuzzing con hypothesis . . . . .	182
<b>8</b>	<b>Top 50 Ejemplos Prácticos</b>	<b>186</b>
8.1	Firma digital con cryptography . . . . .	187
8.2	Encriptar mensaje con cryptography . . . . .	189
8.3	Determina si un número es primo . . . . .	191
8.4	Factoriza un número en primos . . . . .	193
8.5	Computación cuántica con qiskit . . . . .	194
8.6	Fuerza bruta con itertools . . . . .	196
8.7	Acortador de enlaces con flask . . . . .	198
8.8	Construye una API con flask . . . . .	200
8.9	Trabaja con ficheros con os . . . . .	201
8.10	Bases de datos con sqlite . . . . .	203
8.11	Patrones con re . . . . .	205
8.12	Analítica de futbol con seaborn . . . . .	207
8.13	Programar tareas con schedule . . . . .	210
8.14	Gráficas con matplotlib . . . . .	211
8.15	Redes neuronales con tensorflow . . . . .	213
8.16	Logging y verbosity con logging . . . . .	219
8.17	Cálculo simbólico con sympy . . . . .	221
8.18	Creando un juego con pygame . . . . .	222
8.19	Scraping web con beautifulsoup . . . . .	223
8.20	Unir pdfs con pypdf . . . . .	226
8.21	Crear excels con pyexcel . . . . .	227
8.22	Benchmark con timeit . . . . .	229
8.23	Rotar imagen con scikit-image . . . . .	231

---

8.24	Simulando parada de bus con simpy	232
8.25	Coordenadas y distancia con geopy	234
8.26	Genética y ADN con biopython	235
8.27	Visualizar moléculas con rdkit	236
8.28	Polinomios con numpy	238
8.29	Simulando apuestas con numpy	241
8.30	Simulación Monte Carlo con numpy	243
8.31	Análisis financiero con yfinance	246
8.32	Programación asincrona con aiohttp	249
8.33	Crear baraja de Poker con itertools	252
8.34	Barajar cartas con shuffle	253
8.35	Ordenar con bubble sort	255
8.36	Convertir binario a decimal	257
8.37	Usar código C con cffi	258
8.38	Martingala y apuestas con random	260
8.39	Temporizador con time	262
8.40	Calcular impuestos	262
8.41	Plot interactivo con bokeh	264
8.42	Simula hipotecas con matplotlib	266
8.43	Palabras El Quijote con wordcloud	269
8.44	Programas ejecutables con pyinstaller	270
8.45	Calcular interés compuesto	272
8.46	Busca el número que falta	273
8.47	Comprimir información	274
8.48	Interfaz de usuario con pyqt	275
8.49	Dashboard con streamlit	277
8.50	Problema de la mochila	279
<b>9</b>	<b>Top 100 Errores Comunes</b>	<b>282</b>
9.1	Listas: Acceder último elemento	283
9.2	Listas: Append devuelve None	283

---

---

9.3	Listas: Copia referencia . . . . .	283
9.4	Listas: Confundir append y extend . . . . .	284
9.5	Tuplas: De un valor . . . . .	284
9.6	Diccionarios: Sustituto if . . . . .	285
9.7	Diccionarios: Unir diccionarios . . . . .	285
9.8	If: Orden de operadores . . . . .	286
9.9	If: Paréntesis innecesario . . . . .	286
9.10	If: Condición con booleano . . . . .	287
9.11	If: Condición con lista . . . . .	287
9.12	If: Usar el operador ternario . . . . .	288
9.13	If: Ordenar checks . . . . .	288
9.14	If: Operador walrus . . . . .	289
9.15	Match: Usa match . . . . .	289
9.16	Operadores: Confundir orden . . . . .	290
9.17	Operadores: Confundir and . . . . .	290
9.18	Operadores: Usa in . . . . .	291
9.19	Bucles: Iterar elementos . . . . .	291
9.20	Bucles: Iterar índices y elementos . . . . .	292
9.21	Bucles: Iterar al revés . . . . .	293
9.22	Bucles: Iterar dos listas . . . . .	293
9.23	Bucles: Iterar y modificar Listas . . . . .	294
9.24	Bucles: Iterar diccionarios . . . . .	294
9.25	Bucles: Uso else en for . . . . .	295
9.26	Comprehension: Generar listas secuenciales . . . . .	295
9.27	Comprehensions: Modificar con list comprehensions . . . . .	296
9.28	Comprehensions: Filtrar con list comprehensions . . . . .	296
9.29	Funciones: Usar lambda . . . . .	297
9.30	Funciones: Uso de early return . . . . .	297
9.31	Funciones: Función sin implementar . . . . .	298
9.32	Funciones: Evitar uso de global . . . . .	299
9.33	Funciones: Retorno de None . . . . .	299

---

9.34 Funciones: No usar la función main . . . . .	300
9.35 Funciones: Anotaciones . . . . .	300
9.36 Generadores: Uso de sum . . . . .	301
9.37 Generadores: Confundir generadores con listas . . . . .	301
9.38 Generadores: Usar generador terminado . . . . .	302
9.39 Clases: Usar métodos estáticos . . . . .	302
9.40 Clases: Comparar objetos . . . . .	303
9.41 Clases: No definir str . . . . .	304
9.42 Excepciones: Manejar excepciones . . . . .	304
9.43 Excepciones: Manejar excepciones genéricas . . . . .	305
9.44 Excepciones: Ignorar excepciones . . . . .	306
9.45 Excepciones: Usar context managers . . . . .	306
9.46 Paquetes: Nombrar fichero como paquete . . . . .	307
9.47 Paquetes: Incluye tu licencia . . . . .	307
9.48 Paquetes: Usar fichero requirements.txt . . . . .	307
9.49 Comentarios: Redundantes . . . . .	308
9.50 Comentarios: Confundir con docstrings . . . . .	308
9.51 Comentarios: Documentar funciones . . . . .	308
9.52 General: Evita cálculos innecesarios . . . . .	309
9.53 General: Medir tiempo una vez . . . . .	310
9.54 General: Escapar comillas . . . . .	310
9.55 General: Uso incorrecto de sorted . . . . .	311
9.56 General: Confundir identidad con igualdad . . . . .	311
9.57 General: Nombrar variables mal . . . . .	312
9.58 General: Uso de asserts en producción . . . . .	312
9.59 General: Código específico por OS . . . . .	313
9.60 General: Uso de magic numbers . . . . .	313
9.61 General: Rellena tu gitignore . . . . .	314
9.62 General: Precisión en float . . . . .	314
9.63 General: Usa alias con import . . . . .	315
9.64 General: Usar f-strings . . . . .	315

---

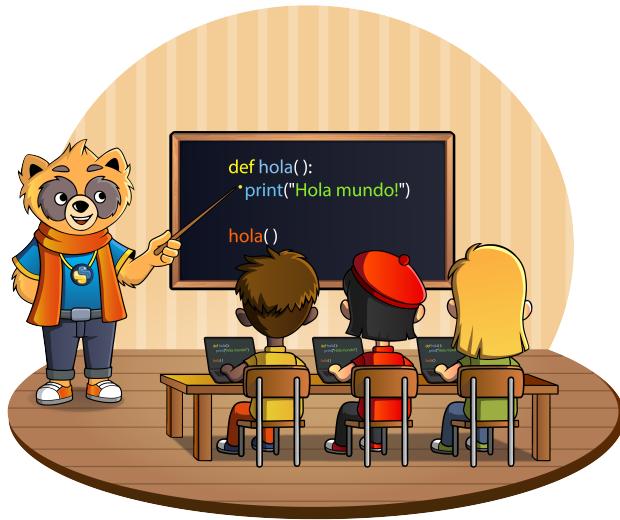
9.65 General: Uso de print en producción . . . . .	316
9.66 General: Intercambiar variables . . . . .	316
9.67 General: Extraer tuplas a variables . . . . .	317
9.68 General: Uso de all . . . . .	317
9.69 General: No repeter código 1 . . . . .	317
9.70 General: No repetir código 2 . . . . .	318
9.71 General: Uso de tuplas con constantes . . . . .	318
9.72 General: env son strings . . . . .	319
9.73 General: Confunde cuadrado . . . . .	319
9.74 General: Sigue el formato de PEP8 . . . . .	320
9.75 General: Saturar un valor . . . . .	320
9.76 Módulos: Importe circular . . . . .	321
9.77 Módulos: Importe explícito . . . . .	321
9.78 Numpy: Mezcla de tipos . . . . .	322
9.79 Numpy: Usar operaciones vectorizadas . . . . .	322
9.80 Numpy: Uso de mask . . . . .	323
9.81 Numpy: Confundir multiplicación . . . . .	323
9.82 Numpy: Uso de append . . . . .	324
9.83 Numpy: Tamaño y overflow . . . . .	324
9.84 Numpy: Optimizar columnas . . . . .	325
9.85 Pandas: Usa chunksize . . . . .	326
9.86 Pandas: Renombrar columnas . . . . .	326
9.87 Pandas: Usar category . . . . .	327
9.88 Reinventar la rueda: 1 . . . . .	327
9.89 Reinventar la rueda: 2 . . . . .	328
9.90 Reinventar la rueda: 3 . . . . .	328
9.91 Reinventar la rueda: 4 . . . . .	329
9.92 Reinventar la rueda: 5 . . . . .	329
9.93 Reinventar la rueda: 6 . . . . .	330
9.94 Reinventar la rueda: 7 . . . . .	330
9.95 Vulnerabilidad: Usar picke . . . . .	331

---

9.96 Vulnerabilidad: Inyección SQL . . . . .	331
9.97 Vulnerabilidad: No usar eval y exec . . . . .	332
9.98 Vulnerabilidad: Usar pip sin saber qué hay . . . . .	333
9.99 Vulnerabilidad: Request sin timeout . . . . .	333
9.100 Vulnerabilidad: Cuidado con random . . . . .	334
<b>Conclusiones</b>	<b>335</b>

---

# 1 Presentación El Libro De Python



Si has llegado aquí es porque quieres aprender Python. Con El Libro de Python aprenderás todo lo necesario para incorporar este popular lenguaje en tu día a día. Veremos lo siguiente:

- 🔨 Las herramientas que Python ofrece. Tipos, bucles, condicionales, funciones, clases.
- 💡 Cómo usarlas de manera adecuada para solucionar problemas.
- ⚠️ Cómo gestionar cuando las cosas salen mal. Errores, excepciones.
- 🖊️ Cómo escribir código que funcione. Escribiremos tests.
- 💧 Cómo escribir código limpio. *Pythonic* y fácil de leer.
- 🤔 Cómo tomar tus decisiones. Decide qué es mejor.
- 📝 50 ejemplos prácticos viendo el potencial de Python.
- ❌ 100 errores comunes en Python.

Vamos a por ello.

---

## 1.1 ¿Qué es El Libro de Python?

El Libro de Python es un proyecto personal que nació durante la pandemia del COVID-19 en 2020 como forma de afrontar los interminables días de aislamiento.

Comenzó con unas notas personales rápidas para repasar Python, pero terminó convirtiéndose en el blog que muchos conocéis:  
 [www.ellibrodepython.com](http://www.ellibrodepython.com) .

Gracias a todos los que os habéis pasado por él. Gracias por vuestras sugerencias y correcciones.

El blog siempre ha tenido una misión muy clara:

-  Enseñar Python para resolver problemas reales.
-  Gratis y libre para todo el mundo.
-  Sin publicidad, sin cookies, sin cosas raras.
-  En español.

Para mantener esta misión, hemos decidido publicar este libro. Un libro más completo, más cuidado, con más ejemplos, más ilustraciones y para todos los niveles. Una parte de sus beneficios será destinada a mantener el blog y asegurarse de que continúa como bien público y gratuito, fiel a su misión.

Para seguir este libro son necesarios conocimientos básicos de programación. Deberías haber escrito algo de código sencillo anteriormente, saber instalar Python y ejecutar un programa.

Según la regla de Pareto, el 20% de los conceptos son usados el 80% de las veces. En este libro nos centramos en ese 20% que te permitirá entender el 80% de Python. Es el equivalente a estudiar las palabras más usadas de un idioma.

El libro está enfocado en Python como medio, no como fin. Programar

---

ya no es solo para programadores. Es para cualquier profesional que trabaje con datos, cree prototipos, investigue, haga simulaciones, sea profesor o automatice tareas. El libro busca llevar Python a todo el mundo.

Nos enfocamos en los aspectos prácticos con ejemplos reales y dejamos de lado los conceptos complejos que solo utilizan los programadores avanzados.

Es un libro lleno de anglicismos. Consideramos que conceptos como *lazy evaluation*, *tuple unpacking* o *duck typing* no deben ser traducidos. Es mejor usar el original.

Es un libro con vida. Es revisado periódicamente y actualizado con vuestras sugerencias. Si lo has comprado obtendrás las nuevas versiones automáticamente en tu correo.

Es un libro generalista. No pretende abarcar y explicar todos y cada uno de los conceptos existentes en Python, sino dar al lector las herramientas más usadas con aplicaciones prácticas.

## 1.2 ¿Qué no es El Libro de Python?

No pretende ser un manual complicado, lleno de tecnicismos y aburrido de leer.

No pretende cubrir todo. Introduce los conceptos y paquetes más usados de Python con los que podrás hacer de todo.

No pretende centrarse en la teoría entrando en conceptos avanzados que el usuario medio de Python no le interesan. Este libro deja de lado todo lo relacionado con el *metaprogramming*, la manipulación del *abstract syntax tree* o uso del *global interpreter lock*. El 99% de las personas no necesita saber esto.

---

## 1.3 ¿Cómo leer El Libro de Python?

Repite conmigo. A programar se aprende programando. Consumir contenido técnico como este libro no sirve de nada si no se pone en práctica. Otra vez. A programar se aprende programando.

Para leer este libro se recomienda tener un ordenador a mano e ir ejecutando todos los ejemplos. En vez de copiar y pegar, escríbelos. Letra a letra. Línea a línea. Esto te ayudará a entenderlo.

Juega con el código que presentamos en el libro. Experimenta. Cámbialo. Mira que pasa. Recomendamos al lector usar un enfoque de aprendizaje basado en proyectos. Piensa en un pequeño proyecto que te gustaría hacer con Python y úsalo como excusa para aprender. A lo largo del libro te daremos alguna idea.

Recomendamos tener Google a mano para resolver cualquier duda. También sitios como StackOverflow. Y no olvidemos un LLM (Large Language Model) como ChatGPT. Este tipo de herramientas nos puede ayudar cuando tengamos alguna duda, o dar sugerencias. Aunque, por supuesto, hay que tener cuidado ya que aún se equivocan mucho.

Prácticamente todo el código que mostramos se puede ejecutar tal cual. En algunos casos incluimos la salida que el programa produce, lo que facilita la lectura.

Es posible que no entiendas todo en la primera lectura. Es normal. Este libro es un compañero de viaje en tu aprendizaje de Python. Vuelve a él cuando lo necesites.

Por último, enhorabuena por haberte decidido a leer El Libro de Python. Sin duda leer este libro te pondrá por encima del 95% de las personas, que escriben código con copia pegas de Google, StackOverflow y ChatGPT, sin entenderlo y sin que apenas funcione.

---

## 1.4 Más allá de Python

Programar va más allá de simplemente escribir código. Igual que saber todas las palabras de un idioma no te hace hablarlo, saber la sintaxis de Python no te hace programar bien.

Programar es un arte donde se ponen instrucciones sobre un lienzo para resolver un problema. Pero hay que saber usar las herramientas adecuadas.

Aunque este libro versa sobre Python, la finalidad es la de resolver problemas a través de la programación. Veamos algunos consejos para ello:

- 🧠 Cuando un problema parezca muy complicado de abordar, divídelo en subproblemas más pequeños y sencillos. Divide y vencerás. Mejor muchas funciones sencillas que una muy compleja.
- 🔍 No reinventes la rueda. Por raro que sea tu problema es posible que alguien lo haya intentado resolver antes. Busca en Internet.
- 👀 Ten claro el problema que estás resolviendo. Con él bien planteado tienes la mitad resuelto. No lo pierdas nunca de vista.
- ⚖️ Sé consciente de los *trade-offs*. No se puede tener todo de todo. Hay veces que un código más legible es más lento. O un código más corto es ilegible. Elige en función de tu caso. Toda elección implica un coste de oportunidad, algo a lo que renuncias.
- 📄 Cuando tengas un problema que parece no tener salida, aplica la técnica del pato de goma o *rubber duck*. Pon un pato de goma en la mesa y explícaselo. Verbalizarlo te ayudará.
- ✎️ No olvides la diferencia entre validar y verificar. El código puede funcionar, pero si no resuelve tu problema, no servirá

---

de nada. Los perfiles técnicos se centran demasiado en verificar y poco en validar.

- ⚙ El software tiene una complejidad que crece sin límites. La mejor solución es la más sencilla. Se puede matar moscas con un cañón, pero no es lo mejor. La perfección se consigue no cuando no hay nada más que añadir, sino cuando no hay nada más que quitar.
- 🔮 Siempre que resuelvas un problema, ten claro los subproblemas que pueden aparecer cuando las cosas salen mal. Gestiona estos casos. Tarde o temprano acaban apareciendo. Mejor que tu código esté preparado.

## 1.5 Pros/Contras de Python

Python lleva varios años encabezando la lista de lenguajes de programación con más crecimiento y más usados. Por algo será.

Ha democratizado el acceso a la programación, igual que la imprenta de Gutenberg democratizó el acceso a los libros, y las cámaras de Kodak democratizaron el acceso a la fotografía.

Ya no solo los programadores escriben código. Es un lenguaje sencillo pensado para todos. Pero no sirve para todo. Como todo en esta vida, todo tiene sus pros y sus contras.

👉 Es un lenguaje muy fácil de utilizar. No hace falta saber complicados conceptos de programación para usarlo.

👉 Permite iterar muy rápido, convirtiendo en poco tiempo una idea a código. Esto lo hace muy útil en *rapid prototyping*.

👉 Tiene una gran comunidad y existen miles de paquetes que permiten hacer de todo.

---

👍 Gestiona la memoria automáticamente. No tienes que preocuparte por asignar o liberar memoria como en otros lenguajes como C.

👍 Permite actuar como *wrapper* de otros lenguajes de programación. Es decir, permite interactuar por debajo con código más eficiente como C o Rust.

👎 No es el lenguaje más rápido del mercado. Es bastante lento, sobre todo si lo comparas con lenguajes de más bajo nivel.

👎 No es el lenguaje más seguro. Utiliza tipado dinámico, lo que implica que las funciones no tienen un tipo determinado en tiempo de compilación. Nadie utilizaría Python para escribir un software crítico de aviónica.

👎 Python no detecta errores al compilar. Técnicamente, ni siquiera se compila. Otros lenguajes tienen este primer filtro, donde el compilador nos protege de ciertas cosas. Python no. Esta falta de protección lo hace poco seguro.

## 1.6 Paquetes en Python

Un paquete en Python es un código organizado bajo un mismo *namespace*. Es una forma de agrupar un conjunto de código relacionado. Y lo mejor de todo. Puedes publicarlo para que otra gente pueda usarlo.

Python tiene un gestor de paquetes llamado `pip`. Nos permite publicar y descargarnos los paquetes que ha hecho otra gente. Si quieras resolver algún problema, busca ahí porque seguramente alguien haya publicado ya un paquete que lo resuelve.

- 🔗 <https://pypi.org/project/pip/>

---

Hay literalmente de todo. Y esto es lo que hace a Python un lenguaje muy rico. Algunos ejemplos de paquetes conocidos:

- `numpy` : Cálculo numérico y científico.
- `requests` : Gestión de peticiones HTTP.
- `matplotlib` : Representación de gráficos y visualizaciones.
- `tensorflow` : Para *machine learning* e inteligencia artificial.
- `flask` : Creación de aplicaciones web de manera muy fácil.
- `biopython` : Herramientas de bioinformática y manipulación de ADN.
- `astropy` : Algoritmos para astronomía.
- `moviepy` : Edición de video.
- `quarto` : Creación de *dashboards* web.

Puedes instalar cualquiera de ellos usando `pip`.

```
pip install numpy
```

Otra ventaja de `pip` es que cualquiera puede publicar un paquete. Si tienes algo que compartir con el mundo, te invitamos a que lo hagas. Ayudarás a otros con tu trabajo.

Sin embargo no olvidemos que como cualquier persona puede publicar paquetes, la calidad puede variar mucho. Si usas un paquete externo, revisa cuantas descargas o estrellas tiene, investiga si hay otras personas usándolo o hecha un vistazo para ver que no hay nada raro. Existen paquetes con código malicioso que puede ser muy peligroso según tu aplicación. Cuidado.

## 1.7 El Código Pythonic

En Python existe el término *Pythonic*. Tu código puede ser o no *Pythonic*. Por supuesto buscamos que lo sea.

---

Escribir código *Pythonic* significa seguir los principios y convenciones propias del lenguaje. Esto suele ir acorde a código simple, legible y que se beneficie al máximo de la sintaxis de Python.

Esto es similar a las expresiones idiomáticas de un lenguaje. En español se dice “pasarlo bien” y en Inglés te entenderían si dijeras “to pass it well”. Pero es mejor decir “to have a good time”.

Veamos un par de ejemplos con código. Ambos códigos realizan lo mismo y son perfectamente válidos, pero uno representa una forma *Pythonic* de hacer las cosas y el otro no.

✗ Esta forma de iterar una lista no es *Pythonic*.

```
# No es Pythonic
index = 0
for valor in lista:
    print(index, value)
    index += 1
```

✓ Esta forma si es *Pythonic*.

```
# Si es Pythonic
for index, valor in enumerate(lista):
    print(index, valor)
```

✗ Esta forma de ver si un número es par no es *Pythonic*.

```
# No es Pythonic
def es_par(numero):
    if numero % 2 == 0:
        return True
    else:
        return False
```

✓ Esta forma si es *Pythonic*.

```
# Si es Pythonic
def es_par(numero):
```

```
return numero % 2 == 0
```

Más adelante veremos todo esto en detalle. Pero el mensaje es el siguiente. Está bien que las cosas funcionen. Pero está bien intentar que nuestro código sea *Pythonic*. Esto hace que a otra gente le resulte más fácil de leer. También que sea más simple.

## 1.8 Python, ChatGPT y LLMs

En los últimos años se ha producido un *boom* con las Inteligencias Artificiales, ChatGPT o LLM (*Large Language Models*) en general.

Estas herramientas hacen un poco de todo, pero en el caso particular de la programación, son capaces de escribir código relativamente bien. Veamos un ejemplo con ChatGPT.

❓ Le preguntamos lo siguiente.

```
# Tengo una lista l = [6, 4, 7, 81, 0]
# Escribe en Python una función que la ordene.
```

🤖 Y ChatGPT nos responde con el siguiente código.

```
def ordenar_lista(lista):
    return sorted(lista)

l = [6, 4, 7, 81, 0]
ordenada = ordenar_lista(l)

print("Lista ordenada:", ordenada)
# Lista ordenada: [0, 4, 6, 7, 81]
```

❓ Probamos otro ejemplo. Le preguntamos.

```
# Escribe una función en Python para contar
# cuántas vocales hay en una cadena de texto.
```

🤖 Y nos responde con.

---

```
def contar_vocales(cadena):
    vocales = "aeiou"
    return sum(1 for char in cadena.lower() if char in
               vocales)

print(contar_vocales("Hola Mundo"))
# 4
```

En ambos casos clava la respuesta. Ciento es que son ejemplos sencillos y que una búsqueda rápida en Google habría dado el mismo resultado. Pero sin duda nos ahorra perder tiempo navegando por varias páginas web llenas de texto buscando la respuesta.

Estas herramientas funcionan relativamente bien, sobre todo cuando preguntamos cosas sencillas. Pero se complica cuando preguntamos sobre paquetes externos no muy usados, cuando es necesario contexto sobre un repositorio grande de código o con conceptos avanzados.

Aun así sirven de ayuda para que alguien que no conozca Python pueda expresar lo que desea en lenguaje natural y obtener un código a cambio. Y que en muchas ocasiones funciona.

Llegados a este punto tal vez digas: “Aprender Python es una tontería, ChatGPT me lo hace todo. Ya no es necesario saber programar.”

No creemos que esto sea así. Las Inteligencias Artificiales aún tienen alucinaciones. Es decir, se inventan cosas que no tienen sentido. Por otro lado, funcionan bien con ejemplos sencillos, pero cuando las cosas se complican empiezan a fallar. Y por último, alguien con una buena base de Python es capaz de escribir mejores *prompt*, por lo que saber Python importa.

Un *prompt* es simplemente la entrada, la pregunta que le das a ChatGPT. En este libro, entre otras cosas, te damos criterio para no tragarte cualquier tontería que ChatGPT suelte.

---

Cierto es que estas herramientas están mejorando a una velocidad increíble, y es posible que en unos años cambien la forma de programar de forma radical.

Para anticiparte a este cambio, te recomendamos empezar a probar alguna de las siguientes herramientas:

- **ChatGPT**: Creado por OpenAI, el primer *chatbot*. Úsallo para preguntarle cosas de Python.
- **Claude**: Similar a ChatGPT.
- **Cursor**: Entorno de desarrollo basado en VSCode que integra de forma nativa cualquiera de los anteriores. Úsallo para programar en Python y “hablar” con tu código.

Un apunte extra sobre Cursor. ChatGPT y Claude son *chatbots* genéricos, pero Cursor es una herramienta centrada en el desarrollo software y programación. Es un entorno de desarrollo o editor de código con un asistente de Inteligencia Artificial integrado. Algunas características interesantes:

- Permite la predicción en tiempo real de código. Da sugerencias bastante interesantes.
- Permite dar contexto a tus consultas incluyendo otros ficheros, documentación o enlaces.

Dicho esto, veamos algunos consejos para usar Inteligencia Artificial en programación:

- Incluye el máximo detalle en tu *prompt*. No es lo mismo pedir un código para procesar unos pocos datos que millones. Cuánto más detalle y contexto mejor.
- Pide que te explique el código. Te ayudará a entender las cosas y no copiar pegar sin entender nada.

- 
- ⚠ En algunas ocasiones, puede responder con código que no funciona, que no compila. En ese caso, pregunta de nuevo incluyendo el error que has obtenido.
  - 🧠 Nunca copies y pegues código salido de ChatGPT o similar sin haberlo entendido antes.

Como puedes ver, es una herramienta excelente, pero tenemos que tener un espíritu crítico ante las respuestas que nos da, y no conformarnos con lo primero. Úsalo, pero con precaución.

## 1.9 Primeros pasos con Python

Como en todo lenguaje de programación, nuestro primer paso es mostrar por pantalla el mensaje “Hola Mundo”. Crea un fichero `ejemplo.py` y añade el siguiente código.

```
print("Hola mundo")
# "Hola mundo"
```

Lo podrás ejecutar de diferentes maneras:

- Con `python ejemplo.py` en tu terminal.
- Con tu entorno de desarrollo favorito. PyCharm, VS Code, Cursor.

Como puedes ver, incluimos un comentario `#` con la salida que produciría el código si fuera ejecutado. Lo mostramos de aquí en adelante en todo el libro.

Ahora vamos a definir unas variables. Una de tipo `str` y otra de tipo `int`.

```
nombre = "Alicia"
edad = 20
print(f"{nombre} tiene {edad} años")
```

```
# Alicia tiene 20 años
```

Ahora creamos una `list`, similar a los *array* de otros lenguajes. Permiten almacenar un conjunto de varios datos, en este caso de `int`.

```
edades = [18, 25, 32]
```

Realizamos unas operaciones con nuestra `list`. Calcular la media.

```
print(sum(edades)/len(edades))  
# 25.0
```

Podemos también crear funciones.

```
def media(edades):  
    return sum(edades)/len(edades)
```

Y usarla.

```
print(media(edades))  
# 25.0
```

Python tiene un comportamiento muy interesante en el siguiente caso. En otros lenguajes esto daría `0`.

```
print(2/4)  
# 0.5
```

También podemos añadir comentarios.

```
# Almacenamos las edades en una lista  
edades = [18, 25, 32]
```

Podemos iterar las listas.

```
for edad in edades:  
    print(edad)
```

---

Y para que veas el humor de los creadores de Python, hay algún que otro *easter egg*. Ejecuta el siguiente código.

```
import this
```

O el siguiente.

```
import antigravity
```

Como puedes ver, Python hace mucho con muy poco. La sintaxis es sencilla y se asemeja mucho al lenguaje natural. Esto lo diferencia de otros lenguajes de programación, que requieren años de estudio y están diseñados para ser usados por profesionales.

Python pone al alcance de cualquiera la capacidad de hablar con una máquina para pedirle que haga una tarea determinada.

---

## 2 Elige Tipo y Estructura de Datos



Un diccionario almacena palabras. Una caja de herramientas almacena herramientas. De la misma manera, Python ofrece diferentes contenedores para almacenar información. Los dividimos en:

- 📚 **Tipos de datos:** Lo más básico y fundamental. Por ejemplo `int` puede almacenar números `7` y `str` texto `"python"`.
- 📁 **Estructuras de datos:** Permiten almacenar datos más complejos con múltiples elementos, tipos de datos mezclados y operaciones complejas sobre ellos. Tenemos `list`, `dict` o `set`. Y otros tantos usando paquetes externos.

Cada uno tiene unas ventajas e inconvenientes. Veamos los diferentes tipos y estructuras de datos que Python ofrece y algunos consejos para decidir cual elegir.

---

## 2.1 Tipos de Datos

### 2.1.1 Int

Los `int` permiten almacenar números sin contenido decimal.

```
i = 12  
print(i)      # 12
```

Con `type` podemos ver el tipo de nuestra variable `i`, que es efectivamente `int`.

```
print(type(i)) # <class 'int'>
```

Si vienes de otros lenguajes donde hay múltiples tipos de `int` como `int16`, `int32`, `uint64`, tenemos buenas noticias. En Python solo existe uno y permite almacenar valores muy grandes.

```
x = 250**250  
print(x)
```

Se asigna memoria según sea necesario sin que el programador deba preocuparse. Los siguientes valores ocupan un número diferente de bytes en memoria. Como cabría esperar, el `5**10000` necesita más memoria que el `10`.

```
import sys  
print(sys.getsizeof(5**10000)) # 3120  
print(sys.getsizeof(10))       # 28
```

Los `int` soportan las operaciones comunes, siendo `**` el exponente.

```
a, b = 3, 6  
print(a+b)  # 9  
print(a-b)  # -3  
print(a*b)  # 18
```

```
print(a/b) # 0.5
print(a**b) # 729
```

Y pueden ser declarados tanto en forma decimal como binaria, hexadecimal y octal. Esto es el número 25 expresado con diferentes bases.

```
a = 0b11001 # 25 en binario
b = 0x19    # 25 en hexadecimal
c = 0o031   # 25 en octal
d = 25      # 25 en decimal

print(a, type(a)) #25 <class 'int'>
print(b, type(b)) #25 <class 'int'>
print(c, type(c)) #25 <class 'int'>
print(d, type(d)) #25 <class 'int'>
```

A modo de curiosidad, el siguiente código tiene un resultado diferente en Python versión 2 que 3. En este libro nos centramos en Python 3, que es la versión más moderna.

```
print(1/2) # Python 2: 0
print(1/2) # Python 3: 0.5
```

## 2.1.2 Float

Los float permiten almacenar valores decimales.

```
f = 0.35
print(f)        # 0.35
print(type(f)) # <class 'float'>
```

También admiten notación científica.

```
f = 1.93e-3
```

Y para sorpresa de muchos, la precisión no es infinita. En este caso

---

`f` no es realmente `1.0`, pero se almacena como tal.

```
f = 0.9999999999999999
print(f)      # 1.0
print(1 == f) # True
```

Un `float` también puede almacenar el valor infinito.

```
x = float('inf')
print(x)      # inf
print(type(x)) # <class 'float'>
```

Dado que no existe precisión infinita hay números que no se pueden almacenar exactamente. Por ejemplo, el número `1.3` no se puede almacenar exactamente.

Como puedes ver, en el decimal 1.3 existe un pequeño error. Para ampliar información, busca sobre el estándar IEEE 754.

```
f = 1.3
print(f"{f:.20}")
# 1.3000000000000000444
```

No podría faltar la prueba por excelencia de todo lenguaje de programación. Las matemáticas nos enseñaron que el resultado sería `0`, pero Python, al igual que otros lenguajes nos sorprende con lo siguiente.

```
print(0.1 + 0.1 + 0.1 - 0.3)
# 5.551115123125783e-17
```

A efectos prácticos este tipo de errores afectan a muy poca gente. La mayoría de los usuarios de Python lo puede ignorar.

También es posible redondear un `float` de varias maneras:

- ⚡ Redondeo más cercano.
- ⌛ Redondeo hacia arriba.

- 
- ⤵ Redondeo hacia abajo.

```
import math

f = 2.675
print(round(f, 2))      # 2.67
print(math.ceil(f))     # 3
print(math.floor(f))    # 2
```

Y como los `int`, tenemos las operaciones matemáticas comunes.

```
a, b = 3.3, 6.5
print(a+b)    # 9.8
print(a-b)    # -3.2
print(a*b)    # 21.45
print(a/b)    # 0.5076923076923077
print(a**b)   # 2346.0680721890503
```

### 2.1.3 Boolean

Los `bool` permiten almacenar un tipo que solo toma dos valores: `True` o `False`.

```
b = True
print(b)          # True
print(type(b))  # <class 'bool'>
```

Python permite convertir a `bool` cualquier cosa, interpretando:

- ☀️ `True`: Cuando no es cero o no está vacío como `list` o `dict` vacíos.
- ⚪ `False`: Cuando es cero o está vacío.

Es decir, fíjate en los siguientes ejemplos.

```
print(bool(10))    # True
print(bool(0))     # False
print(bool([]))    # False
```

```
print(bool([1])) # True  
print(bool(None)) # False
```

Aunque lo veremos más adelante en la sección de listas y condicionales, gracias a esto se puede hacer lo siguiente.

```
# Forma Pythonic  
l = []  
if not l:  
    print("Lista vacía")
```

El siguiente código sería equivalente, pero no es una forma *Pythonic* de hacerlo.

```
# Forma NO Pythonic  
if len(l) == 0:  
    print("Lista vacía")
```

Existen otras curiosidades como la siguiente. Python interpreta el `True` como `1`.

```
print(True + True) # 2  
print(True * 10) # 10
```

Y es común usar sus operadores lógicos como:

- `and` : Es `True` si ambos valores son `True` , `False` de lo contrario.
- `or` : Es `True` si al menos un valor es `True` , `False` de lo contrario.
- `not` : Invierte `True` por `False` y viceversa.

```
print(True and False) # False  
print(True or False) # True  
print(not True) # False
```

Por último, soporta los operadores matemáticos vistos anteriormente.

```
a, b = True, True
print(a+b) # 2
print(a-b) # 0
print(a*b) # 1
print(a/b) # 1.0
print(a**b) # 1
```

## 2.1.4 String

Los `str` permiten almacenar *strings*, o texto. Se pueden definir con comillas `" "`.

```
s = "el libro de python"
print(s)          # el libro de python
print(type(s)) # <class 'str'>
```

O también con comillas simples `' '`.

```
s = 'el libro de python'
```

O triples `"""` si queremos varias líneas.

```
print("""el libro
de
python""")
```

Existen múltiples formas combinar `str` con otros tipos como `int`.

```
x = 5
s = "El número es: %d" % x
print(s) # El número es: 5
```

También con `%` similar a lenguajes como C.

```
s = "Los números son %d y %d." % (5, 10)
print(s) #Los números son 5 y 10.
```

---

Con `format`.

```
s = "Los números son {} y {}".format(5, 10)
print(s) # Los números son 5 y 10
```

Con `format` y el nombre de la variable.

```
s = "Los números son {a} y {b}".format(a=5, b=10)
print(s) # Los números son 5 y 10
```

Pero sin ninguna duda, nuestra forma preferida es usando los *f-string* introducidos en la PEP 498. Fíjate en la `f` que se usa antes del `"`.

```
a, b = 5, 10
s = f"Los números son {a} y {b}"
print(s) # Los números son 5 y 10
```

Son muy potentes ya que permiten realizar operaciones dentro.

```
a, b = 5, 10
s = f"a + b = {a+b}"
print(s) # a + b = 15
```

Y también permiten redondear los `float`. Usando `.2f` podemos redondear a 2 decimales. Es importante notar que esto no modifica el contenido de la variable.

```
f = 123.456789
print(f"{f:.2f}")
# 123.46
```

Por otro lado, los `str` tienen diferentes operadores asociados. Con `in` podemos saber si un *string* está contenido en otro.

```
print("Python" in "El Libro De Python")
# True
```

El operador suma `+` esta definido sobre `str` y permite unir dos o más.

```
a, b = "El Libro", "De Python"  
print(a + " " + b)  
# El Libro De Python
```

Curiosamente el operador multiplicación `*` también. Repite el `str` tantas veces como indiquemos.

```
s = "Python"  
print(s*3)  
# PythonPythonPython
```

Con `len()` podemos saber la longitud.

```
print(len("El Libro De Python"))  
# 18
```

Dado que `str` es iterable, podemos indexarlo. Así podemos acceder a la primera letra.

```
x = "El Libro De Python"  
print(x[0]) # E
```

Y a la última letra.

```
x = "El Libro De Python"  
print(x[-1]) # n
```

O varios elementos. Veremos esto en más detalle cuando trabajemos con listas.

```
x = "El Libro De Python"  
print(x[0:8]) # El Libro  
print(x[9::]) # De Python
```

El tipo `str` ofrece otros muchos métodos. Algunos ejemplos.

```
print(x.lower())      # el libro de python  
print(x.split(" ")) # ['El', 'Libro', 'De', 'Python']
```

## 2.1.5 Enum

Los `enum` permiten definir un tipo propio que solo puede tomar unos valores determinados. Por ejemplo, diferentes monedas.

-  EURO
-  DOLAR

```
from enum import Enum
class Moneda(Enum):
    DOLAR = 1
    EURO = 2

c = Moneda.DOLAR
print(c)          # Moneda.DOLAR
print(type(c))   # <enum 'Moneda'>
```

Podemos iterar el `enum` para saber todos sus posibles valores

```
for c in Moneda:
    print(c)
# Moneda.DOLAR, Moneda.EURO
```

Aunque lo siguiente es posible, no lo hagas.

```
print(Moneda['DOLAR']) # Moneda.DOLAR
print(Moneda(1))        # Moneda.DOLAR
```

Mejor haz esto. Más legible, menos propenso a errores, más fácil de mantener, y tu entorno de desarrollo seguramente ofrecerá auto completado.

```
print(Moneda.DOLAR)
```

Los `enum` permiten usar variables definidas como `Moneda.EURO` o `Moneda.DOLAR` en vez de usar cadenas como `"euro"` o `"dolar"` propensas a ser mal escritas sin un tipo concreto.

## 2.1.6 None

El tipo `None` nos permite indicar una variable sin valor asignado, similar al *null*, *nil* u *Option* en otros lenguajes de programación.

```
x = None
print(x)          # None
print(type(x))   # <class 'NoneType'>
```

Se trata de un *singleton* ya que cualquier `None` es igual a `None`.

```
a = None
b = None

print(a is None)  # True
print(b is None)  # True
print(a is b)     # True
print(a == b)     # True
```

Es común ver `None` en las funciones. Una función devuelve `None` si no tiene un `return` explícito.

```
def funcion():
    pass

print(funcion()) # None
```

También los diccionarios devuelven `None` para *keys* que no existen.

```
dic = {}
print(dic.get('noexiste'))
# None
```

## 2.2 Conversión Entre Tipos

Es posible realizar conversiones entre tipos de dos formas:

- 
- 🤔 **Implícito**: No se indica nuestra intención de hacer la conversión, pero Python la hace automáticamente.
  - 💡 **Explícito**: Si se indica la intención de hacer la conversión, y esta puede dar lugar a efectos no deseados como la pérdida de precisión numérica.

💡 Veamos conversiones implícitas. Tenemos dos `int` pero después de dividir, nos encontramos con un `float`. Se ha producido una conversión implícita.

```
a, b = 3, 2
print(type(a))    # <class 'int'>
print(type(b))    # <class 'int'>
print(type(a/b)) # <class 'float'>
```

Tenemos un `bool` y un `int` y el resultado es un `int`. El `bool` se ha convertido para poder realizar la operación. Otra conversión implícita.

```
print(True + 1)
# 2
```

Tenemos un `int` que se evalúa como un `bool`. Esta es una de nuestras características favoritas de Python que no todos los lenguajes ofrecen. Otra conversión implícita.

```
if 1:
    print("True")
```

💡 Ahora veamos conversiones explícitas. Podemos convertir de `int` a `float`. Como ves, la diferencia es que tenemos que ser explícitos, indicando que queremos realizar la conversión.

```
a = 1
b = float(a)

print(type(a)) # <class 'int'>
```

```
print(type(b)) # <class 'float'>
```

Pero esto supone una pérdida de decimales. Teníamos 5.3 pero un int sólo puede representar 5.

```
a = 5.3  
b = int(a)  
  
print(b) # 5
```

Lo mismo entre int y bool. Tenemos un 7 pero lo más parecido que puede representar bool es True.

```
a = 7  
b = bool(a)  
  
print(b) # True
```

Y similar entre str y bool.

```
print(bool(""))      # False  
print(bool("Python")) # True
```

Aunque no siempre podemos convertir todo. Un str no puede ser representado como int. En este caso obtenemos un error directamente.

```
a = int("El Libro De Python")  
# ValueError: invalid literal
```

## 2.3 Estructuras de Datos

### 2.3.1 Listas

La estructura list nos permite almacenar un conjunto de datos en una variable. Tanto del mismo tipo.

```
l = [1, 2, 3]
print(l)          # [1, 2, 3]
print(type(l))   # <class 'list'>
```

Como de tipos diferentes.

```
l = [1, "Python", 3]
print(l)
# [1, 'Python', 3]
```

Se puede acceder a los elementos por su índice. Con `l[0]` accedemos a su primer elemento.

```
print(l[0])
# 1
```

A diferencia de otros lenguajes, Python nos protege para que no accedamos a un índice fuera del rango de la lista. Aunque pueda parecer lógico, otros lenguajes no lo ofrecen. Esto es una fuente de bugs y vulnerabilidades de lo más variadas. Pero no en Python.

Si intentamos acceder al elemento `l[10]` nos dará el siguiente error porque solo existen `3`.

```
l = [1, 2, 3]
print(l[10])
# IndexError: list index out of range
```

Puedes acceder a cada elemento de la lista de la siguiente manera. Esto se conoce como iterar.

```
for i in l:
    print(i)
# 1
# 2
# 3
```

Con `len()` podemos saber su tamaño.

```
print(len([1, 2, 3]))  
# 3
```

Con `append()` podemos añadir elementos.

```
a = [1, 2]  
a.append(3)  
print(a) # [1, 2, 3]
```

Con `reverse()` invertir el orden.

```
a = [1, 2, 3]  
a.reverse()  
print(a) # [3, 2, 1]
```

Y también definen operadores como multiplicación `*` y suma `+`.

```
print([1, 2] * 2)      # [1, 2, 1, 2]  
print([1, 2] + [3, 4]) # [1, 2, 3, 4]
```

Podemos crear `list` de muchas formas. En capítulos posteriores explicaremos esto en detalle, viendo *list comprehensions* y generadores. De la siguiente manera podemos crear una lista que contiene los números del `0` al `4`. Vemos dos formas de hacerlo.

```
l1 = [i for i in range(5)]  
l2 = list(range(5))  
  
print(l1) # [0, 1, 2, 3, 4]  
print(l2) # [0, 1, 2, 3, 4]
```

Antes hemos visto como acceder a un elemento en concreto con `1[0]`, pero podemos acceder a varios con diferentes criterios. Esto se conoce como *slicing*. Algunos ejemplos:

- `1[0:2]` : Toma los 2 primeros elementos.
- `1[4:]` : Toma del cuarto elemento al final.

- 
- `l[::2]` : Toma un elemento si un elemento no.
  - `l[-1]` : Toma el último elemento.

```
l = [1, 2, 3, 4, 5, 6]
print(l[0:2])    # [1, 2]
print(l[4:])     # [5, 6]
print(l[::-2])   # [1, 3, 5]
print(l[-1])     # 6
```

Por otro lado, el siguiente ejemplo causa mucha confusión.

- Creamos `a`. Creamos `b=a`.
- Cambiamos `b`. Pero también `a` cambia.

Esto no resulta muy intuitivo. Al hacer `b=a` estamos haciendo que ambas variables sean en realidad una referencia de la misma.

```
a = [1, 2, 3]
b = a
b[0] = 0

print(a) # [0, 2, 3]
print(b) # [0, 2, 3]
```

Esto se debe a que las `list` son un tipo mutable. Esto implica que al hacer `b = a` estamos haciendo que `b` sea una referencia a `a`, pero no una copia. Si haces lo mismo con un `int`, verás que el comportamiento es diferente. Esto es porque un `int` es un tipo inmutable.

Para salir de dudas podemos usar el operador `is`. Si devuelve `True` significa que ambas variables son en realidad lo mismo. Si conoces el término, puntero, esto es algo similar.

```
print(a is b) # True
```

Ahora imagina que quieres que `b` sea una copia independiente.

---

Existen varias formas de clonar la lista en una nueva. Aunque ten en cuenta que al duplicarlo, el consumo de memoria será el doble. Irrelevante con listas pequeñas, pero muy importante si trabajas con GB de datos.

A continuación puedes ver diferentes formas de clonar una lista.

```
import copy

a = [1, 2, 3]

b1 = a
b2 = a.copy()
b3 = a[:]
b4 = list(a)
b5 = copy.copy(a)
b6 = copy.deepcopy(a)

print(a is b1) # True
print(a is b2) # False
print(a is b3) # False
print(a is b4) # False
print(a is b5) # False
print(a is b6) # False
```

Existe también otro matiz que causa mucha confusión. No es lo mismo:

-  `copy` : Se hace una copia del objeto superficial, pero no de los anidados.
-  `deepcopy` : Se hace una copia completa del objeto y sus estructuras anidadas.

En otras palabras. Si tienes una lista dentro de una lista y quieres hacer una copia independiente, deberás usar `deepcopy`.

```
import copy

a = [[1, 2], [3, 4]]
```

```
b7 = copy.copy(a)
b8 = copy.deepcopy(a)

a[0][0] = 0

print(b7) # [[0, 2], [3, 4]]
print(b8) # [[1, 2], [3, 4]]

print(a[0] is b7[0]) # True
```

Es importante notar que esto aplica a cualquier tipo mutable, no sólo a las `list`.

### 2.3.2 Tuplas

La estructura `tuple` permite almacenar valores de manera similar a las `list`.

```
t = (1, 2, 3)
print(t)          # (1, 2, 3)
print(type(t)) # <class 'tuple'>
```

También se pueden definir así.

```
t = 1, 2, 3
```

Y a diferencia de las `list`, las `tuple` son inmutables.

```
t = (1, 2, 3)
#t[0] = 5 # Error! TypeError
```

Se pueden indexar. Así accedemos al primer elemento.

```
print(t[0])
# 1
```

Se pueden iterar.

```
tupla = [1, 2, 3]
for t in tupla:
    print(t)
# 1, 2, 3
```

Se pueden anidar. Es decir, tener una dentro de otra.

```
t = 1, 2, ('a', 'b'), 3
print(t)          # (1, 2, ('a', 'b'), 3)
print(t[2][0])  # a
```

Se puede convertir de `list` a `tuple`.

```
lista = [1, 2, 3]
tupla = tuple(lista)
```

Y una herramienta muy potente es el *tuple unpacking*. Esto permite asignar una `tuple` con varios elementos a variables. En una sola línea.

```
l = (1, 2, 3)
x, y, z = l
print(x, y, z) # 1 2 3
```

### 2.3.3 Sets

La estructura `set` permite almacenar valores, pero no permite duplicados ni garantiza el orden. Puedes ver como los duplicados se eliminan al crear el set.

```
s = set([2, 2, 1, 1])
print(s)          # {1, 2}
print(type(s)) # <class 'set'>
```

Podemos ver su tamaño con `len()`.

```
s = {1, 2, 3, 4}
```

```
print(len(s)) # 4
```

Con `add()` añadimos elementos.

```
s = {1, 2}  
s.add(3)  
print(s) # {1, 2, 3}
```

Con `remove()` eliminamos un valor concreto.

```
s = {1, 2}  
s.remove(2)  
print(s) # {1}
```

Aunque los `set` son mutables, los elementos que almacena deben ser inmutables. Dado que `list` es mutable, no podemos usarla en un `set`.

```
s = {1, [1, 2]}  
# TypeError: unhashable type: 'list'
```

Ofrecen también algunos operadores especiales:

- ⚡ `union` o `|` : Unión entre dos set, combinación de los valores de ambos set.
- ✎ `intersection` o `&` : Intersección entre dos set, resultado de tomar los valores comunes.
- - `difference` o `-` : Diferencia entre dos set, los valores que no existen en ambos.

```
print({1, 2} | {3}) # {1, 2}  
print({1, 2} & {2, 3}) # {2}  
print({1, 2} - {2}) # {1}
```

O equivalente.

```
print({1, 2}.difference({2})) # {1}  
print({1, 2}.union({3})) # {1, 2, 3}
```

```
print({1, 2}.intersection({2, 3})) # {2}
```

### 2.3.4 Diccionarios

El tipo `dict` permite almacenar datos en forma de clave-valor, donde la clave es única.

- `nombre` y `edad` son las claves.
- `Sara` y `27` son los valores.

```
d = {"nombre": "Sara", "edad": 27}  
print(d)          # {'nombre': 'Sara', 'edad': 27}  
print(type(d))   # <class 'dict'>
```

Una forma alternativa de crear un `dict` es de la siguiente forma. Es equivalente a la forma anterior.

```
d = dict(nombre='Sara', edad=27)  
print(d) # {'nombre': 'Sara', 'edad': 27}
```

O usando `dict` con una `list` de `tuple`. Todas formas de hacer lo mismo.

```
d = dict([('nombre', 'Sara'), ('edad', 27)])  
print(d) # {'nombre': 'Sara', 'edad': 27}
```

Podemos acceder al valor dada una clave.

```
print(d['nombre'])  
# Sara
```

Aunque si la clave no existe tendremos una excepción `KeyError`.

```
print(d["noexiste"])  
# KeyError: 'noexiste'
```

Una forma más segura que nos evita la excepción es usando `get()`.

```
print(d.get('nombre')) # Sara
```

Si no existe, en este caso obtendremos `None` en vez de `KeyError`.

```
print(d.get('noexiste')) # None
```

Podemos pasar a `get()` el valor a usar cuando no se encuentre. En este caso obtendremos `''` en vez de `None`.

```
print(d.get('noexiste', ''))
```

Por otro lado podemos modificar el valor de una clave.

```
d1['nombre'] = "Laura"
```

También podemos añadir una nueva clave.

```
d = {"nombre": "Sara", "edad": 27}
d["nuevaclave"] = "valor"
print(d)
# {'nombre': 'Sara', 'edad': 27, 'nuevaclave': 'valor'}
```

Podemos iterar los `dict` viendo todas sus claves.

```
d = {"nombre": "Sara", "edad": 27}
for x in d:
    print(x)
# nombre, edad
```

Y si queremos las claves y los valores con `items()`.

```
d = {"nombre": "Sara", "edad": 27}

for x, y in d.items():
    print(x, y)
# nombre Sara
# edad 27
```

### 2.3.5 NamedTuple

La estructura `namedtuple` permite almacenar valores en una tupla con un nombre asociado. Es similar a crear una clase como veremos más adelante.

```
from collections import namedtuple

Coordenada = namedtuple('Coordenada', ['x', 'y', 'z'])
c = Coordenada(10, 20, 30)

print(c)          # Coordenada(x=10, y=20, z=30)
print(type(c))   # <class '__main__.Coordenada'>
```

Y podemos acceder a los valores por su nombre.

```
print(f"x={c.x} y={c.y} z={c.z}")
# x=10 y=20 z=30
```

### 2.3.6 Counter

La estructura `Counter` permite almacenar elementos inmutables como `int` y nos ofrece algunas operaciones interesantes.

```
from collections import Counter

c = Counter([25, 25, 30, 32, 32, 35])

print(c)          # Counter({25: 2, 32: 2, 30: 1, 35: 1})
print(type(c))   # <class 'collections.Counter'>
```

El más interesante es `most_common()` que nos dice qué valor es el más repetido y cuantas veces lo está. Es decir, en nuestra lista el valor `25` es el mas repetido con `2` veces.

```
print(c.most_common(1))
```

```
#[(25, 2)]
```

### 2.3.7 Clases

Más adelante veremos la Programación Orientada a Objetos y las `class`. Si las estructuras y tipos que hemos visto no te gustan, puedes crear una propia con el comportamiento que deseas definiendo tu propia `class`.

## 2.4 Estructuras en NumPy

El paquete `numpy` ofrece entre otras cosas estructuras de datos como `np.array`. Aunque no es parte de Python es un estándar *de facto* ampliamente usado. Las principales ventajas:

- ⚡ Velocidad: Lo que usas es código Python, pero el código que se ejecuta por debajo es C, por lo que es muy eficiente y rápido.
- 🦉 Funcionalidades: Ofrece gran cantidad de funcionalidades como funciones estadísticas, trabajo con polinomios, matrices o transformadas.
- 😊 Comunidad: Gran comunidad de usuarios con documentación extensa y muchos recursos de aprendizaje.

Empecemos por el tipo `np.array`. Permite definir algo similar a una `list`.

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr)          # [1 2 3]
print(type(arr))   # <class 'numpy.ndarray'>
```

Podemos usar el *slicing* visto anteriormente con `list` de la misma forma.

```
print(arr[0])      # 1
print(arr[-1])     # 3
print(arr[::-1])   # [3 2 1]
```

Ofrece funciones estadísticas muy usadas, parte del objeto. Esta es la primera diferencia con `list`.

```
print(arr.mean()) # 2.0
print(arr.std())  # 0.816496580927726
print(arr.cumsum()) # [1 3 6]
```

Y también tiene un amplio rango de funciones externas que pueden ser usadas. Podemos ver el percentil `50%` de nuestros valores.

```
print(np.percentile(arr, 50)) # 2.0
```

Con `all` si todos los elementos cumplen una condición. En este caso miramos los valores que son positivos.

```
todos_positivos = np.all(arr > 0)
```

Con `any` podemos ver si al menos un elemento cumple una condición.

```
algun_positivo = np.any(arr > 0)
```

Con `unique` obtenemos los elementos sin duplicados.

```
arr = np.array([1, 2, 2, 3, 3, 3])
print(np.unique(arr)) # [1 2 3]
```

Con `sort` podemos ordenar los valores.

```
arr = np.array([3, 1, 2])
print(np.sort(arr))
# [1 2 3]
```

---

Podemos usar varias dimensiones, desde dos.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr[0][0]) # 1
print(arr[1][2]) # 6
```

Hasta un número arbitrario de dimensiones.

```
arr = np.array([[[1, 2], [3, 4]], [[7, 8], [9, 10]]])
print(arr[0][0][0]) # 1
print(arr[1][1][0]) # 9
```

Cuando existen varias dimensiones, podemos usar `axis` para determinar si queremos realizar el cálculo sobre:

- 🌎 Todo: Se calcula la `mean` de todos los valores.
- 🤡 Vertical: Se calcula la `mean` de cada columna.
- 🤡 Horizontal: Se calcula la `mean` de cada fila.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.mean()) # 3.5
print(arr.mean(axis=0)) # [2.5 3.5 4.5]
print(arr.mean(axis=1)) # [2. 5.]
```

También permite lo que se conoce como *broadcasting*. Multiplica el escalar por cada elemento del `np.array`.

```
arr = np.array([1, 2, 3]) * 3
print(arr)
# [3 6 9]
```

Esto es muy diferente a hacerlo con la `list` que vimos anteriormente

```
print([1, 2, 3] * 3)
# [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El *broadcasting* también funciona con la suma `+`.

```
arr = np.array([1, 2, 3])
print(arr + 10)
# [11 12 13]
```

Y con varias dimensiones.

```
arr1 = np.array([1, 2])
arr2 = np.array([[10], [100]])

print(arr1 + arr2)
# [[ 11 12]
#  [101 102]]
```

Pero si las dimensiones no son compatibles, no funcionará.

```
arr1 = np.array([1, 2])
arr2 = np.array([[10, 100, 1000]])

print(arr1 + arr2)
# ValueError: operands could not be broadcast together
# with shapes (2,) (1,3)
```

El `np.array` también nos permite asignar un nombre a cada columna.

```
arr = np.array([('Alicia', 25), ('Bob', 30)],
               dtype=[('nombre', 'U20'), ('edad', 'uint8')])
```

Podemos acceder a cada columna con `[]`.

```
print(arr['nombre'])
# ['Alicia' 'Bob']
```

Pueden ser iterados.

```
for i in arr:
    print(i)
# ('Alicia', 25)
# ('Bob', 30)
```

---

Pero ten cuidado con el `dtype`. Si eliges mal el tipo puedes acabar almacenando algo incorrecto. No podemos usar `uint8` para almacenar decimales.

```
arr = np.array([1.1, 2.3, 4.75], dtype=np.uint8)
print(arr)
# [1 2 4]
```

## 2.5 Estructuras en Pandas

El paquete `pandas` tampoco es parte de Python, pero es otro estándar *de facto* similar a `numpy`.

La estructura de datos `pd.DataFrame` permite almacenar datos en una tabla. Cada columna tiene un nombre y almacena un conjunto de valores.

```
import pandas as pd

df = pd.DataFrame({
    'TemperaturaC': [33, 35, 37, 31, 32, 29, 28],
    'Lluvia': [0, 0, 0, 1000, 300, 0, 0])
print(type(df))
# <class 'pandas.core.frame.DataFrame'>
```

Las dimensiones deben ser iguales. De lo contrario obtenemos un error.

```
df2 = pd.DataFrame({
    'a': [33, 35],
    'b': [0]})

# ValueError: All arrays must be of the same length
```

Podemos acceder a cada columna con el nombre.

---

```
print(df.TemperaturaC)
```

Aunque también podemos así.

```
print(df["TemperaturaC"])
```

Tenemos métodos como la media `mean`.

```
print(df.TemperaturaC.mean())
# 32.142857142857146
```

Podemos calcular el incremento entre valores consecutivos con `diff`.

```
print(df.TemperaturaC.diff())
```

Podemos convertir cada columna en una `list`.

```
print(df.TemperaturaC.to_list())
# [33, 35, 37, 31, 32, 29, 28]
```

Como puedes ver por defecto está indexado desde `0` a `n-1`. Pero podemos cambiar nuestro `df` a usar los días de la semana como índice.

```
df.index = ['Lunes', 'Martes', 'Miercoles', 'Jueves',
           'Viernes', 'Sabado', 'Domingo']
```

Ahora podemos obtener temperatura y lluvia para el lunes.

```
print(df.loc['Lunes'])
```

O la temperatura de un día concreto.

```
print(df.TemperaturaC.Domingo)
```

Podemos crear un `pd.DataFrame` nuevo filtrando el existente. Por ejemplo, donde ha llovido.

```
print(df[df['Lluvia'] > 0])
#           TemperaturaC   Lluvia   TemperaturaF
# Jueves          31     1000        87.8
# Viernes         32      300        89.6
```

También podemos ordenar por columna.

```
print(df.sort_values(by='TemperaturaC', ascending=
    False))
#           TemperaturaC   Lluvia
# Miércoles       37       0
# Martes          35       0
# Lunes           33       0
# ...
```

Con `describe` podemos obtener información estadística como media, desviación estándar o percentiles.

```
print(df.TemperaturaC.describe())
# count      7.000000
# mean      32.142857
# std       3.184785
# min      28.000000
# 25%      30.000000
# 50%      32.000000
# 75%      34.000000
# max      37.000000
```

También podemos crear nuevas columnas en la tabla. Por ejemplo una columna con la temperatura en grados Fahrenheit.

```
df['TemperaturaF'] = df['TemperaturaC'] * 9/5 + 32
```

Una de las características más usadas de `pandas` es su integración con `matplotlib`, lo que nos permite representar gráficamente prácticamente lo que queramos. Por ejemplo, representamos la evolución de la temperatura a lo largo de la semana.

---

```
import matplotlib.pyplot as plt
df.Temperatura.plot()
plt.xlabel("Día")
plt.ylabel("Temperatura (°C)")
plt.title("Evolución Temperatura")
plt.show()
```

## 2.6 Eligiendo Estructuras y Tipos

La teoría de los tipos y estructuras de datos que ofrece Python es importante, pero hay que ponerla en práctica. Veamos algunas directrices para que sepas cuál de todas usar en función de tu caso:

- **Mutabilidad:** ¿Quieres un tipo que sea mutable (pueda cambiar)? ¿O qué sea inmutable (no pueda cambiar)?
- **Duplicados:** ¿Quieres almacenar valores duplicados?
- **Tipos:** ¿Quieres almacenar elementos del mismo tipo o diferentes?
- **Velocidad:** ¿Trabajas con muchos datos y necesitas velocidad?
- **Inserción:** ¿Quieres insertar elementos al principio, al final o por el medio?
- **Orden:** ¿Te importa el orden de los elementos?
- **Dependencias:** ¿Estás dispuesto a usar una dependencia externa?

Visto esto tenemos un problema. No podemos tener todo a la vez. Cada elección que hacemos supone un coste de oportunidad a algo que renunciamos. Veamos unas directrices para saber qué tipo o estructura elegir.

### 2.6.1 Lista vs Set

La `list` mantiene el orden de inserción, el `set` no.

```
print([3, 1, 4]) # [3, 1, 4]
print({3, 1, 4}) # {1, 3, 4}
```

La `list` permite duplicados, el `set` no.

```
print([1, 1, 1]) # [1, 1, 1]
print({1, 1, 1}) # {1}
```

La `list` permite ser indexada, el `set` no.

```
l = [1, 2, 3]
s = {1, 2, 3}

print(l[0]) # 1
print(s[0]) # TypeError: 'set' object is not
             subscriptable
```

Si queremos ver si un elemento está contenido en nuestra estructura.

Por ejemplo, ver si `4` esta en la secuencia `0, 1, 2, 3, 4, 5` :

- 🕹 Usando `list` es mas lento.
- 🚦 Usando `set` es mas rápido.

En el siguiente ejemplo apenas se nota la diferencia, pero cuando la secuencia tiene millones de elementos, importan. Y mucho.

```
l = [0, 1, 2, 3, 4, 5]
s = {0, 1, 2, 3, 4, 5}

# Lento
if 4 in l:
    print("Esta")

# Rápido
if 4 in s:
```

```
print("Esta")
```

## 2.6.2 Lista vs Diccionario

Los `dict` permiten almacenar contenido en forma de *key-value*, las `list` no. Esto es útil cuando queremos asignar nombres a nuestros datos.

```
persona = {"nombre": "Miguel", "edad": 20}
```

Podríamos almacenar lo mismo en una `list` pero nos faltaría el nombre del campo.

```
persona = ["Miguel", 20]
```

Los `dict` permiten buscar el *value* de un *key* muy rápidamente. Formalmente se realiza con complejidad  $O(1)$ , lo que significa que no importa lo grande que sea el `dict`, encontraremos lo que buscamos en el mismo tiempo.

Las `list` son más lentas para buscar. Si queremos saber si nuestra lista contiene un determinado elemento, tendremos que recorrerla entera. Cuanto más grande, más tiempo tardaremos.

## 2.6.3 Lista vs Tupla

Las `list` son mutables pero las `tuple` no. Por lo tanto:

- 🛡️ Si quieres asegurarte de que nadie modifica el contenido a lo largo de tu programa, usa `tuple`. Por ejemplo, unas constantes.
- ✎️ Si quieres poder modificar el contenido a lo largo de tu programa, usa `list`.

---

```
l = [1, 2] # Si loquieres modificar
t = (1, 2) # Si no quieres que se modifique
```

## 2.6.4 Diccionario vs pd.DataFrame

Con `pd.Series` podemos almacenar algo parecido a un `dict`.

```
import pandas as pd
x = pd.Series({'nombre': 'Miguel'})
print(x['nombre'])
# Miguel
```

Y con `DataFrame` también.

```
df = pd.DataFrame({
    'Temperatura': [73, 75, 72, 68],
    'Humedad': [55, 60, 65, 70]
})
```

Sin embargo, los tipos de `pandas` ofrecen muchas más funciones ya están pensados para trabajar con grandes conjuntos de datos. Si manejas pocos datos, usa `dict`. Si quieres procesar millones de datos y hacer operaciones con ellos, usa `pandas`.

## 2.6.5 Lista vs np.array

Las `list` y `np.array` pueden parecer similares. Ambas permiten almacenar un conjunto de datos.

```
# Lista
x1 = [1, 2, 3]

# np.array
x2 = np.array([1, 2, 3])
```

---

Sin embargo `np.array` viene con muchas operaciones bastante optimizadas. Como puedes ver calcular una media sobre una `list` es mas lento que sobre un `np.array`.

```
from timeit import timeit
import numpy as np

x1 = [i for i in range(1000000)]
x2 = np.array(x1)

print("lista:", timeit('sum(x1)/len(x1)', number=100,
                      globals=globals()))
print("np.array:", timeit('x2.mean()', number=100,
                         globals=globals()))

# lista: 0.4844
# np.array: 0.031
```

Por lo tanto, si quieres almacenar pocos datos, usa `list`. Si quieres almacenar muchos datos, realizar operaciones con matrices o usar funciones estadísticas, usa `np.array`.

## 2.6.6 `np.array` vs `pd.Series`

Los `np.array` y `pd.Series` son muy parecidos. De hecho el segundo está basado en el primero, pero con funcionalidades añadidas.

Por norma general, si buscas velocidad, usa `np.array`. Puedes ver como `np.array` es 3 veces más rápido que `pd.Series`.

```
import numpy as np
import pandas as pd
from timeit import timeit

datos_ndarray = np.random.rand(1000000)
datos_series = pd.Series(datos_ndarray)
```

```
print("Tiempo numpy:", timeit('datos_ndarray.mean()',  
    number=1000, globals=globals()))  
print("Tiempo pandas:", timeit('datos_series.mean()',  
    number=1000, globals=globals()))  
  
# Tiempo numpy: 1.9  
# Tiempo pandas: 5.4
```

Sin embargo si necesitas procesar los datos, agruparlos, o realizar operaciones, `pd.Series` ofrece más opciones.

## 2.6.7 Clase vs NamedTuple

La `class` y `namedtuple` son muy similares. Podemos implementar la misma idea con ambas, como una estructura de datos que almacena un punto en dos dimensiones `x` e `y`.

Con `namedtuple`.

```
from collections import namedtuple  
Punto = namedtuple('Punto', ['x', 'y'])  
p = Punto(10, 20)  
  
print(f'Coordenadas: x={p.x}, y={p.y}')  
# Coordenadas: x=10, y=20
```

Lo mismo con una `class`.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
p = Punto(10, 20)  
print(f'Coordenadas: x={p.x}, y={p.y}')  
# Coordenadas: x=10, y=20
```

Recomendamos lo siguiente:

- 
- 📱 La `namedtuple` es inmutable. No podrás modificar los valores.
  - ✎ Si quieres modificar `x` e `y` después de crear el objeto, usa una `class` ya que si es mutable.
  - 🚗 La `namedtuple` puede ser más eficiente al ser más sencilla.
  - ✈️ Si necesitas crear métodos que realicen operaciones sobre los valores, usa `class`.

En la sección de programación orientada a objetos veremos como sacarle el máximo partido a la `class` adentrándonos en el mundo de la programación orientada a objetos.

---

### 3 Control con Bucles y Condicionales



Un código es un conjunto de instrucciones que indican las tareas a realizar. Pero dependiendo de diferentes condiciones, el código puede seguir caminos distintos. En Python podemos controlar el flujo de ejecución de nuestro código con:

- **Condicionales:** Permiten ejecutar una sección de código si se cumple una determinada condición. Tenemos el `if` , `elif` , `else` y `match` .
- **Bucles:** Permiten ejecutar una sección de código múltiples veces. Tenemos el `for` y `while` .

Veamos cómo controlar el flujo de ejecución de tu programa con bucles y condicionales. Estas herramientas te permitirán elegir el camino que sigue tu código.

---

### 3.1 Condicionales con if/else

El uso de `if` nos permite ejecutar una sección de código si se cumple una condición determinada. Por ejemplo, si la `edad` es mayor o igual que `10`, se ejecuta el código que ves.

```
edad = 18
if edad >= 18:
    print("Mayor de edad")
# Mayor de edad
```

También podemos combinar condiciones usando:

- `and` : Devuelve `True` si ambas condiciones son `True` .
- `or` : Devuelve `True` si al menos una condición es `True` .

Usando `and` verificamos que la `edad` sea mayor que 18 y menor que 30.

```
edad = 20
if edad > 18 and edad < 30:
    print("Es joven")
# Es joven
```

Aunque es más *Pythonic* hacerlo de la siguiente manera.

```
edad = 20
if 18 < edad < 30:
    print("Es joven")
# Es joven
```

Usando `or` consideraremos que la edad es incorrecta si es negativa, es decir, menor que cero, o mayor que 150, al no haber nadie tan mayor.

```
edad = -5
if edad < 0 or edad > 150:
    print("Edad incorrecta")
```

```
# Edad incorrecta
```

Podemos expresar lo mismo usando `not`, que permite invertir el `True` por `False` y viceversa.

```
if not (0 <= edad <= 150):
    print("Edad incorrecta")
```

Existen cosas interesantes que nos encontramos en Python, como que cuando evaluamos una lista vacía `[]`, se considera `False`.

```
x = []
if not x:
    print("La lista está vacía")
# La lista está vacía
```

Se podría también hacer de la siguiente manera, pero no es muy *Pythonic*.

```
x = []
if len(x) == 0:
    print("La lista está vacía")
# La lista está vacía
```

Por otro lado, cualquier número diferente de cero se considera `True`.

```
if 10:
    print("Es True")
# Es True
```

Y `None` se considera `False`.

```
x = None
if not x:
    print("El valor es None")
# El valor es None
```

Y mucho cuidado, porque aunque `[]`, `10` y `None` sean evaluados

---

como `False` , `True` y `False` respectivamente, eso no significa que sea comparable con `==` .

```
print([] == False)      # False
print(10 == True)       # False
print(None == False)    # False
```

También podemos añadir un código a ejecutar si la condición no se cumple, usando `else` .

```
edad = -5
if 0 >= edad >= 150:
    print("Edad correcta")
else:
    print("Edad incorrecta")
# Edad incorrecta
```

Y si queremos tener varias condiciones, podemos añadir tantas cuanto queramos con `elif` . Esto nos permite definir múltiples comportamientos según la condición que se dé.

```
hora = 5
if 5 <= hora < 12:
    print("Mañana")
elif 12 <= hora < 18:\n    print("Tarde")
elif 18 <= hora < 21:
    print("Noche")
else:
    print("Madrugada")
# Mañana
```

Un truco interesante es definir nuestro `if` en una única línea de código. Aunque de acuerdo a la E701 no es recomendable.

```
a = 10
if a > 5: print("Es > 5")
# Es > 5
```

---

También tenemos el operador ternario, que permite definir en una única línea la condición con el código a ejecutar en ambos casos. Es simplemente una estructura `if` `else` pero expresada en una sola línea.

```
x = 3
print("Es 5" if x == 5 else "No es 5")
# No es 5
```

Otro ejemplo similar. Si queremos dividir `a/b`, pero nos queremos asegurar que `b` no es cero, ya que dividir por cero no puede realizarse, podemos usar el operador ternario de la siguiente forma.

```
a = 10
b = 0
c = a/b if b != 0 else "Error"
print(c)
# Error
```

Por último, el operador *walrus* a veces nos puede venir bien con el `if`. En la misma línea permite realizar la asignación y la comparación.

```
a = [1, 2, 3]
if (n := len(a)) < 10:
    print(f"Tiene pocos elementos: {n}")
# Tiene pocos elementos: 3
```

Si no usar el operador *walrus* el código anterior se podría escribir así. Muy parecido pero con una línea más.

```
a = [1, 2, 3]
n = len(a)
if n < 10:
    print(f"Tiene pocos elementos: {n}")
# Tiene pocos elementos: 3
```

## 3.2 Switch usando Match

Python ofrece algo similar al *switch* de otros lenguajes de programación desde la versión 3.10.

Se trata del `match`. Cada `case` define un camino posible. El `_` es la opción por defecto, que se ejecuta si la entrada no coincide con ningún caso.

```
hora = 8
match hora:
    case 8:
        print("Desayuno")
    case 14:
        print("Comida")
    case 21:
        print("Cena")
    case _:
        print("No toca comer")
# Desayuno
```

El `match` nos permite realizar lo mismo que con múltiples `if/elif` como hemos visto anteriormente. Ambos códigos son equivalentes.

```
if hora == 8:
    print("Desayuno")
elif hora == 14:
    print("Comida")
elif hora == 21:
    print("Cena")
else:
    print("No toca comer")
```

También podemos tener en nuestros `case` múltiples condiciones, donde `|` es interpretado como un `or`.

```
mes = 4
match mes:
```

```
case 12 | 1 | 2: print("Invierno")
case 3 | 4 | 5: print("Primavera")
case 6 | 7 | 8: print("Verano")
case 9 | 10 | 11: print("Otoño")
case _: print("Error")
# Primavera
```

Aunque no acaba ahí la cosa. Podemos hacer *matching* de prácticamente cualquier cosa como una tupla.

```
coordenada = (0, 1)
match coordenada:
    case (0, 0):
        print("Coordenada en origen")
    case (x, 0):
        print(f"Coordenada en eje x: {x}")
    case (0, y):
        print(f"Coordenada en eje y: {y}")
    case (x, y):
        print(f"Coordenada en: {x}, {y}")
    case _:
        print("Error")

# Coordenada en eje y: 1
```

### 3.3 Iterando Con Bucles For

Los bucles `for` permiten ejecutar una sección de código un número determinado de veces. El siguiente ejemplo se ejecuta 3 veces y muestra los números `0`, `1` y `2`.

```
for i in range(3):
    print(i)
# 0, 1, 2
```

Los bucles también permiten iterar una lista de la siguiente manera.

```
l = [0, 1, 2]
```

```
for i in l:  
    print(i)
```

El uso de `for` e `in` es bastante interesante porque nos permite iterar cualquier clase iterable. Por ejemplo, una cadena es un iterable.

```
for i in "Python":  
    print(i)  
# P  
# y  
# t  
# h  
# o  
# n
```

O una lista con `str`.

```
colores = ["rojo", "verde", "azul"]  
for color in colores:  
    print(color)  
# rojo  
# verde  
# azul
```

Si vienes de otros lenguajes de programación, tal vez pienses en lo siguiente. Aunque el ejemplo es válido, esto no es una forma *Pythonic* de hacerlo.

```
colores = ["rojo", "verde", "azul"]  
for i in range(len(colores)):  
    print(colores[i])  
# rojo  
# verde  
# azul
```

Los bucles también pueden ser anidados. Es decir, un bucle dentro de otro.

```
matriz = [[1, 2, 3],
```

```
[4, 5, 6],  
[7, 8, 9]]  
for fila in matriz:  
    for columna in fila:  
        print(columna)  
# 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Y hay diferentes herramientas que nos permiten por ejemplo iterar al revés.

```
texto = "Python"  
for i in reversed(texto):  
    print(i) #n,o,h,t,y,P
```

También podemos iterar diccionarios.

```
d = {"nombre": "Juan", "apellido": "Prieto"}  
  
for clave, valor in d.items():  
    print(f"clave: {clave}. valor: {valor}")  
# clave: nombre. valor: Juan  
# clave: apellido. valor: Prieto
```

O podemos iterar dos `list` a la vez con `zip`.

```
l1 = [1, 2, 3]  
l2 = ['a', 'b', 'c']  
  
for i1, i2 in zip(l1, l2):  
    print(i1, i2)  
  
# 1 a  
# 2 b  
# 3 c
```

Si quieres acceder al índice de cada elemento, puedes usar `enumerate`.

```
colores = ["rojo", "verde", "azul"]  
for indice, color in enumerate(colores):
```

```
print(f"{{indice}}={{color}}")
# 0=rojo
# 1=verde
# 2=azul
```

Usando `break` puedes romper el bucle cuando lo deseas. Esto suele ser útil cuando buscamos un elemento en una lista. Una vez encontrado lo que buscamos, podemos terminar con `break`.

A continuación buscamos el `3`. Una vez encontrado, paramos con `break`, ya que no tiene sentido seguir buscando.

```
lista = [1, 2, 3, 4, 5]
busca = 3

for l in lista:
    if l == busca:
        print(f"Encontrado {busca}")
        break

# Encontrado 3
```

Por otro lado podemos usar `continue` para saltar a la siguiente iteración. Es decir, todo lo que quede debajo del `continue` no se ejecutará, pasando a la siguiente iteración.

A diferencia del `break`, el `continue` no para el bucle del todo, sino que salta a la siguiente iteración.

En el siguiente ejemplo saltamos los números pares, es decir, cuyo módulo 2 `\%2` es `==0` e imprimimos los impares.

```
for i in range(8):
    if i % 2 == 0:
        continue
    print(i)

# 1, 3, 5, 7
```

---

También puedes terminar el `for` de manera prematura de otra manera. Si estamos dentro de una función y usamos `return` retornaremos de la función terminando el bucle.

```
def encuentra(lista, busca):
    for l in lista:
        if l == busca:
            return busca

print(encuentra([1, 2, 3, 4, 5], 3))
# 3
```

Otra característica notable del `for` en Python, es la posibilidad de usarlo junto con `else`. Es de hecho algo poco común en otros lenguajes.

El contenido del `else` se ejecuta cuando el bucle termine de manera normal, es decir, sin haber encontrado un `break` en su camino.

```
def encuentra(lista, busca):
    for l in lista:
        if l == busca:
            print(f"{busca} se encontró")
            break
    else:
        print(f"{busca} NO se encontró")
```

Puedes verlo en acción así.

```
n = [1, 2, 3, 4, 5]
encuentra(n, 3)
encuentra(n, 7)

# 3 se encontró
# 7 NO se encontró
```

---

### 3.4 Iterando Con List Comprehension

Las *list comprehension* permiten convertir esto.

```
lista = []
for i in range(10):
    lista.append((i))
print(lista)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

En esto. El comportamiento es el mismo, pero nos ahorraremos un par de líneas de código. Estas son una herramienta muy potente de Python.

```
print([i for i in range(10)])
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

También podemos añadir condiciones. En este caso filtramos solo los números pares. Para ello usamos un `if` con la condición.

```
print([i for i in range(10) if i % 2 == 0])
# [0, 2, 4, 6, 8]
```

Y también podemos hacer operaciones en cada elemento. En este caso elevamos al cuadrado cada número par. Observa el `i**2`.

```
print([i**2 for i in range(10) if i % 2 == 0])
# [0, 4, 16, 36, 64]
```

Si queremos realizar operaciones más complejas, podemos aplicar una función sobre cada elemento.

```
def cuadrado(a):
    return a**2

print([cuadrado(i) for i in range(10) if i % 2 == 0])
# [0, 4, 16, 36, 64]
```

---

También podemos concatenar varias. Suele ser común para pasar de una lista de dos dimensiones a una, conocido como *flatten*.

```
l = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]  
print([elemento for sublist in l for elemento in  
      sublist])  
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Y podemos complicar las cosas tanto como queramos. Por ejemplo, podemos calcular los números primos hasta el 50 de la siguiente manera.

```
p = [x for x in range(2, 50) if all(x % i != 0 for i  
      in range(2, int(x**0.5) + 1))]  
print(p)  
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,  
  47]
```

A veces es mejor tener dividir la *list comprehension* en varias líneas para que resulte más fácil de leer. Mostramos el mismo ejemplo pero más legible.

```
p = [  
    x  
    for x in range(2, 50)  
    if all(  
        x % i != 0  
        for i in range(2, int(x**0.5) + 1)  
    )  
]
```

Las *list comprehensions* son útiles pero no abuses de ellas. A veces pueden hacer que el código sea más difícil de leer. Si ves que complican mucho las cosas, usa bucles “normales”.

Y como apunte final. Hay dos formas de escribir *list comprehension*:

- Con `[]`. Lo que hemos visto hasta ahora. El resultado es una lista.

- 
- Con `()`. El resultado es un generador. Lo veremos en detalle más adelante.

No es lo mismo usar `[]`.

```
print([i for i in range(10)])
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Que usar `()`.

```
print((i for i in range(10)))
# <generator object <genexpr> at 0x104b268c0>
```

Aunque siempre puedes convertir el segundo al primero con `list()`.

```
print(list(i for i in range(10)))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 3.5 Iterando Con Bucles While

A diferencia de los bucles `for` los bucles `while` no iteran sobre un iterable (como una lista) de principio a fin. Se ejecutan continuamente mientras una condición sea cierta.

Por ejemplo, mientras que `i` sea menor que `5`. Como en cada iteración añadimos `1` llega un momento en que la condición ya no es cierta y se para.

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Si no modificas `i` como en este ejemplo, crearás un bucle infinito. Esto sucede porque `i` es siempre `<5`. Por lo tanto el bucle no para nunca.

```
i = 0
while i < 5:
    print(i)
```

También puedes usar el `break` dentro de los bucles `while`. Como puedes ver, el `break` rompe el bucle. Es decir, cuando `i>5` se rompe. Pero es menos legible.

```
i = 0
while True:
    if i >= 5:
        break
    print(i)
    i += 1
```

Usar `while True` crea un bucle infinito si no hay ningún `break` que lo rompa. Por ejemplo, esto genera un bucle infinito.

```
import time

i = 0
while True:
    print(f"Iteración {i}")
    i += 1
    time.sleep(1)
```

Usando `while` también podemos resolver los mismos problemas que con `for`. Como iterar una lista e imprimir sus valores. Sin embargo, esto no es muy *Pythonic*. Funciona, pero no hagas esto.

```
colores = ["rojo", "verde", "azul"]
i = 0
while i < len(colores):
    print(colores[i])
    i += 1
```

Por último el `while` también permite usar `else`. Esta sección se ejecutará si el bucle termina de manera normal. Esto es, sin rom-

---

perte con `break` prematuramente. Un caso práctico de esto es lo siguiente.

Imagina que tienes una tarea que a veces falla. Puedes implementar un sistema que intenta de nuevo hasta `intentos`. Si después de un número máximo de intentos no ha salido bien, se da error.

```
import random

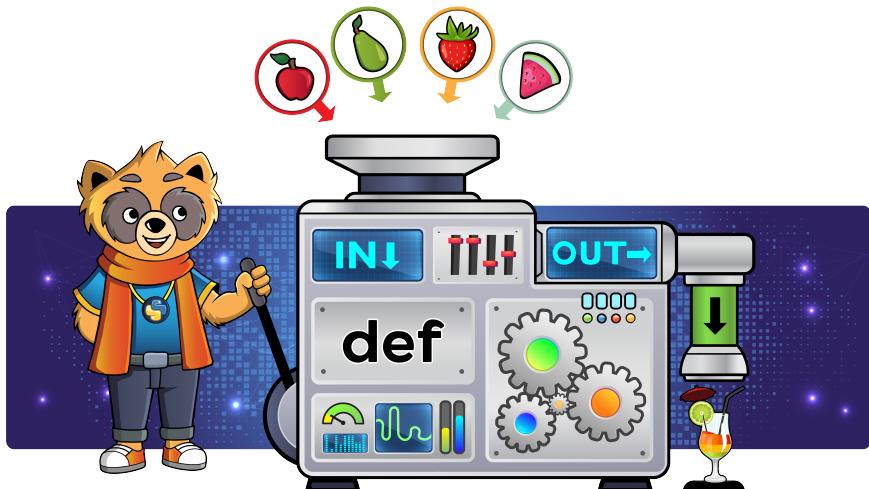
intentos = 5
i = 0

while i < intentos:
    print(f"Intento {i + 1}")
    ok = random.choice([True, False])

    if ok:
        print("Completado")
        break
    else:
        print("Error. Intenta otra vez")
        i += 1
else:
    print("Todos los intentos fallaron")
```

---

## 4 Reutiliza Código con Funciones



Las funciones nos permiten agrupar y reutilizar código. También abstractan la complejidad para quien la usa. No importa lo que haya en su interior, simplemente quieres saber lo que entra y lo que sale. Toda función tiene:

- **Argumentos de entrada:** Lo que entra a la función.
- **Argumentos de salida:** Lo que sale de la función.
- **Código:** Lo que define el comportamiento, indicando cómo las entradas se transforman en las salidas.

A continuación vemos cómo definir funciones con `def`, la diferencia entre paso por valor y referencia, el uso de decoradores con `@`, generadores con `yield`, funciones asíncronas con `async` y funciones lambda con `lambda`.

## 4.1 Introducción a Funciones

Las funciones son un conjunto de instrucciones agrupadas bajo un nombre. Esto es una función. Se definen con `def`. Tiene un nombre, unos parámetros de entrada y una salida.

```
def suma(a, b):
    resultado = a + b
    return resultado
```

Existen otras variantes como funciones que no devuelven ningún argumento o funciones que no aceptan ningún argumento de entrada. Gracias a las funciones podemos:

-  **Reutilizar:** Eliminan código repetitivo. En vez de escribir las mismas 10 líneas una y otra vez, las puedes poner en una función y llamarla cuando la necesites en 1 línea.
-  **Organizar:** Permiten agrupar código relacionado. Hacen que el código sea más fácil de entender. En vez de tener un código enorme, es mejor dividirlo en funciones con una funcionalidad específica.
-  **Abstraer:** Permiten abstraer la complejidad de su interior. Una función puede ser muy compleja internamente, pero a quién la use le basta con saber las entradas que acepta y que salidas devuelve. Es el conocido enfoque de *black box* o caja negra. No importa lo que hay dentro, solo importa saber lo que entra y lo que sale.

Veamos una función que convierte grados Celsius a Fahrenheit.

```
def celsius_a_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit
```

La función puede ser llamada con `()`, pasando dentro los argu-

---

mentos necesarios y capturando lo que devuelve en una variable `temperatura_f`.

```
temperatura_c = 25
temperatura_f = celsius_a_fahrenheit(temperatura_c)
print(temperatura_f)
# 77.0
```

Podemos también simplificar la función a una línea.

```
def celsius_a_fahrenheit(celsius):
    return (celsius * 9/5) + 32
```

Dado que Python no tiene tipado estático ni es compilado, si llamamos a nuestra función con un `str` obtendremos el siguiente error. Esto es debido a que la operación división no está definida para `str` e `int`. Tiene sentido, no puedes dividir `texto` entre `5`.

```
print(celsius_a_fahrenheit("texto"))
# TypeError: unsupported operand type(s)
```

Un apunte importante es que no es lo mismo usar la función con y sin `()`.

- Con `()`: Se llama a la función con unos argumentos de entrada. Se nos devuelve los argumentos de salida de la función.
- Sin `()`: Se accede a la función en sí, que es un objeto de la clase `function`.

```
# Con ()
print(celsius_a_fahrenheit(30))
# 86.0

# Sin ()
print(celsius_a_fahrenheit)
# <function celsius_a_fahrenheit at 0x1009341f0>
```

Las funciones también aceptan varios parámetros de entrada.

```
import math

def hipotenusa(a, b):
    return math.sqrt(a**2 + b**2)

print(hipotenusa(3, 4))
# 5.0
```

Podemos llamar a la función indicando el nombre del argumento, lo que es equivalente.

```
print(hipotenusa(a=3, b=4))
# 5.0
```

Como los argumentos están nombrados, podemos también cambiar el orden.

```
print(hipotenusa(b=3, a=4))
```

Como es de esperar, si pasamos el nombre de un argumento que no existe, tendremos un error.

```
print(hipotenusa(c=10))
# TypeError: hipotenusa()
```

Los argumentos de una función también pueden tener un valor por defecto. Este será usado si no se pasa ninguno. En este caso si `b` no se proporciona, este tomará `1` como valor por defecto.

```
def multiplica(a, b=1):
    return a * b

print(multiplica(10))
# 10

print(multiplica(10, 2))
# 20
```

---

Es importante que los argumentos con valor por defecto vayan después, de lo contrario obtendremos un error. Esta definición no es correcta.

```
def multiplica(a=1, b):  
    return a * b  
  
# SyntaxError
```

Por otro lado, las funciones pueden tener un número variable de argumentos, usando `*args`.

```
def suma(*args):  
    return sum(args)
```

Esta función `suma` puede ser llamada de la siguiente manera.

```
print(suma())  
# 0  
  
print(suma(100, 200))  
# 300  
  
print(suma(5, 3, 2, 2))  
# 12
```

Y si quieres nombrar cada argumento, lo puedes hacer así.

```
def suma(**kwargs):  
    suma = 0;  
    for key, value in kwargs.items():  
        print(key, value)  
        suma += value  
    return suma  
  
print(suma(a=5, b=20, c=23)) # 48  
print(suma(a=1, b=2)) # 3
```

De igual manera, podemos pasar un diccionario como parámetro de

---

entrada. Es equivalente a lo anterior.

```
d = {'a': 5, 'b': 20, 'c':23}
print(suma(**d)) # 48
```

También podemos tener funciones que acepten un número de parámetros fijos (`arg1` y `arg2`) y otros variables (`args`). La siguiente función permite entre `2` y `n` argumentos.

```
def argumentos_variables(arg1, arg2, *args):
    print(f"Argumentos fijos: {arg1}, {arg2}")
    print(f"Argumentos variables: {args}")
```

Con `0` argumentos nos da un error, ya que `arg1` y `arg2` son obligatorios.

```
argumentos_variables()
# TypeError:
```

Usando `2` argumentos.

```
argumentos_variables(1, 2)
# Argumentos fijos: 1, 2
# Argumentos variables: ()
```

Usando `4` argumentos.

```
argumentos_variables(1, 2, 3, 4)
# Argumentos fijos: 1, 2
# Argumentos variables: (3, 4)
```

Pero no acaba ahí la cosa. También podemos tener un número variable de argumentos a los que les podemos dar un nombre. Esta función admite lo siguiente:

- Dos argumentos fijos: `arg1` y `arg2`. Esto siempre tienen que estar presentes.
- Varios variables: `*args`. Podemos pasar los que queramos.

- 
- Y varios variables con nombre: `**kwargs` . Podemos también pasar los que queramos, pero hay que nombrarlos.

```
def argumentos_variables(arg1, arg2, *args, **kwargs):  
    print(f"Argumentos fijos: {arg1}, {arg2}")  
    print(f"Argumentos variables: {args}")  
    print(f"Argumento clave/valor: {kwargs}")
```

Veamos un ejemplo de cómo usarla.

```
argumentos_variables(1, 2, 3, 4, otro_arg=10, mas_arg  
=20)  
# Argumentos fijos: 1, 2  
# Argumentos variables: (3, 4)  
# Argumento clave/valor: {'otro_arg': 10, 'mas_arg':  
20}
```

Una vez vistos los argumentos de entrada, veamos los de salida. Una función puede tener múltiples argumentos de salida. Esta función calcula la media y varianza de unos datos.

```
def media_y_varianza(datos):  
    media = sum(datos) / len(datos)  
    varianza = sum((x - media) ** 2 for x in datos) /  
        len(datos)  
  
    return media, varianza  
  
print(media_y_varianza([10, 20, 30, 40]))  
# (25.0, 125.0)
```

Como puedes ver se devuelve una `tuple` . Si quieres asignar los valores a distintas variables, puedes hacer lo siguiente.

```
media, varianza = media_y_varianza([10, 20, 30, 40])  
print(media) # 25.0  
print(varianza) # 125.0
```

Imagina que no quieres la varianza. Puedes ignorarlo usando `_` .

```
media, _ = media_y_varianza([10, 20, 30, 40])
```

Por otro lado, una función sin un `return` implícito, devolverá automáticamente `None`. La siguiente función no devuelve nada, pero Python devuelve automáticamente `None`.

```
def mi_funcion():
    pass

salida = mi_funcion()
print(salida)
# None
```

Es importante ser consciente de esto porque puede llevar a horas perdidas buscando de donde viene el `None`. En algunos casos, puede ser más difícil darse cuenta.

```
def hora_a_comida(hora):
    match hora:
        case 8:
            return "Desayuno"
        case 14:
            return "Comida"
        case 21:
            return "Cena"

print(hora_a_comida(7))
# None
```

Por lo tanto, es mejor ser explícito y tener en cuenta el `None`. El resultado es el mismo, pero es más explícito.

```
def hora_a_comida(hora):
    match hora:
        case 8:
            return "Desayuno"
        case 14:
            return "Comida"
        case 21:
```

```
        return "Cena"
    case _:
        return None

print(hora_a_comida(7))
# None
```

Las funciones también pueden ser asignadas a variables. En este caso `di_hola` actúa como un alias de `saluda`.

```
def saluda(nombre):
    return f"Hola, {nombre}!"

di_hola = saluda
```

Y podemos llamarla.

```
print(di_hola("Juan"))
# Hola, Juan!
```

Las funciones también pueden ser almacenadas en una lista. En este caso almacenamos `suma` y `resta` en `operaciones`.

```
def suma(x, y):
    return x + y

def resta(x, y):
    return x - y

operaciones = [suma, resta]
print(operaciones[0](10, 5)) # 15
print(operaciones[1](10, 5)) # 5
```

Una función también puede devolver otra función almacenando un contexto, en este caso el `factor`. Esto es conocido como *closure*, y es algo así como una función con otra función dentro.

```
def crear_multiplicador(factor):
    def multiplica(x):
```

```
        return x * factor
    return multiplica

multiplicar_por_2 = crear_multiplicador(2)
multiplicar_por_3 = crear_multiplicador(3)

print(multiplicar_por_2(5))  # 10
print(multiplicar_por_3(5))  # 15
```

Una función también puede ser un argumento de entrada a otra. Dentro de la función podemos llamar a la otra. Puedes ver como `ejecuta` llama a `funcion`.

```
def ejecuta(funcion, a, b):
    return funcion(a, b)

def suma(a, b):
    return a + b

print(ejecuta(suma, 3, 4))
# 7
```

Como puedes ver se puede hacer de todo con las funciones. Esto es gracias a que Python trata a las funciones como objetos de la clase `function`.

```
def suma(a, b):
    return a + b

print(type(suma))  # <class 'function'>
```

Y como último apunte. Es una buena práctica que tus funciones no tengan *side effects*. Es decir, que no modifiquen contenido externo. Este ejemplo no es lo ideal. La función modifica una variable externa.

```
i = 0

def masuno():
    global i
```

```
i += 1
return i

print(masuno()) # 1
print(masuno()) # 2
print(masuno()) # 3
```

## 4.2 Paso por valor y referencia

En muchos lenguajes de programación existen dos formas de pasar los argumentos de entrada a una función:

- **Por valor:** Se crea una copia local de la variable. Cualquier modificación sobre la misma no tendrá efecto sobre la original.
- **Por referencia:** Se actúa directamente sobre la variable de entrada. Las modificaciones afectarán a la variable original.

Veamos un ejemplo de paso por valor. Como puedes ver `x` no cambia. La `funcion` actúa sobre una copia y no modifica el valor original.

```
x = 10
def funcion(entrada):
    entrada += 1

funcion(x)
print(x)
# 10 <- La x NO cambia
```

Ahora veamos un ejemplo de paso por referencia. Como puedes ver `x` cambia. La `funcion` actúa directamente sobre `x`.

```
x = [10]
def funcion(entrada):
    entrada.append(20)

funcion(x)
```

```
print(x)
# [10, 20] <- La x SI cambia
```

Esta diferencia puede resultar algo complicada de entender en Python, ya que no se puede indicar de manera explícita si queremos que sea tratado por valor o por referencia como en otros lenguajes.

La diferencia clave es que el primer ejemplo usa un `int` y el segundo un `list`. El comportamiento está definido por el tipo de variable usada:

- 📋 Los tipos **inmutables** como `int` o `string` se pasan como valor. La función no puede modificar la variable original.
- 🔗 Los tipos **mutables** como `list` se pasan como referencia. La función modifica el valor original.

Ante la duda, podemos usar `id()` para saber si dos variables hacen referencia a la misma. Podemos ver como el `id()` es distinto.

📋 Paso por valor. El `id` es diferente. Son variables diferentes. Modificar una no modifica la otra.

```
x = 10
def funcion(entrada):
    entrada += 1
    print(id(entrada)) # 4341262928

print(id(x)) # 4341262960
funcion(x)
```

🔗 Paso por referencia. El `id` es el mismo. Se trata de la misma variable. Si modificas una, la otra también se modifica.

```
x = [10]
def funcion(entrada):
    entrada.append(20)
    print(id(entrada)) # 4299629568
```

```
print(id(x)) # 4299629568
funcion(x)
```

Puedes ver por tanto como en Python el comportamiento de paso por valor o por referencia está directamente relacionado con los tipos de datos mutables e inmutables. Como apunte final, veamos algunas ventajas e inconvenientes de estos tipos:

-  Inmutables: Los datos están protegidos de modificaciones, lo que lo hace más seguro. Sin embargo puede requerir más memoria, ya que se están continuamente copiando.
-  Mutables: Los datos originales pueden ser modificados. Sin embargo requieren menos memoria, ya que siempre se usa la variable original. Es muy útil cuando trabajamos con tipos que almacenan muchos datos, donde no sería viable copiar continuamente.

## 4.3 Documentando Funciones

El código se lee más veces de las que se escribe. Es importante documentarlo bien para que otra gente lo pueda entender. Aunque suene evidente, la realidad es que documentar es algo que se hace poco y mal.

Para documentar funciones es importante aplicar el enfoque de caja negra. No importa lo que hay dentro, sino lo que entra y lo que sale.

Programas sencillos se pueden documentar de la siguiente manera. No es la mejor forma, pero cumple el objetivo. Encontrarás millones de funciones documentadas así. Es lo más sencillo.

```
def suma(a, b):
    # Toma dos números y devuelve su suma
    return a + b
```

---

Sin embargo Python nos ofrece los *docstrings* (PEP257). Una forma más correcta de documentar. Esto es un *docstring* en una función. Se define usando `"""`. Pueden ser usados también en clases, módulos o métodos.

```
def suma(a, b):
    """Toma dos números y devuelve su suma"""
    return a + b
```

A diferencia de los comentarios con `##` los *docstrings* son accesibles en tiempo de ejecución usando `__doc__`.

```
print(suma.__doc__)
# Toma dos números y devuelve su suma
```

Y de manera similar con `help()`.

```
help(suma)
# Help on function suma in module __main__:
#
# suma(a, b)
#     Toma dos números y devuelve su suma
```

Aunque el comentario anterior es perfectamente válido, si creamos funciones más complejas o desarrollamos librerías *open source*, es posible que debamos incluir más información.

Existen diferentes formas. Google recomienda hacerlo así. Se incluye una descripción de la función, los argumentos de entrada con su tipo y los argumentos de salida.

```
def suma(a, b):
    """
    Devuelve la suma de dos números.

    Args:
        a (int, float): El primer número a sumar.
        b (int, float): El segundo número a sumar.
```

```
    Returns:  
        int, float: La suma de a y b.  
    """  
    return a + b
```

Otra forma es de acuerdo al formato *reStructuredText*. Este nos permite generar documentación automáticamente con herramientas como `sphinx`. Tener la documentación junto al código permite mantenerla actualizada más fácilmente.

```
def suma(a, b):  
    """  
        Devuelve la suma de dos números.  
  
        :param a: El primer número a sumar.  
        :type a: int, float  
        :param b: El segundo número a sumar.  
        :type b: int, float  
        :return: La suma de a y b.  
        :rtype: int, float  
    """  
    return a + b
```

También `numpy` tiene su propia forma de hacerlo y existen decenas de variaciones. Pero sin importar cual, todos dan una información parecida con diferente formato.

Como apunte final:

- Si contribuyes con *open source* es recomendable usar el Inglés para documentar tus funciones. Podrás llegar a más gente con tus paquetes.
- No todas las funciones necesitan comentarios de 50 líneas siguiendo un estándar. Para funciones sencillas, una línea es suficiente. Elige en función de tu caso.

---

## 4.4 Anotaciones En Funciones

Python tiene tipado dinámico, lo que significa que las variables no tienen que ser definidas con un tipo concreto antes de usarlas. El tipo se determina en tiempo de ejecución en función a su valor.

Los argumentos de una función no tienen un tipo específico, `a` y `b` pueden ser de cualquier tipo. Otros lenguajes como Go o C nos exigen determinar el tipo exacto.

```
def multiplica(a, b):
    return a * b
```

Esto hace a Python un lenguaje muy flexible aunque también peligroso. Las siguientes llamadas funcionan. Pero la segunda devuelve un resultado que tal vez no nos esperábamos.

```
print(multiplica(2, 5))
# 10

print(multiplica("2", 5))
# 22222
```

Pero lo siguiente da error, ya que multiplicar dos `str` no está definido.

```
print(multiplica("2", "2"))
# TypeError
```

Para evitar llegar a este tipo de errores en tiempo de ejecución, Python nos ofrece las *type annotations* (PEP 3107). Son una especie de metadatos asociados a los argumentos de entrada y salida de las funciones.

En otras palabras, te permite decirle a la gente que use tu código que tipo de datos deben usar como entrada.

```
# script.py
```

```
def multiplica(a: int, b: int) -> int:  
    return a * b
```

Pero no se imponen en tiempo de ejecución. Es decir, aunque digamos que `a` y `b` se espera que sean `int`, el siguiente código se puede ejecutar al igual que antes. El error se obtiene en tiempo de ejecución.

```
print(multiplica("2", "2"))  
# TypeError
```

Pero existen herramientas como `mypy` que permiten detectar este tipo de llamadas incorrectas. Desde el terminal lo podemos llamar sobre nuestro código.

```
mypy script.py
```

Y en nuestro caso nos dirá.

```
# error: Argument 1 to "multiplica" has incompatible  
#       type "str"; expected "int"  
# error: Argument 2 to "multiplica" has incompatible  
#       type "str"; expected "int"
```

También puedes acceder a estas anotaciones usando lo siguiente.

```
print(multiplica.__annotations__)  
# {'a': <class 'int'>, 'b': <class 'int'>}
```

Estas anotaciones pueden ser usadas en tiempo de ejecución. Puedes crear una función que se asegure que `a` y `b` son `int` y de error de lo contrario.

```
def multiplica(a: int, b: int) -> int:  
    annotations = multiplica.__annotations__  
    if not isinstance(a, annotations['a']):  
        raise TypeError(f"a debe ser int not {type(a)}")
```

```
if not isinstance(b, annotations['b']):
    raise TypeError(f"b debe ser int not {type(a)}")
}

return a * b

print(multiplica("2", "2"))
# TypeError: a debe ser int not <class 'str'>
```

Existe una anotación para cada tipo y las puedes mezclar con valores por defecto.

```
def elevar(b: float, e: int = 2) -> float:
    return b ** e
```

También permite especificar tipos mas complejos. Esta función toma un `dict` que usa `str` como clave y `str` como valor y devuelve una `list` de `int`.

```
from typing import List, Dict

def procesa(cfg: Dict[str, str]) -> List[int]:
    # ...
```

En resumen sobre las anotaciones:

-  **Documentación:** Facilitan la lectura al indicar claramente qué tipo de datos espera la función.
-  **Validación:** Herramientas como `mypy` pueden usar las anotaciones para comprobar que los tipos son correctos. De esta forma nos evitamos sorpresas en tiempo de ejecución, con cosas que se pueden prevenir antes.
-  **Autocompletar:** Editores como PyCharm y VSCode pueden aprovechar las anotaciones para mejorar el autocompletado y la navegación por el código.

---

Aunque verás mucho código sin anotaciones, está bien usarlas. Te recomendamos que lo hagas a partir de ahora.

## 4.5 Recursividad Con Funciones

La recursividad es un concepto en programación donde una función se llama a sí misma un número finito de veces. Esto nos permite resolver problemas que pueden ser divididos en subproblemas más pequeños, similares al problema original.

Un ejemplo es el cálculo del factorial  $n!$ . Esta operación consiste en multiplicar un número por todos los anteriores. El factorial  $4!$  es  $4*3*2*1 = 24$ . Es un problema que se puede resolver con recursividad gracias a que:

- Resolver  $4!$  es resolver  $3!$  y multiplicar por  $4$ .
- Resolver  $3!$  es resolver  $2!$  y multiplicar por  $3$ .
- Resolver  $2!$  es resolver  $1!$  y multiplicar por  $2$ .
- El factorial  $1!$  es siempre  $1$ .

Podemos expresar esto en código de la siguiente manera. Como puedes ver, la función `factorial` se llama a sí misma. En cada nueva llamada, el valor de `n` se reduce en `1`.

```
def factorial(n):  
    # recursivo  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Cualquier función recursiva tiene dos ramas bien identificadas:

- **Llamada recursiva:** La función se llama a sí misma. Es común que la nueva llamada se realice realizando alguna modificación

---

en la entrada. En el caso del factorial,  $n-1$ .

- **Retorno:** En algún momento, la función se deja de llamar a sí misma y retorna. Esta rama es importante, ya que de no existir, la función podría entrar en un bucle infinito.

Aunque la recursividad está bien, no olvides que esto también se puede escribir de manera normal. El siguiente código calcula el factorial pero sin recursividad. Un bucle de los de toda la vida.

```
def factorial(n):
    # no recursivo
    r = 1
    for i in range(2, n+1):
        r *= i
    return r
```

Otro ejemplo es calcular la serie de Fibonacci. Dicha serie calcula el elemento  $n$  sumando los dos anteriores, siendo los dos primeros el 0 y el 1. Es decir, la serie sería la siguiente. Como puedes observar, el siguiente elemento es la suma de los dos anteriores.

```
# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

De manera similar al factorial, podemos calcular el elemento  $n$  de la siguiente forma.

```
def fibonacci(n):
    # recursivo
    return n if n <= 1 else fibonacci(n-1) + fibonacci(n-2)
```

Y también lo podemos expresar de forma no recursiva así. Otra vez, con bucles de toda la vida.

```
def fibonacci(n):
    # no recursivo
    a, b = 0, 1
```

```
for _ in range(n):
    a, b = b, a + b
return a
```

Otro ejemplo interesante de recursividad es convertir una lista de `n` dimensiones en `1` dimensión, algo conocido como *flatten*.

En este caso tenemos una lista con listas dentro. El objetivo es tener una sola lista sin anidados. Este ejemplo es interesante, ya que mezcla recursividad con generadores, algo que veremos más adelante.

```
def flatten(lst):
    for item in lst:
        yield from flatten(item) if isinstance(item,
                                                list) else (item,)

nested_list = [1, [2, [3, 4], 5], 6]
print(flatten(nested_list))
# [1, 2, 3, 4, 5, 6]
```

## 4.6 Decorando Funciones

Los decoradores son funciones que nos permiten modificar el comportamiento de otras funciones. Si has visto una función precedida de `@`, eso es un decorador.

Veamos un ejemplo. Vamos a crear un decorador que muestre el tiempo que se tardó en ejecutar una función. El decorador se define con `def`, como si de una función normal se tratase.

El decorador acepta como entrada una función `func` y la devuelve modificando ligeramente su comportamiento.

```
import time

# Esto es un decorador
def mide_tiempo(func):
```

```
def wrapper(*args, **kwargs):
    t0 = time.time()
    res = func(*args, **kwargs)
    print(f"{func.__name__}: {time.time() - t0:.5f}
          } segundos")
    return res
return wrapper
```

Ahora podemos usar nuestro decorador con `@mide_tiempo` sobre cualquier función. Por ejemplo, la siguiente función determina si un número es primo o no.

El único cambio es que ahora imprime el tiempo que tardó en ejecutarse. El resto es igual.

```
# Esto es usar un decorador
@mide_tiempo
def es_primo(n):
    return n > 1 and all(n % i != 0 for i in range(2,
        int(n**0.5) + 1))

print(es_primo(9999999999973))
# es_primo: 0.56682 segundos
# True
```

Es decir, hemos modificado el comportamiento de la función sin realmente cambiarla. La hemos decorado.

También puedes modificar la función para que devuelva otros argumentos. En este caso devolvemos el tiempo que se tardó en ejecutar además de si es primo o no.

```
def mide_tiempo(func):
    def wrapper(*args, **kwargs):
        t0 = time.time()
        # Devolvemos el resultado
        # y el tiempo que tardó
        return func(*args, **kwargs), time.time() - t0
    return wrapper
```

---

Aunque ambos ejemplos son válidos, hay un matiz importante. Si nos fijamos en los metadatos de la función, estos hacen referencia al decorador no al de nuestra función. Por ejemplo.

```
# El decorador ha cambiado los metadatos
print(es_primo.__name__)
# wrapper
```

Si queremos conservar los metadatos, como el nombre y la documentación, una buena práctica es usar `wraps` de la siguiente manera. Suele ser común ver los decoradores definidos de esta manera.

```
from functools import wraps
import time

def mide_tiempo(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        t0 = time.time()
        return func(*args, **kwargs), time.time() - t0
    return wrapper
```

Si ahora accedes al nombre, puedes ver como lo mantiene.

```
# Con wraps ahora los metadatos no cambian
print(es_primo.__name__)
# es_primo
```

Por otro lado, también es posible pasar parámetros a nuestros decoradores. Al igual que una función puede tener argumentos de entrada y valores por defecto.

Veamos un ejemplo con un decorador que toma intenta ejecutar una función múltiples veces en el caso de que falle. Tenemos dos parámetros:

-  `retries` : Define el número de veces que se intenta ejecutar la función. Después de este número de intentos, se considera

---

que ha fallado.

- `z` `backoff_seconds` : Tiempo de espera entre intentos consecutivos. Es importante definir esto, ya que si una llamada falla, si intentamos sin esperar, es posible que nos encontremos con el mismo error.

Este decorador puede ser útil cuando tengamos funciones que a veces pueden fallar. Nos permitirá intentar de nuevo cuando falle hasta un número máximo de veces.

```
def retry(retries, backoff_seconds=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for intento in range(1, retries + 1):
                try:
                    print(f"Intento {intento}")
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Error: {e}")
                    if intento == retries:
                        raise e
                    time.sleep(backoff_seconds)
            return wrapper
    return decorator
```

Ahora podemos utilizar nuestro decorador de la siguiente manera. Con `random` intentamos reproducir una función que puede fallar con una probabilidad del 30%. Imagina que esto es una petición a un servidor externo o base de datos.

Puedes ver como si se produce el fallo, se intenta otra vez hasta un número máximo de 3 veces, esperando 2 segundos entre intento e intento.

```
@retry(retries=3, backoff_seconds=2)
def funcion_falla():
    import random
```

```
if random.random() < 0.7:  
    raise ValueError("Error aleatorio")  
print("Completado")
```

## 4.7 Yield y Generadores

Los generadores y `yield` fueron introducidos en la PEP255. Su uso devuelve un iterador *lazy*. A continuación veremos que significa esto de *lazy*.

Digamos que un iterador *lazy* no almacena los valores en memoria. Los permite generar “al vuelo”. De ahí su nombre. No los almacena. Veamos dos ejemplos

- `[ ]`: Crea una lista de 1000 elementos. Esto NO es *lazy*.
- `( )`: Crea un generador de 1000 elementos. Esto SI es *lazy*.

```
no_lazy = [i for i in range(1000)] # NO lazy  
si_lazy = (i for i in range(1000)) # SI lazy
```

A primera vista podría parecer que son lo mismo, ya que el resultado del siguiente código es idéntico.

```
for i in no_lazy: print(i)  
for i in si_lazy: print(i)
```

Sin embargo son dos cosas distintas. El tipo del primero es `list` y el segundo `generator`.

```
print(type(no_lazy)) # <class 'list'>  
print(type(si_lazy)) # <class 'generator'>
```

Podemos ver que el tamaño que ocupan en memoria es distinto. Aquí tenemos la primera ventaja de los iteradores *lazy*. Apenas ocupan memoria.

```
import sys
print(f"a: {sys.getsizeof(no_lazy)} bytes")
# a: 8856 bytes

print(f"b: {sys.getsizeof(si_lazy)} bytes")
# b: 112 bytes
```

Pero hay desventajas. El generador no puede ser indexado.

```
print(no_lazy[50]) # 50
print(si_lazy[50]) # TypeError
```

El generador no puede ser iterado dos veces.

```
for i in si_lazy: pass
for i in si_lazy: print(i)
```

Vistas las diferencias, vayamos a lo práctico. Los generadores nos permiten generar valores bajo demanda. Esto nos permite ahorrar recursos. Como has visto los generadores consumen una cantidad de memoria fija. El inconveniente es que los valores no están ahí, sino que se generan según se necesitan.

Esto es muy útil cuando trabajas con datos y ficheros muy grandes. Es común que cuando abres un fichero, como la función `open()` no estés cargando todo el fichero en memoria, sino creando una especie de generador. Este te permite iterar el fichero, pero no está cargado en memoria. Se va cargando según hace falta.

Hemos visto como crear un generador con `( )` usando *list comprehensions*, pero también lo puedes crear usando `yield`.

```
def mi_generador():
    yield 0
    yield 1
    yield 2
```

---

Como puedes ver es un `generator`.

```
print(type(mi_generador()))
# <class 'generator'>
```

Y los puedes usar de la siguiente manera.

```
for i in mi_generador():
    print(i)
# 0, 1, 2
```

Como un generador es un iterador, podemos acceder manualmente al siguiente elemento

```
print(next(g)) # 0
print(next(g)) # 1
print(next(g)) # 2
```

Cuando se llega al final, obtendrás `StopIteration`. Es decir, a la 4 vez que llamas a `next`.

```
print(next(g)) # StopIteration
```

Por ejemplo, puedes crear un generador de números primos. Primero definimos `es_primo` para saber si un número es primo.

```
def es_primo(n: int) -> bool:
    return n > 1 and all(n % i != 0 for i in range(2,
        int(n**0.5) + 1))
```

Y creamos nuestro generador de números primos. Observa que usamos `yield`. Siempre que pienses en generadores, piensa en `yield`.

```
def primos(n):
    num, contador = 2, 0
    while contador < n:
        if es_primo(num):
            yield num
```

```
contador += 1  
num += 1
```

En este ejemplo generamos los `5` primeros números primos. La ventaja de esto es que se van generando según hacen falta.

```
for p in primos(5):  
    print(p)  
# 2, 3, 5, 7, 11
```

Además, existen muchas funciones *generator friendly* como: `sum()`, `any()`, `all()`, `min()`, `max()`, `map()`, `filter()` o `enumerate()` entre otras. Esto es muy útil, ya que permiten trabajar con los datos del generador sin tener que cargar todos sus elementos en memoria.

Por ejemplo, podemos sumar los 100000 primeros números primos con `sum()` usando nuestro generador.

```
print(sum(primos(100000))) # 62260698721
```

Por último, podemos usar las *list comprehension* como hemos visto anteriormente, pero usando generadores. Un ejemplo:

- Generamos los `1000` primeros números primos.
- Elevamos cada uno al cuadrado `**2`.
- Sumamos todo con `sum`.

```
x = sum(x**2 for x in primos(100))  
print(x)  
# 8384727
```

## 4.8 Lambdas y Programación Funcional

Podemos definir las funciones lambda como funciones para vagos. Se pueden declarar en una línea y no hace falta ni asignarlas a una

---

variable. Esto es una función lambda.

```
lambda a, b : a + b
```

Es opcional, pero la puedes asignar a una variable para darle nombre.

```
suma = lambda a, b: a + b
```

Una vez tenemos la función, es posible llamarla como si de una función normal se tratase. Como puedes ver, son equivalentes a las funciones “normales” que hemos visto anteriormente.

```
suma(2, 4)
```

Las funciones lambda están muy relacionadas con la programación funcional. Este es un paradigma de programación donde todo gira en torno a las funciones. Se suele decir que son estas son *first class citizen*.

Por otro lado, en programación funcional es común el uso de las siguientes primitivas, que son aplicadas sobre iterables como listas:

-  `map` : Modifica cada elemento.
-  `filter` : Filtra los elementos que cumplen una condición.
-  `reduce` : Acumula todos los elementos reduciéndolos a un valor con una lógica determinada.

Veamos estas tres funciones en acción combinadas con funciones `lambda`. Imagina que tenemos la siguiente lista.

```
nums = [1, 2, 3, 4, 5]
```

Con `map` podemos modificar cada elemento usando una función `lambda`. Modificamos cada elemento elevándolo al cuadrado.

---

Pero tiene truco. Lo que se nos devuelve no es la lista modificada. Es un iterador.

```
print(map(lambda x: x**2, nums))
# <map object at 0x1046ff940>
```

Al ser un iterador, puedes iterar sobre él de la siguiente forma, observando cómo el resultado son los números elevados al cuadrado. Esto está estrechamente relacionado con los generadores y el concepto de *lazy* visto antes.

```
for i in map(lambda x: x**2, nums):
    print(i)
# 1, 4, 9, 16, 25
```

Si quieres obtener la lista, puedes convertir el iterador a `list`.

```
nums_cuadrado = list(map(lambda x: x**2, nums))
print(nums_cuadrado)
# [1, 4, 9, 16, 25]
```

Hasta aquí nada nuevo. Puedes obtener el mismo resultado con bucles de toda la vida.

```
nums_cuadrado = []
for x in nums:
    nums_cuadrado.append(x**2)
print(nums_cuadrado)
# [1, 4, 9, 16, 25]
```

Con `filter` podemos filtrar los valores. Por ejemplo, nos quedamos con los pares.

```
nums_pares = list(filter(lambda x: x % 2 == 0, nums))
print(nums_pares)
# [2, 4]
```

Que es equivalente a hacer lo siguiente.

```
nums_pares = []
for x in nums:
    if x % 2 == 0:
        nums_pares.append(x)

print(nums_pares)
# [2, 4]
```

Y con `reduce` podemos acumular todos los valores en uno. Por ejemplo, sumamos todos.

```
from functools import reduce

suma = reduce(lambda x, y: x + y, nums)
print(suma)
# 15
```

Lo que es equivalente a esto.

```
suma = 0
for x in nums:
    suma += x
print(suma)
# 15
```

Como puedes ver, `map`, `filter` y `reduce` nos permite hacer cosas que ya sabíamos hacer pero de otra forma. Nos evita los loops y se aproxima más a como se haría en programación funcional.

También existe un pequeño matiz. Ambas `map` y `filter` son *lazy*, lo que significa que devuelven un iterador que calcula los valores según se necesitan. Esto consume muy poca memoria.

El caso de `reduce` es distinto ya que usa *eager evaluation*. Es decir, si calcula el valor resultante.

Aunque anteriormente hemos usado funciones lambda, también podemos usar funciones normales. Aunque requieren alguna línea

---

más de código.

```
def cuadrado(x):
    return x ** 2

nums_cuadrado = list(map(cuadrado, nums))
print(nums_cuadrado)
# [1, 4, 9, 16, 25]
```

Por último, veamos un ejemplo práctico. Tenemos una lista de personas con diferentes atributos.

```
personas = [
    {'Nombre': 'Alicia', 'Edad': 22, 'Sexo': 'F'},
    {'Nombre': 'Bob', 'Edad': 25, 'Sexo': 'M'},
    {'Nombre': 'Charlie', 'Edad': 33, 'Sexo': 'M'},
    {'Nombre': 'Diana', 'Edad': 15, 'Sexo': 'F'},
    {'Nombre': 'Esteban', 'Edad': 30, 'Sexo': 'M'},
]
```

Podemos quedarnos con los nombres de todos.

```
nombres = list(map(lambda p: p['Nombre'], personas))
print(nombres)
# ['Alicia', 'Bob', 'Charlie', 'Diana', 'Esteban']
```

O podemos filtrar los mayores de 32 años.

```
mayores_25 = list(filter(lambda p: p['Edad'] > 32,
                         personas))
print(mayores_25)
# [{'Nombre': 'Charlie', 'Edad': 33, 'Sexo': 'M'}]
```

También podemos ordenar por edad y quedarnos con los nombres. Podemos comprobar que Diana ahora está la primera.

```
nombres_ordenados = list(
    map(lambda x: x["Nombre"],
        sorted(personas, key=lambda p: p['Edad'])))
print(nombres_ordenados)
```

```
# ['Diana', 'Alicia', 'Bob', 'Esteban', 'Charlie']
```

Esta es una de las cosas interesantes de Python. Es un lenguaje de programación multiparadigma, lo que significa que podemos hacer lo mismo de maneras muy diferentes. En este apartado hemos visto como hacerlo de forma más funcional.

## 4.9 Funciones asíncronas

Hasta ahora hemos visto código y funciones que se ejecutan de manera secuencial. Por orden. Una instrucción después de otra. Hasta que una función no se acaba, la siguiente no empieza.

Sin embargo esto puede resultar ineficiente. Imagina una función que realiza una petición a un servidor externo en el otro extremo del mundo. Esta petición tardará varios segundos. ¿Qué hacemos mientras? ¿Esperar sin hacer nada?

Esto no resulta eficiente, ya que esta petición externa está bloqueando nuestro programa. Hasta que el servidor no responda, estaremos esperando sin hacer nada. Podemos aprovechar este tiempo para hacer otras tareas.

Por suerte, Python nos ofrece herramientas para programación asíncrona, donde nuestro programa puede seguir ejecutándose mientras se espera. Esto es muy usado en:

- Web scrapping.
- Bases de datos.
- Entrada y salida con ficheros.
- Desarrollo de interfaces gráficos.

En todos estos casos en algún momento estamos bloqueados esperando una respuesta externa. A continuación veremos las diferencias

---

entre:

- **Programación síncrona:** El modelo tradicional visto hasta ahora, donde terminamos la tarea anterior para ejecutar la siguiente.
- **Programación asíncrona:** Cuando una tarea nos bloquea, continuamos ejecutando otras, lo que nos permite realizar múltiples tareas “en paralelo”.

Veamos un ejemplo síncrono. El siguiente código crea 10 procesos. Imagina que el `sleep` simula el tiempo que tarda un servicio externo en responder. Este programa tarda 10 segundos en ejecutarse. Hasta que no se acaba un proceso, no se empieza el siguiente.

```
import time
def proceso(id_proceso):
    time.sleep(1)
    print("Acaba proceso:", id_proceso)

[proceso(i) for i in range(10)]
```

Veamos un ejemplo asíncrono usando `asyncio`. En este caso cuando se detecta que estamos bloqueados, se continúa ejecutando otro código. Por lo tanto se inician los 10 procesos a la vez, y tras 1 segundo se completa todo. Hemos pasado de `10` segundos a `1`.

```
import asyncio

async def proceso(id_proceso):
    await asyncio.sleep(1)
    print("Acaba proceso:", id_proceso)

async def main():
    await asyncio.gather(*[proceso(i) for i in range(10)])

asyncio.run(main())
```

---

Es importante notar que en este caso el orden no está garantizado. Y menos si dependemos de algo externo cuyo tiempo de respuesta puede ser variable. El uso de `await` indica donde debemos esperar.

Por otro lado, existen paquetes como `threading` que permiten realizar tareas similares aunque con un modelo de concurrencia totalmente distinto, fuera del alcance de este libro.

Suele ser útil cuando se trabaja con librerías que no son compatibles con `async` y permite aprovechar una CPU con varios núcleos. Notar que no ofrece ejecución paralela debido al GIL.

```
import threading
import time

def proceso(id_proceso):
    time.sleep(1)
    print("Acaba proceso:", id_proceso, flush=True)

threads = [threading.Thread(target=proceso, args=(i,))
           for i in range(10)]
[i.start() for i in threads]
[i.join() for i in threads]
```

También existe el paquete `multiprocessing` que si ofrece ejecución paralela, y nos podemos beneficiar de múltiples CPUs. Esta librería resulta útil para realizar tareas intensivas en cálculo como multiplicar matrices muy grandes.

```
from multiprocessing import Process
import time

def proceso(id_proceso):
    time.sleep(1)
    print(f"Acaba proceso: {id_proceso}", flush=True)

def main():
    procesos = [Process(target=proceso, args=(i,)) for
               i in range(10)]
```

---

```
[p.start() for p in procesos]
[p.join() for p in procesos]
if __name__ == '__main__':
    main()
```

---

## 5 Programación Orientada a Objetos



La programación orientada a objetos es un paradigma que existe en casi todos los lenguajes modernos. Es una forma de pensar y estructurar el código que introduce los siguientes conceptos:

- **Clase**: Estas son el molde que permiten crear objetos con diferentes características particulares. Se crean con `class`.
- **Objeto**: Instancias particulares de una clase con diferentes propiedades.
- **Método**: Son las funciones de las clases y se definen con `def`.
- **Herencia**: Permite a una clase heredar y extender el comportamiento de otra.
- **Duck typing**: Permiten tratar a diferentes clases de la misma manera, siempre que tengan los métodos necesarios.

Veamos todos estos ejemplos y como los puedes aplicar en tus programas de Python.

## 5.1 Introducción y contexto

Esto es una clase. La clase `Persona`.

```
class Persona:  
    def __init__(self, nombre, apellido, edad):  
        self.nombre = nombre  
        self.apellido = apellido  
        self.edad = edad  
    def mayor_edad(self):  
        return self.edad >= 18
```

Esto es un objeto. El objeto `p1` de la clase `Persona`.

```
p1 = Persona("Ana", "Ruiz", 20)
```

Esto es un método. El método `mayor_edad`.

```
p1.mayor_edad()
```

Y esto son los atributos, `nombre`, `apellido` y `edad`.

```
print(p1.nombre) # Ana  
print(p1.apellido) # Ruiz  
print(p1.edad) # 20
```

La idea de clase es genérica. Pueden existir diferentes objetos de la misma clase. Decimos que `p1` y `p2` son dos objetos de la clase `Persona`.

- ↔ Son iguales en cierto modo. Se definen con el mismo patrón.
- ✘ Pero diferentes, ya que cada una tiene sus particularidades.

```
p1 = Persona("Ana", "Ruiz", 20)  
p2 = Persona("Bob", "Lopez", 20)
```

Ambas tienen los mismos atributos pero con diferente valor.

```
print(p1.nombre) # Ana
```

```
print(p2.nombre) # Bob
```

Aunque la programación orientada a objetos se puede complicar tanto como queramos, esto son los cuatro conceptos fundamentales sobre los cuales construiremos el resto.

Para entender mejor la programación orientada a objetos y sus ventajas, vamos a escribir el ejemplo anterior sin usarla. Asumamos por un momento que no tenemos clases ni objetos.

El objetivo es el mismo, almacenar los datos de una persona y operar sobre ella. Por ejemplo, con una lista podemos almacenar nombre, apellido y edad.

```
persona = ["Juan", "Prieto", 23]
```

Ahora podemos definir la función `mayor_edad` que recibe como argumento la `persona` y nos dice si es mayor de edad.

```
def mayor_edad(persona):  
    return persona[2] >= 18
```

Llegados a este punto tenemos algo similar a la clase `Persona` vista anteriormente, pero ya empezamos a ver alguna desventaja:

- Acceder a los campos es poco legible. Hacer `persona[0]` es menos claro que `persona.nombre`.
- No hay control sobre la longitud de la lista. Alguien podría crear una lista `persona` con más campos. Nada protege de esto.
- La función `mayor_edad` no está relacionada con la persona. Es una función a parte.
- La variable `persona` es accesible por cualquiera. Cualquier podría modificarla sin ninguna restricción.

Hemos por tanto detectado ya algunos de los problemas que la pro-

---

gramación orientada a objetos puede resolver.

Estos son los conceptos asociados a la programación orientada a objetos más importantes. Una vez los entiendes no podrás volver atrás:

- Herencia: Permite a una clase heredar de otra, heredando todos sus métodos y atributos. Una clase `Persona` puede heredar de una clase `Humano`.
- Cohesión: Cada clase tiene cosas relacionadas entre sí. Se busca que `Persona` tenga cosas de la persona. No tendría el color de su casa. Eso sería un atributo otra clase `Casa`.
- Abstracción: Oculta detalles complejos de dentro de la clase al exterior. El mando del televisor es complejo por dentro, pero a un usuario se muestra solo el volumen y canal. Abstira la complejidad de su interior.
- Polimorfismo: En Python relacionado con el *duck typing*. Si dos clases diferentes tienen los mismo métodos, pueden ser tratadas como iguales.
- Acoplamiento: Medido como el grado de dependencia entre clases. La programación orientada a objetos permite reducirlo.
- Encapsulamiento: La programación orientada a objetos permite ocultar detalles internos. El motor de tu coche esta oculto. El ocultarlo minimiza que alguien lo pueda estropear manipulándolo sin saber.

A continuación vemos de forma práctica como trabajar en Python con programación orientada a objetos.

## 5.2 Crea tus clases y objetos

Python nos permite crear una clase en dos líneas de código utilizando `class`. El siguiente código crea una clase `Persona`. Con `pass` in-

---

dicamos que es una clase vacía. Más adelante veremos cómo añadir atributos y métodos a nuestra clase.

```
# Crea una clase
class Persona:
    pass
```

Es importante notar que de acuerdo a las convenciones de Python, se nombra a las clases en *CamelCase*.

Así si:

- `Persona` ✓
- `CuentaBancaria` ✓

Así no:

- `clase_b` ✗
- `claseDeTipo` ✗

Ahora que tenemos nuestra clase, podemos crear un objeto de la misma. Para ello utilizamos el nombre de la clase seguido de `()`. Esto invoca al constructor de la clase. En este caso el constructor no recibe ningún parámetro.

```
# Crea un objeto
persona = Persona()
```

Es importante entender la diferencia entre **clase** y **objeto**:

- La **clase** hace referencia a algo genérico, en nuestro caso a la idea persona.
- El **objeto** hace referencia a algo concreto, en nuestro caso una persona concreta.

```
print(Persona)
# Salida: <class '__main__.Persona'>
```

```
print(persona)
# Salida: <__main__.Persona object at 0x1030a99d0>
```

Usando `type` podemos saber la clase a la que pertenece un objeto.

```
print(type(persona))
# Salida: <class '__main__.Persona'>
```

Llegados aquí ya tenemos nuestra clase, pero vacía. Vamos a empezar a añadirle cosas.

### 5.3 Crea tu constructor

Cuando usas `()` con la clase, como `Persona()` estás usando el constructor. El constructor permite crear un objeto de una clase determinada. Así se construye un objeto de la clase `Persona`.

```
class Persona:
    pass
p = Persona()
```

Pero en esta clase no hemos definido el constructor. Vamos a definirlo.

```
class Persona:
    def __init__(self, nombre, apellido, edad):
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad
```

Ahora que el constructor acepta tres argumentos, podemos construir nuestro objeto de la siguiente manera.

```
p = Persona("Ana", "Ruiz", 20)
```

También podemos construir el objeto así.

```
p = Persona(  
    nombre="Ana",  
    apellidos="Ruiz",  
    edad=20)
```

El constructor, al igual que cualquier función puede tener argumentos por defecto. Estos serán usados si no se proporciona ese argumento

```
class Persona:  
    def __init__(self, nombre='', apellido='', edad=0)  
        :  
            self.nombre = nombre  
            self.apellido = apellido  
            self.edad = edad
```

En este caso como no proporcionamos `edad` se usa el valor por defecto `0`.

```
p = Persona("Ana", "Ruiz")  
print(p.nombre) # Ana  
print(p.edad) # 0
```

El constructor `__init__` es también útil para verificar que los argumentos con los que construimos el objeto son correctos. En este caso verificamos que la `edad` no es negativa.

```
class Persona:  
    def __init__(self, nombre, apellido, edad):  
        if edad < 0:  
            raise Exception("Edad < 0")  
        self.nombre = nombre  
        self.apellido = apellido  
        self.edad = edad  
  
p = Persona("Ana", "Ruiz", -10)  
# Exception: Edad < 0
```

Aunque esta no es la forma de hacerlo, Python permite crear un

---

objeto de la siguiente manera.

```
# Funciona, pero no lo uses.  
persona = Persona.__new__(Persona)  
Persona.__init__(persona, "Ana", "Ruiz", 20)
```

Se podría decir que Python internamente convierte todo lo que hay dentro de `()` y se lo pasa a la función. Pero no lo hagas así.

## 5.4 Crea tus atributos

Toda clase define unos atributos. En programación orientada a objetos y Python, existen dos tipos de atributos:

- **De clase**: Son atributos comunes a todos los objetos de la clase. Todos los objetos ven el mismo valor.
- **De instancia**: Son atributos particulares de cada objeto. Cada objeto tiene un valor diferente.

El atributo `contador` es un atributo de clase. Todos los objetos de la clase `Persona` tendrán el mismo valor.

Los atributos `nombre`, `apellidos` y `edad` son atributos de instancia. Cada objeto de la clase `Persona` tendrá un valor distinto.

```
class Persona:  
    # Atributo de clase  
    contador = 0  
  
    # Atributos de instancia  
    def __init__(self, nombre, apellido, edad):  
        self.nombre = nombre  
        self.apellido = apellido  
        self.edad = edad  
  
    Persona.contador += 1
```

---

Si creamos dos personas distintas.

```
p1 = Persona("Juan", "Prieto", 23)
p2 = Persona("Ana", "Fuentes", 25)
```

Ambas comparten el valor de `contador`. Atributo de  clase.

```
print(p1.contador) # 2
print(p2.contador) # 2
```

Pero cada una tiene su valor de `nombre`. Atributo de  instancia.

```
print(persona1.nombre) # Juan
print(persona2.nombre) # Ana
```

Otra diferencia es que como `contador` es un atributo de clase, puedes acceder a él desde la clase.

```
print(Persona.contador) # 2
```

Como apunte final y para ser estrictamente correcto, en este ejemplo deberíamos también reducir el `contador` si se destruye el objeto.

```
class Persona:
    contador = 0

    def __init__(self, nombre, apellido, edad):
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad

        Persona.contador += 1

    def __del__(self):
        Persona.contador -= 1
```

Si ahora alguien destruye un objeto con `del`, el contador se actualizará en consecuencia.

```
p1 = Persona("Ana", "Ruiz", 0)
print(Persona.contador) # 1
del p1
print(Persona.contador) # 0
```

## 5.5 Crea tus métodos

Una clase tiene un conjunto de funciones asociadas que permiten trabajar con ella. A estas funciones se las conoce como métodos y hay los siguientes tipos:

- **Mágicos**: También conocidos como métodos *dunder*. Son métodos definidos por Python con comportamientos especiales.
- **De instancia**: Son los métodos “normales” y más comunes, que hemos definido anteriormente. Están pensados para acceder o modificar alguno de los atributos del objeto o instancia. Por ello, tienen acceso al mismo a través de `self`.
- **De clase**: Únicamente pueden acceder y modificar los atributos de clase, no pudiendo modificar objetos concretos ya que no se tiene conocimiento de ellos. Acceden a la clase a través de `cls`.
- **Estáticos**: No pueden acceder a atributos ni de objetos ni de clase. Por lo tanto, únicamente pueden acceder a los parámetros que se pasan por entrada.

A continuación vemos ejemplos de todos estos métodos para ayudarte a elegir la próxima vez que tengas que definir uno.

Los métodos mágicos son los que empiezan y acaban por `--` como `__init__`. Los veremos más en detalle en la sección de métodos *dunder* o mágicos. Por ejemplo, el constructor visto antes es un método mágico.

```
class Persona:  
    def __init__(self, nombre):  
        self.nombre = nombre
```

💡 Los métodos de instancia son los más usados y comunes. Acceden a la información del objeto y nos permiten cambiar o consultar información sobre ellos. El `self` hace referencia al objeto sobre el que se llaman. Todo método de instancia tiene el `self`.

```
class Persona:  
    def __init__(self, edad):  
        self.edad = edad  
  
    # Método de instancia  
    def mayor_edad(self):  
        return self.edad >= 18  
  
p = Persona(20)  
print(p.mayor_edad()) # True
```

⚠️ Los métodos de clase solo pueden acceder a los atributos de la clase. No pueden ni acceder ni modificar los atributos concretos de cada objeto.

A continuación vemos una clase `Persona` que tiene un atributo de clase `contador`. Este atributo se incrementa cada vez que se crea una persona. El método de clase `total_personas` nos devuelve el número de personas que han sido creadas, pero no tiene acceso a `nombre`.

Fíjate en que el método de clase usa `cls`. Es decir, tiene acceso a la clase.

```
class Persona:  
    contador = 0  
  
    def __init__(self, nombre):  
        self.nombre = nombre  
        Persona.contador += 1
```

```
# Método de clase
@classmethod
def total_personas(cls):
    return cls.contador

p1 = Persona("Ana")
p2 = Persona("Juan")

print(Persona.total_personas()) # 2
```

🔒 Los métodos estáticos no tienen acceso ni a los atributos de clase ni de instancia. Son funciones agrupadas bajo una clase. Suelen ser útiles cuando queremos agrupar un conjunto de funciones relacionadas bajo el mismo *namespace*. Como puedes ver no creamos objetos, ya que estos métodos se acceden directamente desde la clase.

Estos métodos estáticos no tienen acceso ni a `cls` ni a `self` como los vistos anteriormente.

```
class Calculadora:
    @staticmethod
    def suma(a, b):
        return a + b

    @staticmethod
    def resta(a, b):
        return a - b

print(Calculadora.suma(10, 5))    # 15
print(Calculadora.resta(10, 5))  # 5
```

## 5.6 Decorador setter

La programación orientada a objetos nos ayuda a proteger los atributos de nuestra clase de modificaciones no deseadas. Esto sigue el principio de encapsulación. Igual que el motor del coche está oculto al exterior para protegerlo, Python nos permite proteger los atributos

---

de nuestras clases.

Imagina que quieres proteger el atributo `edad` para que no se pueda usar un valor negativo. Una forma de hacerlo es verificando la `edad` en el constructor.

```
class Persona:  
    def __init__(self, nombre, apellido, edad):  
        if edad < 0:  
            raise Exception("Edad < 0")  
        self.nombre = nombre  
        self.apellido = apellido  
        self.edad = edad
```

Esto protege de lo siguiente.

```
p = Persona("Ana", "Ruiz", -1)  
# Exception: Edad < 0
```

Sin embargo no estamos protegidos de lo siguiente. Construimos el objeto con una edad válida, pero después la modificamos.

```
p = Persona("Ana", "Ruiz", 20)  
p.edad = -1  
print(p.edad) # -1
```

Para solucionar esto podemos usar los siguientes decoradores, que van muy de la mano:

- `edad.setter` : Permite añadir una lógica a ejecutar cuando se modifique el atributo, en este caso `edad` .
- `property` : Permite acceder a un atributo. Lo veremos en detalle más adelante.

```
class Persona:  
    def __init__(self, nombre, apellido, edad):  
        self.nombre = nombre  
        self.apellido = apellido
```

```
    self.__edad = edad

@property
def edad(self):
    return self.__edad

@edad.setter
def edad(self, edad):
    if edad < 0:
        raise ValueError("Edad < 0")
    self.__edad = edad
```

Algunos apuntes:

- Usando la `--` en `__edad` le decimos a Python que queremos ocultar este atributo del exterior. Es como decir que no queremos que nadie lo modifique. Es interno a la clase.
- Con `@edad.setter` le decimos a Python que cualquiera que quiera modificar `edad` debe hacerlo a través de este método. Y este método tiene una lógica. Prohibimos que la edad sea negativa.

Como puedes ver ahora ya estamos protegidos de una edad negativa.

```
p = Persona("Ana", "Ruiz", 20)
p.edad = -1
# ValueError: Edad < 0
```

Esto es un ejemplo de la famosa encapsulación de la programación orientada a objetos. Encapsula los atributos aislandolos del mundo exterior y pudiendo ser únicamente modificados por unas funciones determinadas, que se encargan de verificar que todo es correcto.

Un apunte importante es que Python permite hacer de todo. Si realmente te empeñas y quieres saltarte la verificación, puedes poner una edad negativa usando `_Persona__edad`. Pero esto es algo más

---

avanzado y un campo al que no deberías acceder.

```
p = Persona("Ana", "Ruiz", 20)
p._Persona__edad = -1
print(p.edad) # -1
```

## 5.7 Decorador property

Anteriormente hemos definido el atributo `edad` para nuestra `Persona`. Sin embargo la edad cambia. En 5 años no tienes la misma edad.

Para que sea consistente, tendríamos que modificar continuamente la edad, y esto es poco práctico. Es mejor almacenar la fecha de nacimiento.

```
class Persona:
    def __init__(self, nombre, apellido, fecha_nac):
        self.nombre = nombre
        self.apellido = apellido
        self.fecha_nac = fecha_nac
```

Con la fecha de nacimiento, podemos calcular la edad. Es tan sencillo como ver el tiempo que ha pasado entre `fecha_nac` y el momento actual en el que estamos.

Podemos entonces definir un método `edad` que nos devuelva la edad, calculada a partir de la fecha de nacimiento. Mucho mejor.

```
from datetime import datetime
class Persona:
    #
    def edad(self):
        hoy = datetime.now()
        edad = hoy.year - self.fecha_nac.year
```

```
if (hoy.month, hoy.day) < (self.fecha_nac.  
    month, self.fecha_nac.day):  
    edad -= 1  
  
return edad
```

Ahora podemos acceder a la `edad` a través del método que hemos creado. Ya no necesitamos actualizar la edad según pasa el tiempo. Se calcula automáticamente.

```
p = Persona("Juan", "Prieto", datetime(1993, 11, 10))  
print(p.edad())
```

El ejemplo es perfectamente válido, pero ahora tenemos `edad()` (un método) en vez de lo que teníamos antes `edad` (un atributo). Puede entonces ser confuso porque ahora al nombre y la edad se acceden de maneras distintas. Y conceptualmente ambos son atributos.

```
print(p.nombre)  
print(p.edad())
```

Precisamente para esto tenemos el decorador `@property`. Es muy *Pythonic* y permite usar `edad` en vez de `edad()`.

```
class Persona:  
    ...  
    @property  
    def edad(self):  
        hoy = datetime.now()  
        edad = hoy.year - self.fecha_nac.year  
  
        if (hoy.month, hoy.day) < (self.fecha_nac.  
            month, self.fecha_nac.day):  
            edad -= 1  
  
        return edad
```

Ahora `edad` se puede acceder como si fuera un atributo. Sin la

---

vista anteriormente.

```
print(p.nombre)
print(p.edad)
```

Pero `edad` es un atributo un poco especial. Su valor no está almacenado como tal. Se calcula cada vez usando la fecha de nacimiento.

Visto esto, ya podemos entender los diferentes tipos de atributos con respecto a la forma de ser evaluados. Estos pueden ser:

-  **Eager:** El `nombre` está almacenado “tal cual” en memoria. No es necesario realizar ningún cálculo para devolverla. Es lo más rápido.
-  **Lazy:** La `edad` no está almacenada. Se calcula cuando se quiere acceder a ella, usando la fecha de nacimiento. Esto requiere realizar cálculos. Puede ser más lento dependiendo de la complejidad de los cálculos. Suele ir asociado a `@property` .

Unas notas para decidir el tipo de evaluación, *eager* o *lazy*:

- En casos como `nombre` está claro que queremos *eager*.
- Con `edad` está claro que queremos *lazy*, usando la fecha de nacimiento.
- En otros casos depende. Usar *lazy* con un atributo que requiere muchos cálculos y que hay miles de procesos accediendo a él cientos de veces por segundo, tal vez sea mala idea usar *lazy* ya que estará continuamente calculando el atributo.

## 5.8 Herencia de clases

La herencia nos permite crear una clase que hereda métodos y atributos de otra. Esto nos permite:

- 
- **Extender**: Podemos heredar de una clase y añadir nuevos métodos o atributos.
  - **Modificar**: Podemos modificar los métodos y atributos existentes.
  - **Reutilizar**: Podemos reutilizar el comportamiento de la clase original.

Volvemos a nuestra clase `Persona`.

```
class Persona:  
    def __init__(self, nombre, apellido, edad):  
        self.nombre = nombre  
        self.apellido = apellido  
        self.edad = edad  
  
    def mayor_edad(self):  
        return self.edad >= 18
```

Podemos crear una clase `Estudiante` que hereda de `Persona`.

```
class Estudiante(Persona):  
    def __init__(self, nombre, apellido, edad, curso):  
        super().__init__(nombre, apellido, edad)  
        self.curso = curso  
  
    def ultimo_curso(self):  
        return self.curso == 12
```

Esta nueva clase `Estudiante`:

- Extiende a `Persona` con un nuevo método `en_ultimo_curso`.
- Modifica a `Persona` añadiendo un atributo `curso`.
- Reutiliza a `Persona` heredando sus métodos existentes `mayor_edad`.

Como puedes ver en el método `__init__` usamos `super`. Esto nos permite acceder a métodos de la clase padre.

---

Para crear un objeto de la clase `Estudiante`, se hace como hemos visto anteriormente. Nada cambia.

```
e = Estudiante("Ana", "Ruiz", 10, 12)
print(e.ultimo_curso()) # True
```

Y puedes comprobar que hereda el método `mayor_edad` de la clase padre. Esta es la magia. En `Estudiante` no lo has definido, pero lo toma de `Persona`.

```
print(e.mayor_edad()) # False
```

Por otro lado, la herencia puede ser múltiple. En este caso una `ClaseC` hereda todos los métodos y atributos de `claseA` y `ClaseB`.

```
class ClaseA:
    def metodoA(self):
        return "metodoA"

class ClaseB:
    def metodoB(self):
        return "metodoB"

class ClaseC(ClaseA, ClaseB):
    def metodoC(self):
        return "metodoC"
```

Vamos como se heredan todos los métodos.

```
obj = ClaseC()
print(obj.metodoA()) # Salida: metodoA
print(obj.metodoB()) # Salida: metodoB
print(obj.metodoC()) # Salida: metodoC
```

Por último veamos un ejemplo práctico. Vamos a extender el comportamiento de la clase `list` de Python. Le vamos a añadir un método `media` que devuelva la media de todos sus valores.

```
class ListaConMedia(list):
    def media(self):
        return sum(self) / len(self)
```

Creamos la lista y usamos el método. Puedes comprobar que el resto de métodos de las `list` como `append` y `len` han sido heredados por nuestra `ListaConMedia`.

```
numeros = ListaConMedia([10, 20, 30, 40])
print("Media =", numeros.media())
# Media = 25.0
```

## 5.9 Polimorfismo y Duck Typing

El polimorfismo es un concepto de la programación orientada a objetos muy relacionado con la herencia. Permite que objetos que heredan de la misma clase puedan ser tratados de la misma manera.

Imagina una clase `Persona` de la que heredan dos clases `Profesor` y `Alumno`. El polimorfismo permite tratar a `Profesor` y `Alumno` de la misma manera, ya que heredan de la misma clase.

Por ejemplo, ambas clases tienen un `nombre` por lo que a una función le daría igual si un objeto es de una clase u otra.

En Python esto es un poco distinto. También mucho más flexible. Python tiene lo que se conoce como *duck typing*. Hay una frase asociada que dice: “If it walks like a duck and it quacks like a duck, then it must be a duck”.

Traducido sería: “Si camina como un pato y hace *cuac* como un pato, entonces debe ser un pato”.

Es una forma de decir que a Python le da igual la clase o el tipo. Si tiene los métodos necesarios, entonces sirve. En este ejemplo puedes

---

ver dos clases diferentes. Python intenta hacer `cuac` con cada animal. Como todos tienen el método `cuac`, le sirve.

```
class Perro:  
    def cuac(self):  
        print("Guau!")  
  
class Pato:  
    def cuac(self):  
        print("Cuac!")  
  
for animal in Perro(), Pato():  
    animal.cuac()  
  
# Guau! # Cuac!
```

Puede parecer lógico, pero en muchos otros lenguajes de programación esto daría un error. Ya que se trata de tipos diferentes que no comparten un interfaz común.

Lo siguiente también funciona. Una vez más, a Python le da igual el `objeto` que le pases. Mientras tenga la función `cuac`, es suficiente.

```
def cuac(objeto):  
    objeto.cuac()  
  
pe = Perro()  
gt = Pato()  
hablar(pe) # Guay!  
hablar(gt) # Cuac!
```

## 5.10 Métodos dunder

Los métodos mágicos o *dunder* de Python son los que empiezan y acaban con `--`, como `__init__`. El nombre de *dunder* viene de *double underscore*. Es decir, doble barra baja.

---

Estos son métodos definidos por Python que pueden ser usados internamente o asociados a operadores. Por ejemplo:

- Si usas `()` por debajo se llama a `__init__`.
- Si usas `+` con dos objetos por debajo se llama a `__add__`.
- Si usas `==` con dos objetos por debajo se llama a `__eq__`.

Dependiendo de lo que quieras hacer y de tu clase, te puede interesar definir estos métodos *dunder*. Esto es muy potente, ya que te permite definir el comportamiento de tu clase.

A continuación vemos los más conocidos. Usaremos como ejemplo una clase `Punto` con dos atributos `x` e `y`. Esta clase representa un punto en el espacio en dos dimensiones.

Método `__init__`. Como hemos visto anteriormente es el constructor y nos permite crear el objeto. Se llama al usar `()`. El método `__new__` tiene cierta relación, pero vamos a dejarlo por ahora.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
p = Punto(1, 2)  
  
print(p.x) # 1  
print(p.y) # 2
```

Método `__del__`. Se puede ver como lo contrario de `__init__`. Uno construye, otro destruye. Así puedes implementar la lógica de tu destructor.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
def __del__(self):
    print("Objeto destruido")
```

Este método se llamará cuando destruyas tu objeto con `del`. Tú decides lo que poner ahí. Es común verlo en sitios donde sea importante liberar recursos tras haber terminado.

```
p = Punto(1, 2)
del p
# Objeto destruido
```

Método `__str__`. Si hacemos `print` de nuestro objeto, la información que se nos muestra no es muy útil.

```
p = Punto(1, 2)
print(p)
# <__main__.Punto object at 0x102b5a100>
```

Si definimos `__str__` al usar `print` se mostrará lo que queramos. Es útil definir este método en tus clases, ya que ayuda a ver el contenido del objeto.

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"x={self.x} y={self.y}"

p = Punto(1, 2)
print(p) # x=1 y=2
```

Métodos `__add__` y `__sub__`. Imagina que quieres sumar `+` o restar `-` dos objetos. Estos métodos definen la lógica de la suma y de la resta.

- $\oplus$  Definimos la suma de dos puntos como la suma de las coordenadas `x` por un lado e `y` por otro.

- 
- Definimos la resta de dos puntos como la resta de las coordenadas `x` por un lado e `y` por otro.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, b):  
        return Punto(self.x + b.x, self.y + b.y)  
  
    def __sub__(self, b):  
        return Punto(self.x - b.x, self.y - b.y)  
  
    def __str__(self):  
        return f"x={self.x} y={self.y}"
```

Ahora que ya hemos definido ambos operadores, los podemos usar. Si no los defines, obtendrás `TypeError`.

```
p1 = Punto(5, 10) + Punto(3, 1)  
p2 = Punto(-1, 0) - Punto(-5, 0)  
print(p1) # x=8 y=11  
print(p2) # x=4 y=0
```

Métodos `__eq__` y `__ne__`. Estos dos métodos permiten definir cuando dos objetos son iguales o distintos. Consideramos dos objetos iguales si sus coordenadas son iguales.

```
class Punto:  
    # ...  
  
    def __eq__(self, b):  
        return self.x == b.x and self.y == b.y  
  
    def __ne__(self, b):  
        return not self == b  
  
igual = Punto(1, 1) == Punto(1, 1)  
print(igual) # True
```

---

---

Puedes comprobar que son iguales. También existen otros métodos como `__lt__` o `__gt__` que se usan para definir cuando un objeto es menor `<` o mayor `>` que otro.

```
print(Punto(1, 1) == Punto(1, 1)) # True
```

Métodos `__iter__` y `__next__`. Estos métodos permiten hacer que una clase sea iterable, es decir, que la podamos usar con un `for`. Uno define el objeto iterador, y el otro define el número siguiente a iterar. Definimos una clase que itera números pares hasta un valor `limite`.

```
class Pares:
    def __init__(self, limite):
        self.i = 0
        self.limite = limite

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= self.limite * 2:
            raise StopIteration
        par = self.i
        self.i += 2
        return par
```

Podemos generar los primeros 5 números pares de la siguiente manera.

```
for i in Pares(5):
    print(i)
# 0, 2, 4, 6, 8
```

Métodos `__enter__` y `__exit__`. Están relacionados con los *context manager* y los veremos con más detalle en el capítulo de excepciones.

---

Por ahora nos basta con saber que permiten ejecutar una acción al entrar a un bloque `with` y otra al salir. Hacen una especie de sándwich.

```
class GestorContexto:  
    def __enter__(self):  
        print("Entra")  
        return None  
  
    def __exit__(self, exc_type, exc_value, traceback):  
        :  
        print("Sale")  
  
with GestorContexto() as f:  
    print("Dentro")  
  
# Entra  
# Dentro  
# Sale
```

Una vez vistos los más importantes, veamos ahora una aplicación práctica usando el método *dunder* `__add__` visto anteriormente.

Tenemos una clase `Moneda` que puede almacenar valores en \$ y el €. *A priori* no es posible sumar \$ con €, ya que no podemos mezclar peras con manzanas.

Sin embargo, podemos definir `__add__` para que se realice una conversión a \$. Una vez ambos valores han sido convertidos, se suman.

```
class Moneda:  
    ratios = {'USD': 1.0, 'EUR': 1.10}  
  
    def __init__(self, moneda, cantidad):  
        self.moneda = moneda  
        self.cantidad = cantidad  
  
    def convertir(self, nueva_moneda):
```

```
if self.moneda != nueva_moneda:  
    return self.cantidad * Moneda.ratios[self.  
        moneda] / Moneda.ratios[nueva_moneda]  
return self.cantidad  
  
def __add__(self, other):  
    if isinstance(other, Moneda):  
        cantidad_usd = self.convertir('USD') +  
            other.convertir('USD')  
        return Moneda('USD', cantidad_usd)  
    raise TypeError("Error")  
  
def __str__(self):  
    return f'{self.moneda} {self.cantidad:.2f}'
```

Ahora podemos sumar diferentes monedas, ya que se realiza una conversión automática.

```
gasto1 = Moneda('USD', 5.25)  
gasto2 = Moneda('EUR', 7.99)  
  
print(gasto1 + gasto2) # USD 14.04
```

Existen otros métodos *dunder* que te invitamos a revisar, pero hemos cubierto los principales que te serán suficiente en la mayoría de tus programas.

## 5.11 Uso de dataclass

Si quieres definir una clase para almacenar información sin métodos adicionales, puedes usar `dataclass`. En unas pocas líneas puedes definirlo.

```
from dataclasses import dataclass  
  
@dataclass  
class Punto:  
    x: int
```

```
y: int
```

Y la puedes usar para crear un objeto así. Como ves, también permite imprimir el punto con sus atributos.

```
p = Punto(10, 20)
print(p)
# Punto(x=10, y=20)
```

Realmente no aportan nada nuevo. Puedes escribir un código con el mismo comportamiento como mostramos. Pero contiene mucho código repetitivo.

```
class Punto:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Punto(x={self.x}, y={self.y})"

    def __eq__(self, other):
        if isinstance(other, Punto):
            return self.x == other.x and self.y ==
                   other.y
        return False
```

En resumen, `dataclass` te permite definir una clase sencilla ahorrando código.

## 5.12 Monkey patching

El *monkey patching* es una técnica en Python que permite modificar el comportamiento de una clase en tiempo de ejecución. En otras palabras, nos permite cambiar un método por otro. Imagina que tenemos un método `saludo`.

---

```
class Persona:  
    def saludo(self):  
        print("¡Hola!")
```

Gracias al *monkey patching* podemos cambiar el método `saludo` por `nuevo_saludo` en tiempo de ejecución.

```
def nuevo_saludo(self):  
    print("Hola, mundo!")  
  
Persona.saluda = nuevo_saludo  
p = Persona()  
p.saluda() # Hola, mundo!
```

Esto es bastante interesante para pruebas. También si quieras modificar la funcionalidad de un código de un tercero rápidamente.

No obstante úsalo con cuidado, ya que puede resultar confuso. Por ejemplo, no es recomendable modificar los paquetes de Python, ya que todo el mundo espera un comportamiento determinado. Confundirías a los lectores de tu código.

---

## 6 Gestiona Errores con Excepciones



Las cosas no siempre salen como uno espera. A lo largo de la ejecución de un código nos podemos encontrar con problemas para continuar:

- 📁❌ Si intentas abrir un fichero que no existe.
- +!? Si intentas sumar `2` + `"2"`.
- 🌐🚫 Si intentas acceder a una web que no está disponible.

Python gestiona estos errores con excepciones, donde tenemos dos partes bien definidas:

- ⚾ `raise` : El que lanza la excepción indicando el problema que ha sucedido.
- 🤕 `except` : El que gestiona la excepción buscando una solución.

A continuación, veremos como detectar y manejar estas excepciones, buscando fallar de una manera elegante, ordenada y segura.

---

## 6.1 Gestión de errores

Las cosas no siempre salen como uno espera. A lo largo de su ejecución tu código se puede encontrar con una instrucción que no puede ejecutar. Esto puede suceder por diferentes motivos.

Un fichero que se quiere abrir pero no existe. Una división por cero. Una petición HTTP a una web que está caída.

Cada lenguaje gestiona los problemas de una forma diferente. Python lo hace con excepciones. Las excepciones son una forma de controlar el comportamiento de un programa cuando se produce un error o excepción.

Un programa sencillo puede no necesitar una gestión sofisticada de excepciones, pero existen programas que tienen que correr 24 horas al día 7 días a la semana, que están expuestos a servicios externos que pueden fallar, entradas de usuarios, etc.

Para gestionar errores adecuadamente tenemos que:

- 🔎 **Detectar**: Detectar que se ha producido un error. Cada lenguaje utiliza una manera distinta, siendo las excepciones la preferida de Python.
- 🚧 **Actuar**: Actuar dependiendo del tipo de error que se haya producido.

🔍 Los errores se pueden **detectar** de varias formas:

- 0 : **Código de errores**: La función devuelve `0` si no ha habido ningún error y `!=0` si ha habido error. Se asignan códigos, por ejemplo el `1` es un error, el `2` es otro distinto. Común en C.
- 📄: **Valor y código error**: La función devuelve dos valores, uno con lo esperado y otro con el código de error asociado, que es `None` si no se ha producido error. Común en Golang.

- 
- : **Excepciones**: La función no tiene que devolver nada asociado al error. Cuando se produce un error o excepción, se ejecuta la lógica para manejarla. Permiten diferenciar claramente el código normal del código a ejecutar cuando las cosas salen mal. Común en Python, Java, C++.

Y ante un error podemos **actuar** de diferentes formas:

- **Intentar otra vez**: Se define una lógica de *retry* indicando que queremos intentarlo un número determinado de veces cada cierto intervalo (*backoff*). Útil cuando hay esperanzas de que acabe funcionando.
- **Informar y continuar**: Se informa al usuario con un *log* que indique el error y continuamos. Útil cuando el error no es crítico y se puede continuar la ejecución.
- **Parar la ejecución**: Se para la ejecución del programa, terminando de manera ordenada y liberando los recursos. Útil cuando el error es crítico y no se puede seguir.

Dado que Python funciona principalmente con excepciones, a continuación veremos qué son, cómo manejarlas y cómo definirlas.

## 6.2 Excepciones en Python

Al igual que muchos otros lenguajes, Python permite gestionar errores con excepciones. Esta forma es bastante común, ya que permite separar la lógica de nuestras funciones de la lógica de la gestión del error.

También es menos propenso a errores, ya que si se produce una excepción, no pasará desapercibida. Otros métodos pueden hacer que pasen desapercibidas, lo que es muy peligroso. Que algo falle, pero no sepas que ha fallado es lo peor que puede pasar.

---

Python nos permite trabajar con excepciones con las siguientes palabras clave:

- `try` : Sección de código a ejecutar donde se podría producir una excepción.
- `except` : Sección de código a ejecutar si se produce una excepción. Podemos tener múltiples `except` donde cada uno indica el código a ejecutar para cada excepción distinta. En otros lenguajes se conoce como *catch*.
- `finally` : Sección de código a ejecutar se produzca o no excepción. Este código se ejecuta siempre.
- `else` : Sección de código a ejecutar si no se produce una excepción.
- `raise` : Permite lanzar una excepción. Útil si quieres definir una excepción propia con su lógica asociada.

Si no gestionamos las excepciones, la ejecución termina de forma abrupta indicando un error. Esto no es recomendable.

```
print(10/0)
# ZeroDivisionError: division by zero
```

Veamos el uso de `try` y `except` . El bloque `try` indica que existe la posibilidad que ese bloque de código tenga una excepción. El bloque `except` indica que hacer si se produce.

```
try:
    print(10/0)
except:
    print("Error")
# Error
```

Puedes ver como en este caso ya no se termina de forma abrupta. Hemos manejado nuestra excepción.

Es una buena práctica incluir más información sobre la excepción,

---

para saber que ha pasado realmente.

```
try:  
    print(10/0)  
except Exception as e:  
    print(f"Error: {e}")  
# Error: division by zero
```

Aunque lo más correcto es manejar la excepción determinada. El uso de `Exception` es genérico, maneja cualquier excepción. El uso de `ZeroDivisionError` es más apropiado, maneja una excepción concreta.

```
try:  
    print(10/0)  
except ZeroDivisionError as e:  
    print(f"Ha ocurrido una excepción: {e}")  
# Ha ocurrido una excepción: division by zero
```

Si tenemos un bloque `try` que puede lanzar múltiples excepciones, podemos tener múltiples `except`. Uno con cada excepción posible.

```
a, b, c = 5, 0, "1"  
try:  
    c = a/b  
    d = b + c  
except ZeroDivisionError:  
    print("No se puede dividir entre cero!")  
except TypeError:  
    print("Problema de tipos!")
```

Veamos el uso de `finally`. No importa si se produce o no excepción, el código del `finally` siempre se ejecuta.

```
try:  
    x = 10/0 # <- Intenta con 10/2  
except ZeroDivisionError:  
    print("No puedes dividir por cero")  
finally:
```

```
    print("Se ha terminado")
```

Veamos el uso del `else`. Este bloque se ejecuta si no se ha producido excepción.

```
try:  
    x = 10/0  
except ZeroDivisionError:  
    print("No puedes dividir por cero")  
else:  
    print("Todo ha salido bien")  
# No puedes dividir por cero
```

En este caso si, ya que `10/5` si se puede dividir.

```
try:  
    x = 10/5  
except ZeroDivisionError:  
    print("No puedes dividir por cero")  
else:  
    print("Todo ha salido bien")  
# Todo ha salido bien
```

Y un ejemplo más completo sería el siguiente. Nótese que es una función con fines didácticos.

```
def divide(x, y):  
    try:  
        resultado = x/y  
    except ZeroDivisionError:  
        print("No puedes dividir por cero")  
    else:  
        print("Todo ha salido bien:", resultado)  
        return resultado  
    finally:  
        print("Se ha terminado")
```

Si se produce una excepción.

```
divide(10, 0)
```

```
# No puedes dividir por cero  
# Se ha terminado
```

Si no se produce una excepción.

```
divide(10, 2)  
# Todo ha salido bien: 5.0  
# Se ha terminado
```

## 6.3 Lanza Excepciones

Una excepción tiene dos partes:

- 🚨 `raise` : El que lanza la excepción. Veremos a continuación como hacerlo.
- 🤑 `except` : El que la captura. Si nadie la captura, el programa terminará de forma abrupta. No queremos esto.

Por ejemplo, tenemos una función que acepta una `edad` y queremos asegurarnos de que no es negativa. Nadie puede tener `-5` años.

🚨 Podemos lanzar con `raise` una excepción cuando pase esto.

```
def set_edad(edad):  
    if edad < 0:  
        raise ValueError("No puede ser <0")  
    # ...
```

Si alguien intenta lo siguiente, se encontrará una excepción. Esto es una excepción sin capturar o manejar.

```
set_edad(-10)  
# ValueError: No puede ser <0
```

🤑 Pero la podemos capturar con `except` como hemos visto antes.

```
try:
```

```
    set_edad(-1)
except ValueError as e:
    print(f"Error: {e}")
# Error: No puede ser <0
```

También puedes usar la excepción genérica `Exception`. Pero es recomendable ser específico.

```
raise Exception("Información sobre tu excepción")
```

En programas más complejos donde existen muchas excepciones distintas puede ser interesante definir las nuestras.

Siguiendo con el ejemplo de la edad, vamos a crear diferentes excepciones que se lanzarán si:

- Es negativa.
- ↗ Es demasiado alta.
- ÷ Es decimal.

Definimos una excepción `EdadIncorrecta` que hereda de la excepción genérica `Exception`. Y definimos tres tipos. Incluimos también una breve descripción de cada una.

```
from enum import Enum
class EdadErrorTipo(Enum):
    NEGATIVA = 1
    DEMASIADO_ALTA = 2
    DECIMAL = 3

class EdadIncorrecta(Exception):
    def __init__(self, edad, error: EdadErrorTipo):
        self.edad = edad
        self.error = error
        super().__init__(self.get_error_message())

    def get_error_message(self):
        if self.error == EdadErrorTipo.NEGATIVA:
```

```
        return f"No puede ser negativa: {self.edad}\n    }\n    elif self.error == EdadErrorTipo.\n        DEMASIADO_ALTA:\n            return f"Irrealmente alta: {self.edad}"\n    elif self.error == EdadErrorTipo.DECIMAL:\n        return f"No puede ser decimal: {self.edad}\n    }\nelse:\n    return "Edad incorrecta"
```

Una vez tenemos la excepción definida la podemos lanzar de la siguiente manera.

```
def set_edad(edad):\n    if edad < 0:\n        raise EdadIncorrecta(edad, EdadErrorTipo.\n            NEGATIVA)\n    elif edad > 150:\n        raise EdadIncorrecta(edad, EdadErrorTipo.\n            DEMASIADO_ALTA)\n    elif not isinstance(edad, int):\n        raise EdadIncorrecta(edad, EdadErrorTipo.\n            DECIMAL)
```

Si es negativa.

```
set_edad(-10)\n# EdadIncorrecta: No puede ser negativa: -10
```

Si es demasiado alta.

```
set_edad(4000)\n# EdadIncorrecta: Irrealmente alta: 4000
```

O si es decimal.

```
set_edad(1.5432)\n# EdadIncorrecta: No puede ser decimal: 1.5432
```

---

El tener diferentes excepciones personalizadas como acabamos de ver puede ser interesante, ya que nos informa mejor del error.

Siempre que trabajemos con errores o gestión de excepciones es importante dar el máximo detalle posible, ya que este nos ayudará a entender mejor la causa del problema.

## 6.4 Uso de Context Managers

Como hemos visto el `finally` nos permite ejecutar un código se produzca o no excepción. Es común usarlo cuando abrimos un fichero, ya que una vez abierto es importante cerrarlo.

```
fichero = open('fichero.txt', 'r')
try:
    contenido = fichero.read()
finally:
    fichero.close()
```

Es importante ya que si lo dejamos abierto puede quedar bloqueado y no dejar a otros escribir en él. O pueden quedar recursos no liberados consumiendo memoria de manera innecesaria. O puede haber problemas de *file descriptor leaks* algo más avanzados.

Este problema se extiende más allá de los ficheros, siendo similar en bases de datos, *mutex*, *locks* o \*\*recursos de red. El patrón es el mismo siempre:

- Usas algo, tomando posesión de un recurso.
- Lo usas para lo que quieras.
- Una vez has terminado lo liberas para que otros puedan usarlo.

Con el `finally` podemos hacer esto, igual que en el ejemplo del fichero. Sin embargo imagina que te olvidas del `finally`. Ese recurso

---

quedaría sin liberar.

Para solucionar esto, tenemos los *context managers*. Además de abbreviar el código, gestionan automáticamente la liberación de recursos. Ya no es necesario añadir el `finally` explicitamente.

Este ejemplo hace lo mismo que el anterior, pero usando un *context manager*. Como ves no se llama al `.close()`, pero Python lo hace automáticamente por nosotros.

```
with open('fichero.txt', 'r') as fichero:  
    contenido = fichero.read()
```

Python nos garantiza que cuando salgamos del bloque `with` y sin importar lo que haya pasado, el fichero será cerrado. Así de fácil.

También puedes crear tu *context manager* definiendo los siguientes métodos.

```
class MiContextManager:  
    def __enter__(self):  
        print("Entra")  
        return self  
  
    def __exit__(self, exc_type, exc_value, traceback)  
        :  
        print("Sale")
```

Es como un sandwich:

- ➡ `__enter__` : Acción que se ejecuta al entrar en el bloque `with`.
- 🍔 Contenido dentro del `with`.
- ⚡ `__exit__` : Acción que se ejecuta al salir del bloque `with`.

Y lo puedes usar así.

```
with MiContextManager() as manager:
```

```
        print("Dentro del with")
# Entrá
# Dentro del with
# Sale
```

Aunque haya una excepción se garantiza que se ejecuta el `__exit__`. Con esto ya no hay problema de que se nos olvide el `finally`. Python hace el trabajo por nosotros. Mucho más seguro.

```
with MiContextManager() as manager:
    raise Exception("Error")
# Entrá
# Sale
# Exception: Error
```

El *context manager* `open` que vimos antes se podría escribir así. Podemos ver que el `__enter__` abre el fichero y el `__exit__` lo cierra pase lo que pase.

```
class MiOpen:
    def __init__(self, nombre, mode):
        self.nombre = nombre
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.nombre, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        :
        if self.file:
            self.file.close()
        return False
```

Lo puedes usar de forma similar.

```
with MiOpen('fichero.txt', 'r') as fichero:
    contenido = fichero.read()
```

---

Aunque es muy usado en casos donde haya que liberar recursos como cerrar un fichero, existen otros ejemplos muy interesantes. Podemos hacer uno que mida el tiempo.

```
import time
class MideTiempo():
    def __enter__(self):
        self.tiempo_empieza = time.time()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.elapsed = time.time() - self.
            tiempo_empieza
        print(f"Tiempo: {self.elapsed:.4f} segundos")
```

Esto medirá el tiempo que tarda en ejecutarse lo que haya dentro del bloque `with`.

```
with MideTiempo() as _:
    [i**2 for i in range(1000000)]
# Tiempo: 0.2037 segundos
```

Aunque fuera del alcance de este libro, se puede ver también en el paquete `threading`.

```
import threading
lock = threading.Lock()
with lock:
    print("El lock es nuestro")
```

Con bases de datos. Aunque este caso no cierra la conexión, sino que hace un *rollback* si algo sale mal.

```
import sqlite3
with sqlite3.connect('basedatos.db') as connection:
    result = connection.execute("SELECT nombre FROM
        tabla;")
```

---

También podemos crear ficheros temporales.

```
import tempfile

with tempfile.TemporaryDirectory() as tmpdir:
    print(f"Directorio temporal {tmpdir}")
```

Podemos unir varios o anidarlos.

```
with open('f1.txt', 'r') as f1, open('f2.txt', 'w') as f2:
    datos = f1.read()
    f2.write(datos)
```

En resumen. Cuando tengas un código que tome control de un recurso (como un fichero o base de datos), plantéate usar los *context managers* para asegurarte que una vez ese recurso ha sido utilizado, se libera para que el resto pueda usarlo.

## 6.5 Dos filosofías para manejar errores

Como hemos visto anteriormente muchas excepciones ocurren por motivos conocidos *a priori*.

Dividir por cero resulta en la excepción `ZeroDivisionError`.

```
print(2/0)
# ZeroDivisionError
```

Acceder a una clave que no existe en un diccionario resulta en la excepción `KeyError`.

```
c = {}
print(c['noexiste'])
# KeyError
```

En base a estos dos ejemplos, podemos pensar: Si no queremos ex-

---

cepciones, es fácil. No divides por cero. No accedes a una clave que no existe. De esta manera, ya no tendrás excepciones.

Justo esta reflexión nos abre la puerta a las dos formas de gestión de errores que tenemos en Python:

- 🧐 **Comprobar antes de actuar:** Se intenta que no se produzca el error o excepción, verificando antes todas las condiciones que lo podrían producir. En Inglés, LBYL (Look Before You Leap).
- 💥 **Dejar que falle:** No importa que se produzca un error, pero se añade una lógica para gestionarlo por si sucede. En Inglés, EAFP (Easier to Ask for Forgiveness than Permission). Es decir, mejor pedir perdón que permiso.

Esto es 🧐 LBYL. Antes de intentar acceder al elemento `dic["c"]` del diccionario, nos aseguramos de que existe, para evitar que se produzca un error.

```
if "c" in dic:  
    valor = dic["c"]  
else:  
    valor = None
```

Esto es 💥 EAFP. Accedemos directamente al valor `dic["c"]` sin importar si existe o no. Si existe bien. Si no existe, se producirá una excepción, pero la gestionamos.

```
try:  
    valor = dic["c"]  
except KeyError:  
    valor = None
```

Otro ejemplo de 🧐 LBYL. Antes de borrar el fichero, nos aseguramos de que existe.

---

```
if os.path.exists(ruta_fichero):  
    os.remove(ruta_fichero)
```

El equivalente EAFP. Borramos el fichero sin importar si existe o no. Si existe, se borra. Si no existe, lo gestionamos.

```
try:  
    os.remove(ruta_fichero)  
except FileNotFoundError:  
    print("no existe")
```

Es importante notar que ambas opciones son válidas, por lo que depende de tu caso cuál elegir. Algunas notas:

- En general EAFP puede resultar mejor, ya que Python está pensado para funcionar con excepciones.
- En casos puntuales LBYL puede sufrir de *race conditions*. Por ejemplo, se verifica que el fichero existe con `exists` pero podría pasar que un instante después ya no, por lo que `remove` podría fallar.
- El uso de LBYL puede resultar ineficiente, sobre todo si la comprobación es cara. Este enfoque evita excepciones pero continuamente está verificando que se puede hacer lo que se quiere.

---

## 7 Testea tu Código



Los *tests* son código que verifica que otro código funciona correctamente y permiten:

- ✓ Comprobar que nuestro código funciona como esperamos.  
Verifica que funciona hoy.
- 🛡️ Proteger el código de modificaciones futuras. Verifica que mañana sigue funcionando.

Escribir *tests* consiste en:

- ✉️ Probar que dadas unas entradas conocidas, las salidas son las esperadas.
- ❓ Probar que entradas inesperadas dan salidas esperadas. Como pedir `\%&?` cervezas en un bar. El camarero no debería *crashear*. Tu programa tampoco.

Veamos como usar `pytest` para proteger tu código de *bugs*.

---

## 7.1 Introducción al Testing

El fin de los *test* es detectar *bugs*. Estos se definen como errores o defectos donde el comportamiento del código no es el esperado. Un *bug* se traduce como bicho al español.

Algunos dicen que el origen de la palabra *bug* viene de literalmente una polilla atrapada en un equipo informático del año 1947 que causaba un funcionamiento incorrecto.

Imagina que tienes una función `suma`. Cualquier código por sencillo que sea debería ir acompañado de *tests*. Estos permiten verificar que el comportamiento es el adecuado.

```
# suma.py
def suma(a, b):
    return a + b

def test_suma():
    assert suma(1, 2) == 3
    assert suma(5, 5) == 10
```

Como puedes ver `test_suma` verifica que:

- Si la entrada es `1` y `2` la salida es `3`.
- Si la entrada es `5` y `5` la salida es `10`.

Con esto ya tendríamos nuestro primer *unit test*. Ahora toca ejecutarlos. Antes de nada debes instalar `pytest`.

```
pip install pytest
```

Con `pytest` podemos ejecutar los test. El `PASSED` nos dice que todos los `assert` han sido correctos.

```
pytest -v suma.py
# suma.py::test_suma PASSED
```

---

Es importante llevar los *test* al límite buscando entradas no esperadas. Podemos probar que pasa si sumamos `1` y `"2"`. Aquí ya depende del comportamiento que deseas. En nuestro caso verificamos que se produce una excepción, ya que estos dos valores no deberían poder sumarse.

```
import pytest

def test_suma_exception():
    with pytest.raises(TypeError):
        suma(1, "2")
```

También podemos probar que pasa si sumamos `\%` y `?`. Tal vez esto debería dar un error, ya que no tiene sentido sumar estos dos valores. Sin embargo nuestra función los suma, devolviendo `\%?`.

```
import pytest

def test_suma_exception():
    with pytest.raises(TypeError):
        suma("%", "?")
```

Por norma general, cuantos más tests, mejor. Pero es importante entender que no garantizan nada. Sobre todo si son incompletos o no prueban suficientes combinaciones. Quédate con esta frase:

- ! Un test prueba la presencia de un *bug*, pero no garantiza su ausencia.

Es decir, si eres capaz de escribir un test que falla, has demostrado que algo no funciona. Pero si todos pasan, no hay ninguna garantía que el código está 100% libre de *bugs*. Cuanto mas y mejores sean los tests, más seguridad habrá.

Por otro lado, los tests son también código y suelen ir separados del código principal. Algunos consejos para organizar tus tests:

- 
- 📁 Crea una carpeta `tests` que almacene todos tus test.
  - 🖊 Si un módulo se llama `modulo.py` pon los test de ese módulo en `test_modulo.py`.

Pero antes de entrar en el mundo del *testing*, vamos a aclarar toda la jerga que existe alrededor. Algunos conceptos interesantes:

- 🐛 **Bug**: Es un defecto de nuestro código. Algo que no funciona como se espera. Es un pequeño bicho difícil de encontrar que hace que las cosas no salgan como uno espera. Los *tests* pretenden eliminarlos.
- 📋 **Test Case**: Es una *test* que verifica un determinado caso de uso. El ejemplo que vimos anteriormente de `test_suma`.
- 📂 **Test Suite**: Es un conjunto de *test cases*. Una agrupación para organizar mejor nuestros *tests*.
- 💣 **Flaky Test**: Es un test que a veces pasa y otras no, teniendo un comportamiento aleatorio. Es un test que no tiene un resultado fiable. Idealmente no deberíamos tener ninguno así, pero en la práctica a veces pasa.
- 📈 **Coverage**: Es una métrica que mide el porcentaje de nuestro código que tiene *tests*. Existen diferentes métricas. Nuestro objetivo es tener 100% en todas. Sin embargo esta métrica no garantiza nada.
- 💬 **Fuzzing**: Es una técnica de *testing* que consiste en proporcionar entradas aleatorias e inesperadas para detectar fallos. Igual que pedirle `\%&?` cervezas al camarero.
- 🎁 **Black Box**: Es un tipo de *testing* que consiste en ver el sistema a testear como una caja negra. Es decir, algo a lo que le damos una entrada, nos proporciona una salida, pero no tenemos ni idea de lo que hay dentro ni como funciona.
- 🏆 **Continuous Integration**: Es una práctica en desarrollo software donde cada cambio en el código es verificado ejecutando

---

todos los tests. Si los tests no pasan, no se permite incluir el cambio. Proporciona feedback inmediato, mejora la calidad del código y detecta los *bugs* temprano.

-  **Mock:** Se utiliza en *testing* para sustituir a un objeto por un *mock*. Por ejemplo si quieras testear las comunicaciones con un avión es posible que no dispongas de uno. Pero puedes crear un *mock* del avión, simulándolo. Te da igual que lo que haya al otro lado sea un avión real o no, siempre y cuando recibas la misma información.
-  **Test Vector:** Conjunto de entradas y salidas esperadas. Por ejemplo, para la función suma un *test vector* podría ser `5, 3, 8`. Es decir, si las entradas son `5` y `3` la salida esperada será `8`.

Por otro lado, el concepto de *test* es bastante genérico, ya que existen muchos tipos:

-  **Unitarios:** Se centran en testear unidades pequeñas de código como podrían ser funciones o clases.
-  **Integración:** Se centran en testear la integración de todos los componentes de un sistema.
-  **Rendimiento:** Se centra en testear la velocidad y eficiencia.

Dependiendo de la complejidad de tu programa, es posible que no necesites todos estos tipos. Pero cualquier programa por sencillo que sea debería tener al menos unitarios. No pienses que los *tests* son para desarrolladores avanzados. Literalmente en dos líneas puedes tener uno.

Escribir *tests* es un arte. Normalmente es mejor que la persona que escribe el código no los escriba, ya que estará sesgada. Es interesante tener a alguien externo con un punto de vista no influenciado.

Una persona que piensa *out of the box* suele ser buena escribiendo test.

---

Como hemos comentado, no basta con probar el *happy path* o camino esperado. Hay que buscarle las cosquillas al código buscando los casos raros donde podría fallar.

Una de las ventajas de escribir los *tests* como código es que se ejecutan automáticamente. Esto hace que sea muy fácil ejecutarlos para verificar que no se ha roto nada.

Idealmente tu código no debería admitir cambios hasta que no se haya verificado que todos los tests pasan correctamente. Hay herramientas como Jenkins o Github Actions que permiten esto. Hasta que todos los test no están en verde, el nuevo cambio no se puede añadir.

Un error común es pensar que los tests nos garantizan la ausencia de *bugs* en nuestro código. Pueden detectar la presencia, pero no la ausencia. Si no los escribes bien, pueden pasar, pero eso no implica nada.

Con esto, ya estamos en condiciones de empezar a escribir nuestros primeros test con `pytest`.

## 7.2 Testing con `pytest`

El paquete `pytest` permite escribir tests para tu código y ejecutarlos de manera muy sencilla. Tiene todo lo que necesitas y es un estándar *de facto* en la industria. Si alguna vez has pensado que no tienes tiempo para escribir tests, con `pytest` ya no hay excusas. Escribe tests. Siempre.

Pero antes necesitamos tener algo que testear. Todo parte de un código con unos requisitos y una lógica esperada. Una vez lo tenemos, los tests simplemente se aseguran de que estos requisitos se cumplen siempre.

---

Para nuestro ejemplo, vamos a escribir un código que detecte si la Estación Espacial Internacional o International Space Station (ISS) está volando por encima de nosotros. Despues, escribiremos tests para asegurarnos de que funciona correctamente.

Los requisitos de nuestro proyecto son:

- 🛸 La posición de la ISS se obtiene de la API <http://api.opennotify.org/iss-now.json>.
- 🗺 Nuestra posición en la tierra con longitud y latitud es pasada en el constructor.
- 📈 Sabiendo nuestras coordenadas y las de la ISS, determinamos si está pasando encima de nosotros.

Como puedes comprobar en tu navegador, la API devuelve el siguiente *json*. Incluye la posición de la ISS, la hora y si la petición fue exitosa.

```
{  
    "message": "success",  
    "iss_position": {  
        "longitude": "31.9896",  
        "latitude": "-26.6497"  
    },  
    "timestamp": 1732180867  
}
```

Como te puedes imaginar el uso de esta API requiere acceso a Internet. Veremos más adelante como hacer que nuestros tests no dependan de servicios externos.

Por otro lado, es importante entender que la órbita de la ISS es baja. Unos 418 km. Esto significa que da vueltas a 418 km sobre nosotros. Sin embargo gira más rápido que la tierra, dando una vuelta cada 90 minutos.

Con esta información estamos ya listos para escribir nuestro código.

---

Definimos una clase `DistanciaISS` .

```
# iss.py
from geopy.distance import geodesic
from math import sqrt
import requests

class DistanciaISS:
    URL = "http://api.open-notify.org/iss-now.json"
    T = 0.01

    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def coordenadas_iss(self):
        try:
            response = requests.get(self.URL)
            response.raise_for_status()
            iss_data = response.json()
            iss_lat = float(iss_data['iss_position']['latitude'])
            iss_lon = float(iss_data['iss_position']['longitude'])
            return iss_lat, iss_lon
        except (requests.RequestException, ValueError,
                KeyError) as e:
            raise ValueError(f"Error al obtener posición ISS: {str(e)}")

    def encima(self):
        iss_lat, iss_lon = self.coordenadas_iss()
        lat_diff = abs(iss_lat - self.lat)
        lon_diff = abs(iss_lon - self.lon)
        return lat_diff <= self.T and lon_diff <= self.T
```

Tenemos los siguientes métodos:

- `__init__` : Constructor de nuestra clase al que le pasamos nuestras coordenadas geográficas de latitud y longitud.

- 
- `coordenadas_iss` : Usando la API mencionada anteriormente devuelve la posición actual de la Estación Espacial Internacional en latitud y longitud.
  - `encima` : Devuelve `True` si la ISS está encima nuestra, con una tolerancia de `T`.

Y estamos listos para usarlo. Indicamos que nuestras coordenadas son las siguientes, que corresponden a la ciudad de Buenos Aires. Si devuelve `True`, significa que la ISS está pasando por encima de Buenos Aires.

```
iss = DistanciaISS(-34.6037, -58.3816)
print(iss.encima())
# False
```

Ahora que tenemos el código, veamos como escribir tests para asegurarnos de que funciona correctamente y protegerlo de futuras modificaciones. Empecemos con un test sencillo. Es común escribir uno o varios tests por cada función o método.

Empezamos escribiendo un test para `__init__`. Verificamos que si creamos nuestro objeto con una latitud `lat` y longitud `lon` determinadas, los valores son guardados correctamente.

```
# iss_test.py
import pytest
from distancia_iss import DistanciaISS

def test_init():
    distancia_iss = DistanciaISS(10, 20)
    assert distancia_iss.lat == 10
    assert distancia_iss.lon == 20
```

Para ejecutar el test, puedes usar el siguiente comando en el terminal.

```
pytest -v
```

---

---

Y obtendrás el siguiente informe. Esto nos dice que nuestro test ha pasado. Esto significa que todos los `assert` cumplen la condición esperada.

```
iss_test.py::test_init PASSED [100%]
```

El comando anterior busca automáticamente todos los tests de la carpeta. Si quieres ejecutar los tests de un fichero puedes hacer lo siguiente.

```
pytest -v iss_test.py
```

Y si quieres ejecutar un test en concreto.

```
pytest -v iss_test.py::test_init
```

Llegados a este punto, enhorabuena. Tienes tu primer test y puedes ejecutarlo. Ahora cada vez que realices un cambio puedes ejecutarlo para ver que no has roto nada. Pero tenemos más cosas que testear.

Ahora vamos a escribir un test para `coordenadas_iss`. Aquí nos interesa ver que extraemos correctamente la latitud y longitud de la respuesta de la API. Sin embargo, se nos presenta un problema. Existe una dependencia externa: la API. Esta requiere acceso a Internet.

Una buena práctica en estas ocasiones es utilizar un *mock*. Este nos permite simular la API. Es decir, nuestro test podrá funcionar sin acceso a Internet.

Es importante saber delimitar bien estas líneas. Donde empieza y donde acaba lo que queremos testear. En el caso de *unit tests*, lo que queremos probar es nuestro código. Imagina que la API deja de funcionar. En este caso, nuestro test no funcionaría. Pero no es justo, ya que el código en realidad funciona bien.

---

Podemos hacer un *mock* de la API de la siguiente forma. Simplemente le estamos diciendo a Python que cuando se llame a la función `requests.get` del módulo `iss`, en vez de llamar al código normal que accede a Internet, devuelve el siguiente valor. En `raise_for_status` indicamos `None` para decir que no ha habido ningún problema.

```
# iss_test.py
import pytest
from unittest.mock import patch
from iss import DistanciaISS

def test_coordenadas_iss():
    with patch('iss.requests.get') as mock_get:
        mock_get.return_value.json.return_value = {
            'iss_position': {
                'latitude': '-50.0',
                'longitude': '-30.1',
            },
            'message': 'success',
            'timestamp': 1596563200
        }
        mock_get.return_value.raise_for_status =
            lambda: None

        distancia_iss = DistanciaISS(0, 0)
        assert distancia_iss.coordenadas_iss() ==
            (-50.0, -30.1)
```

El test anterior verifica que ante la respuesta que hemos usado como *mock*, se extrae la longitud y latitud correctamente. Pero no podemos fiarnos de nadie. Hay que ver que pasa cuando el resultado no es el esperado.

Imagina que la API es modificada y por error nos devuelve unos parámetros de longitud y latitud incorrectos. Por ejemplo en vez de un número, nos devuelve `texto`. Un test interesante es ver que se lanza una excepción.

---

Puede parecer evidente, pero imagina que el código procesa `texto` y devuelve un par de coordenadas válido. Esto sería peligroso.

```
def test_coordenadas_iss_no_numericos():
    with patch('iss.requests.get') as mock_get:
        mock_get.return_value.json.return_value = {
            'iss_position': {
                # Verificamos que esta entrada ...
                'latitude': 'texto',
                'longitude': 'texto',
            },
            'message': 'success',
            'timestamp': 1596563200
        }
        mock_get.return_value.raise_for_status =
            lambda: None

        distancia_iss = DistanciaISS(0, 0)
        # ... da lugar a una excepción
        with pytest.raises(Exception, match="Error"):
            distancia_iss.coordenadas_iss()
```

Ahora imagina que la API deja de funcionar o está *offline*. Con nuestro `mock` podemos simular esto. Simplemente usamos `raise_for_status` con una excepción. Verificamos que `coordenadas_iss` propaga la excepción y no devuelve coordenadas.

```
def test_coordenadas_iss_error():
    with patch('iss.requests.get') as mock_get:
        mock_get.return_value.json.return_value = {}
        # Forzamos un error en el get ...
        mock_get.return_value.raise_for_status.
            side_effect = Exception("Error")

        distancia_iss = DistanciaISS(0, 0)
        # ... y verificamos que da una excepción
        with pytest.raises(Exception, match="Error"):
            distancia_iss.coordenadas_iss()
```

---

Continuemos escribiendo tests para `encima`. Esta función llama en su interior a `coordenadas_iss`, y como esta a su vez usa la API que requiere Internet, vamos a hacer un *mock*.

Con este *mock* hacemos que cuando se llame a `coordenadas_iss` se devuelvan las coordenadas de Madrid.

```
def test_encima():
    with patch.object(DistanciaISS, 'coordenadas_iss',
                      return_value=(40.4167, -3.7033)):
        iss = DistanciaISS(40.4167, -3.7033)
        assert iss.encima() == True
```

Llegados a este punto, tenemos unos pocos test que verifican que nuestro código funciona correctamente. Hemos cubierto un poco de todo. Hemos cubierto el camino esperado, pero también posibles problemas. Hemos comprobado como en todos los casos el comportamiento es el deseado.

### 7.3 Múltiples tests con parametrize

Aunque anteriormente hemos escrito tests para todas las funciones, no son muy exhaustivos. Hay que probar con más combinaciones de valores.

Podemos escribir un test que pruebe con estas coordenadas.

```
def test_encima_1():
    with patch.object(DistanciaISS, 'coordenadas_iss',
                      return_value=(1, 1)):
        distancia_iss = DistanciaISS(1.001, 1.001)
        assert distancia_iss.encima() == True
```

Y otro test con coordenadas distintas.

```
def test_encima_2():
```

```
with patch.object(DistanciaISS, 'coordenadas_iss',
    return_value=(1, 1)):
    distancia_iss = DistanciaISS(0.999, 0.999)
    assert distancia_iss.encima() == True
```

Pero si te fijas, en ambos ejemplos tenemos un montón de código duplicado. Solo cambia una línea, pero repetimos 4.

Precisamente para esto `pytest` nos ofrece `parametrize`. Permite probar diferentes combinaciones sin tener que repetir tanto código. Esto nos sirve para introducir el concepto de *test vectors*. Un *test vector* tiene dos partes:

- **Entrada**: Los argumentos que se pasan a la función. En nuestro caso las coordenadas de nuestra posición y las de la ISS.
- **Salida esperada**: El resultado esperado. En este caso la distancia esperada entre nuestra posición y la ISS.

Los *test vectors* permiten continuamente verificar que ante unas entradas determinadas, las salidas son las esperadas. Puedes ver en `parametrize` nuestro test vector. Simplemente un montón de combinaciones.

```
@pytest.mark.parametrize("lat, lon, iss_lat, iss_lon,
    encima", [
    (40.4, -3.7, 40.4, -3.702, True),
    (34.0, -118.2, 34.001, -118.2, True),
    (51.5, -0.1, 0.4, -3.3, False)
])
def test_distancia_iss(lat, lon, iss_lat, iss_lon,
    encima):
    with patch.object(DistanciaISS, 'coordenadas_iss',
        return_value=(iss_lat, iss_lon)):
        distancia_iss = DistanciaISS(lat, lon)
        assert distancia_iss.encima() == encima
```

Si lo ejecutas verás como se han generado múltiples tests, uno para cada entrada.

```
iss_test.py::test_distancia_iss [40.4--3.7-40.4--3.702-  
    True]  
iss_test.py::test_distancia_iss  
    [34.0--118.2-34.001--118.2-True]  
iss_test.py::test_distancia_iss [51.5--0.1-0.4--3.3-  
    False]
```

## 7.4 Usando fixture en pytest

Imagina que tienes múltiples test que usan los mismos datos. En este caso ambos tests usan la misma lista.

```
def test_sum():  
    assert sum([1, 2, 3, 4, 5]) == 15  
  
def test_length():  
    assert len([1, 2, 3, 4, 5]) == 5
```

Sin embargo esto da lugar a código duplicado. Si en vez de 2 funciones tienes 10, aún más. Y si en algún momento quieres cambiar los datos, tendrías que modificarlo en todos los sitios.

Para resolver este problema puedes crear una variable `datos` y usarla en todos los test. Esta solución es válida y en casos sencillos puede ser suficiente.

```
datos = [1, 2, 3, 4, 5]  
  
def test_sum():  
    assert sum(datos) == 15  
  
def test_length():  
    assert len(datos) == 5
```

Sin embargo `pytest` ofrece `fixture` para hacer algo similar, pero con un mayor control. El siguiente código tiene el mismo compor-

---

tamiento. Pero como puedes ver, se puede pasar el `fixture` a cada test de manera individual.

```
import pytest

@pytest.fixture
def datos():
    return [1, 2, 3, 4, 5]

def test_sum(datos):
    assert sum(datos) == 15

def test_length(datos):
    assert len(datos) == 5
```

Una propiedad interesante es que `pytest` se encarga de mantener el valor del `fixture`. Es decir, si algún test cambia o elimina `datos` no pasa nada. Cada test accede a una copia independiente. Esta es una de las diferencias de la forma anterior.

```
@pytest.fixture
def datos():
    return [1, 2, 3, 4, 5]

def test_sum(datos):
    assert sum(datos) == 15
    # Modificamos datos
    datos.append(6)

def test_length(datos):
    # Pero datos aquí no cambia
    assert len(datos) == 5
```

Volviendo a nuestro ejemplo de la ISS, podemos usar el `fixture` de la `respuesta` de la siguiente manera.

```
@pytest.fixture
def respuesta():
    return {
```

```

        'iss_position': {'latitude': 0, 'longitude': 0},
        'message': 'success',
        'timestamp': 1596563200
    }

@pytest.fixture
def iss():
    return DistanciaISS(0, 0)

def test_coordenadas_iss_no_numericos(respuesta, iss):
    with patch('iss.requests.get') as mock_get:
        mock_get.return_value.json.return_value =
            respuesta
        mock_get.return_value.raise_for_status =
            lambda: None

    assert iss.encima() == True

```

Por último, es posible configurar el `scope` de un `fixture`. Se hace de la siguiente manera.

```

@pytest.fixture(scope="session")
def datos():
    return [1, 2, 3, 4, 5]

def test_sum(datos):
    assert sum(datos) == 15
    datos.append(6)

def test_length(datos):
    # Este test falla, datos ahora tiene len 6.
    assert len(datos) == 5

```

El `scope` nos permite decir cuando sea crea de nuevo. Por defecto se usa `function` lo que significa que cada función recibe el `fixture` nuevo.

Pero existen otros como `sesion`. Este crea un único `fixture` para toda la sesión. Por tanto si lo modificamos en un test, afectará al

---

resto.

## 7.5 Otras funcionalidades de pytest

Otra ventaja de `pytest` es que tiene una amplia variedad de plugins para extender su funcionalidad. Algunos vienen por defecto y otros hay que instalarlos.

-  `skip`
-  `timeout`
-  `flaky`
-  `freeze time`
-  `benchmark`

Para instalar todos.

```
pip install pytest-timeout  
pip install pytest-rerunfailures  
pip install pytest-freezegun  
pip install pytest-benchmark
```

 `skip` . Para saltar un test y que no se ejecute. Pero recuerda, si el test falla la solución no es saltarlo. Es entender el problema y arreglarlo.

```
@pytest.mark.skip(reason="No se ejecuta")  
def test_que_no_se_ejecuta():  
    pass
```

Al ejecutar los tests, verás los que se saltan y su razón.

```
iss_test.py::test_que_no_se_ejecuta SKIPPED (No se  
ejecuta)
```

 `timeout` . Para establecer un tiempo máximo para tus tests. Si pasado ese tiempo no ha acabado, se considerará fallido.

---

En este ejemplo el test debería pasar, pero como tarda **2** segundos y el *timeout* es de **1** segundo, falla.

```
import time
@pytest.mark.timeout(1)
def test_con_timeout():
    time.sleep(2)
    assert 1 == 1
```

Si lo ejecutas obtendrás lo siguiente.

```
iss_test.py::test_con_timeout FAILED
```

⚠ **flaky** . Permite marcar un test como *flaky* o inestable. Es decir, que falla a veces. Puedes especificar el número máximo de veces que se intente otra vez. Si después de todas esas veces no ha pasado con éxito, se considerará fallido.

Los tests deben ser deterministas en la medida de lo posible y no debería haber tests *flaky*, pero a veces es inevitable.

```
import random

@pytest.mark.flaky(reruns=10)
def test_flaky():
    dado = random.choice([1, 2, 3, 4, 5, 6])
    assert dado == 1
```

Al ejecutarlo verás lo siguiente. El test se ejecuta múltiples veces, hasta un máximo de **10** . Si al menos una pasa, se considera pasado.

```
iss_test.py::test_flaky RERUN
iss_test.py::test_flaky RERUN
iss_test.py::test_flaky PASSED
1 passed, 2 rerun in 0.09s
```

⌚ **freeze\_time** . Permite congelar el instante temporal. Es decir, puedes establecer la fecha actual. Es similar a hacer un *mock* como

---

hemos visto anteriormente.

```
@pytest.mark.freeze_time('2023-01-01')
def test_fecha_actual():
    assert date.today() == date(2023, 1, 1)
```



`benchmark` . Permite medir el tiempo de ejecución de tu test.

```
def tarea_pesada():
    return sum(i * i for i in range(100000000))

@pytest.mark.benchmark()
def test_tarea_pesada_benchmark(benchmark):
    result = benchmark(tarea_pesada)
    assert result == 3333332833333350000000
```

Al ejecutarlo se mostrará el tiempo que tardó. El test se ejecuta múltiples veces y se realiza una media del tiempo. En este caso tarda unos `4.4` segundos.

Name (time in s)	Min	Max	Mean
<hr/>			
test_tarea_pesada_benchmark	4.48	4.61	4.54

También puedes poner una condición con un `assert` done el test solo pasará si tarda menos de un tiempo determinado. En este ejemplo el test falla si tarda más de `5` segundos.

```
@pytest.mark.benchmark()
def test_tarea_pesada_benchmark(benchmark):
    result = benchmark(tarea_pesada)
    assert result == 3333332833333350000000
    assert benchmark.stats['mean'] < 5
```

De esta manera nos protegemos de que en un futuro alguien modifique el código y lo haga ineficiente.

---

## 7.6 Coverage con pytest-cov

Una métrica muy importante en el mundo del testing es el *code coverage* o cobertura de código. Esta métrica nos dice que porcentaje de nuestro código está cubierta por al menos un test. Nuestro objetivo es tener 100% de *code coverage*.

Definimos una función `opera` que toma dos parámetros `a` y `b` y dependiendo de la `operacion` realiza una suma o una resta.

```
# ejemplo.py
def opera(a, b, operacion):
    if operacion == "suma":
        return a + b
    elif operacion == "resta":
        return a - b
    else:
        raise ValueError(f"Operación no válida: {operacion}")
```

Creamos un fichero con un test.

```
# ejemplo_test.py
import pytest
from ejemplo import opera

def test_opera():
    assert opera(2, 3, "suma") == 5
```

Instalamos el paquete `pytest-cov`.

```
pip install pytest-cov
```

Si ahora ejecutar los test con `--cov`, `pytest` te dará un reporte de *coverage*. En este caso vemos que tenemos un 50%.

```
pytest --cov=ejemplo
# Name          Stmts   Miss  Cover
# -----
```

```
# ejemplo.py       6      3      50%
```

Es del 50%, ya que únicamente estamos probando la operación `suma` y no estamos probando otros caminos. Es un test incompleto. Para tener el 100% necesitaríamos probar todos los caminos:

- $+$  La suma.
- $-$  La resta.
- $!$  Un operador no reconocido.

El siguiente test prueba los tres caminos.

```
def test_opera():
    assert opera(2, 3, "suma") == 5
    assert opera(2, 3, "resta") == -1
    with pytest.raises(ValueError):
        opera(2, 3, "multiplicacion")
```

Por lo que ya tenemos un *coverage* del 100%.

```
pytest --cov=ejemplo
# Name          Stmts   Miss  Cover
# -----
# ejemplo.py      6       0    100%
# -----
# TOTAL          6       0    100%
```

Aunque un *coverage* del 100% es lo que debemos buscar, esto no ofrece ninguna garantía. Un código con *coverage* del 100% puede tener un *bug*. Recuerda que un test muestra la presencia de un *bug* pero no su ausencia.

Vamos a demostrarlo. Imagina que hemos cometido un error. Y en la `operacion` resta usamos un  $+$  en vez de  $-$  como debe ser. Este código estaría mal.

```
def opera(a, b, operacion):
    if operacion == "suma":
```

```
        return a + b
    elif operacion == "resta":
        return a + b # <- Cometemos un error. MAL!
    else:
        raise ValueError(f"Operación no válida: {operacion}")
```

Ahora en nuestro test, no elegimos los valores adecuados. Si restamos con `0` y `0` el resultado es `0`. Hemos llegado al resultado correcto pero con un código que está mal.

```
def test_opera():
    assert opera(2, 3, "suma") == 5

    # Este assert pasa. Pero tenemos un bug.
    assert opera(0, 0, "resta") == 0
```

Este test pasa. Y el test *coverage* es del 100%. Esto nos podría dar la confianza de que el código está bien. Pero no.

El *bug* que hemos introducido se podría detectar con el siguiente código. Es el contraejemplo que demuestra que algo no está bien.

```
def test_opera():
    assert opera(5, 1, "resta") == 4
```

Por tanto es importante probar tantas combinaciones como sea posible y probar de todo. Números positivos, negativos, cero, decimales e incluso entradas inesperadas.

Por otro lado, si quieres excluir alguna función de la métrica de *coverage* lo puedes hacer con el comentario `## pragma: no cover`. Python no lo tendrá en cuenta para calcular el porcentaje.

```
def opera(a, b, operacion): # pragma: no cover
    # ...
```

Y como último apunte, `pytest` genera reportes de *coverage* muy

---

interesantes. Cuando tienes cientos de funciones, es interesante verlo en un interfaz más agradable.

```
pytest --cov=ejemplo --cov-report=html
```

Coverage report: 100%					
	File	Functions	Classes		
coverage.py v7.6.7					
File ▼	function	statements	missing	excluded	coverage
ejemplo.py	opera	5	0	0	100%
ejemplo.py	(no function)	1	0	0	100%
<b>Total</b>		<b>6</b>	<b>0</b>	<b>0</b>	<b>100%</b>

## 7.7 Fuzzing con hypothesis

El *fuzzing* consiste en la generación de una gran cantidad de entradas aleatorias con valores de todo tipo. Su objetivo es buscar un caso donde el comportamiento no sea el esperado. A diferencia de los test que hemos visto hasta ahora, con *fuzzing* la generación es automática.

El hecho de que sea automática es muy ventajoso, ya que se pueden generar miles de combinaciones. Escribir esto a mano sería muy tedioso.

El *fuzzing* nos permite detectar:

- ⚡ Posibles *edge cases* o casos límite que no habíamos considerado.
- ⚡ Problemas de *performance*. Tal vez una entrada determinada tarda demasiado tiempo.

- 
- 🔒 Inconsistencias. A veces conocido como *invariants* e idempotencia. Por ejemplo si ordenas una `list` y la vuelves a ordenar el resultado debe ser el mismo. O si inviertes el orden de una lista dos veces el resultado debe ser igual al inicial.

A continuación veremos como usar el paquete `hypothesis` para hacer esto. Primero, instala el paquete.

```
pip install hypothesis
```

De la siguiente manera puedes generar un ejemplo de un `int`. Se trata de un `int` aleatorio.

```
import hypothesis.strategies as st
print(st.integers().example())
# -14251
```

También podemos añadir restricciones. Por ejemplo entre `0` y `100`.

```
print(st.integers(min_value=0, max_value=100).example()
      ())
# 95
```

También podemos generar texto en `str`. Como puedes ver se incluyen letras de todo tipo.

```
import hypothesis.strategies as st
print(st.text().example())
# È a úæ4c
```

Si quieres múltiples ejemplos, puedes hacer lo siguiente.

```
for _ in range(5):
    print(st.integers().example())
```

Y esto no es todo. Con `hypothesis` podemos generar prácticamente de todo:

- 
- `integers` : Números enteros o `int` como hemos visto.
  - `text` : Texto o `str` como hemos visto.
  - `floats` : Números `float`.
  - `booleans` : Tipos `bool`.
  - `lists` : Listas o `list`.
  - `tuples` : Tuplas o `tuple`.
  - `dictionaries` : Diccionarios o `dict`.
  - `sets` : Sets o `set`.

Pero vamos a ver lo que realmente nos importa. Esto es la integración de `hypothesis` con `pytest`.

Veamos un ejemplo. Definimos una función que invierte el orden de un `str`. Es decir, pasa de `abc` a `cba`.

```
def invertir(s: str) -> str:  
    return s[::-1]
```

Un buen ejemplo de *fuzzing* es ver que si usamos `invertir` dos veces, el resultado es igual al inicial.

En el siguiente ejemplo tenemos un test que genera `10000` ejemplos de `str` totalmente aleatorios, verificando que la propiedad se cumple.

```
from hypothesis import given, settings  
from hypothesis import strategies  
  
@settings(max_examples=10000)  
@given(strategies.text())  
def test_invertir(s):  
    assert invertir(invertir(s)) == s
```

Lo podemos ejecutar de la siguiente manera y nos da unas estadísticas del número de ejemplos que se han probado. En nuestro caso todo ha funcionado correctamente.

---

```
pytest -v --hypothesis-show-statistics
# 10000 passing examples
# 0 failing examples
# 0 invalid examples
```

En resumen:

- Escribe tests siempre. Aunque tu código sea sencillo. Te ayudará a comprobar que funciona y a protegerlo de futuras modificaciones.
- Incluye tests que usen entradas inesperadas, para verificar que tu código es resistente a todo.
- Usa métricas de *coverage* como ayuda, pero recuerda que un 100% de *coverage* no da ninguna garantía.

---

## 8 Top 50 Ejemplos Prácticos



Ahora que ya dominas Python veamos unos ejemplos prácticos en diferentes sectores:

- 🤖 Inteligencia artificial, redes neuronales, genética y ADN.
- 📊 Dashboards y representación gráfica.
- 💰 Análisis financiero, apuestas, estadística y matemáticas.
- 🔒 Encriptado, firma digital, computación cuántica, factorización de números primos y fuerza bruta.
- 🎮 Desarrollo de juegos, webs, APIs e interfaces de usuario.
- 💾 Implementación de algoritmos sencillos.

Estos ejemplos dan ideas de lo que se puede hacer con Python, dejando al lector unos ejercicios para que explore más allá.

Aquí puedes encontrar los códigos completos:

- ⚡ [github.com/ellibrodepython/50-ejemplos-python](https://github.com/ellibrodepython/50-ejemplos-python)

---

## 8.1 Firma digital con cryptography

Veamos como proteger la información para asegurarnos de que el autor es quien dice ser. Antiguamente se usaban las firmas con pluma. Cada persona tenía una firma reconocible y difícil de copiar.

Actualmente tenemos la firma digital donde la integridad de la información está asegurada por la criptografía. La firma digital es un sistema criptográfico donde existen dos claves relacionadas matemáticamente:

- Una pública  accesible por cualquiera. Permite verificar una firma.
- Otra privada  que debe mantenerse en secreto. Permite crear una firma.

Con la firma digital podemos verificar el origen o integridad de un mensaje. Si firmas un mensaje con tu clave privada, cualquiera podrá verificar su autenticidad con tu clave pública.

Veamos un ejemplo donde firmas un mensaje con tu clave privada. La clave privada es el equivalente a la pluma y a tu firma, algo que solo tú puedes crear. Empezamos importando los módulos que necesitamos.

```
from cryptography.hazmat.primitives.asymmetric import
    ec
from cryptography.hazmat.primitives import
    serialization
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature
```

Podemos crear una clave privada de la siguiente manera. La clave pública deriva de la privada.

```
privada = ec.generate_private_key(ec.SECP256R1())
```

```
publica = privada.public_key()
```

Puedes ver la privada así. En aplicaciones reales no debes enseñar esta clave a nadie, o se podrían hacer pasar por ti.

```
pem = privada.private_bytes(  
    serialization.Encoding.PEM,  
    serialization.PrivateFormat.PKCS8,  
    serialization.NoEncryption())
```

Podemos firmar un mensaje utilizando `sign`. Esto es algo muy usado en transacciones blockchain. Esta firma permite reconocer que el creador de ese mensaje es quien tiene la clave privada. Haciendo esto es como si autorizaras esta transacción.

```
firma = privada.sign(  
    b"Envia 1 Euro al IBAN 123",  
    ec.ECDSA(hashes.SHA256()))
```

Ahora, un tercero podría verificar que el mensaje fue efectivamente creado por el conocedor de `privada`. Para esto sólo se necesita `publica`. Podemos verificar que el mensaje no ha sido alterado.

```
try:  
    publica.verify(  
        firma,  
        b"Envia 1 Euro al IBAN 123",  
        ec.ECDSA(hashes.SHA256()))  
    print("Verificacion OK")  
except InvalidSignature:  
    print("Verificacion NOK")  
# Verificacion OK
```

Ahora pongamos que alguien intercepta la comunicación y realiza lo siguiente:

- 🐺 Cambia el mensaje de `1 euro`.
- 💰 Cambia la cantidad por `100 euros`.

- 
-  Cambia la cuenta por `456` .

Si ahora intentamos verificar la firma con `verify` obtendremos un error `InvalidSignature` . Nos hemos protegido de un atacante modificando el mensaje.

```
try:  
    publica.verify(  
        firma,  
        b"Envia 100 Euros al IBAN 456",  
        ec.ECDSA(hashes.SHA256()))  
    print("Verificacion OK")  
except InvalidSignature:  
    print("Verificacion NOK")  
# Verificacion NOK
```

En pocas palabras, la firma digital permite evitar que alguien suplante tu identidad y diga cosas que tú no has dicho. Eso sí, siempre y cuando la clave privada solo la conozcas tú.

### Ejercicios:

- Intenta usar otra clave pública distinta para verificar la firma con `verify` y explica lo que sucede.

## 8.2 Encriptar mensaje con cryptography

Veamos como encriptar un mensaje para que solo el destinatario autorizado pueda desencriptarlo y verlo. Existen dos formas de hacer esto.

-  Con criptografía simétrica. En este caso la clave para encriptar y desencriptar son la misma. Ya los Romanos usaban este tipo de criptografía.
-  Con criptografía asimétrica. Existen dos claves diferentes. Una para encriptar y otra para desencriptar.

---

A continuación veremos como realizarlo con criptografía asimétrica. Como hemos indicado, existen dos claves:

- Una clave pública 🔑 que cualquier puede conocer. Se usa para encriptar.
- Una clave privada 🗝 que debe mantenerse en secreto. Se usa para desencriptar.

Importamos lo necesario.

```
from cryptography.hazmat.primitives.asymmetric import
    rsa, padding
from cryptography.hazmat.primitives import hashes
```

Generamos una clave `privada`. La `publica` se deriva de la privada.

```
privada = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048)
publica = clave_privada.public_key()
```

Ahora encriptamos un mensaje. El `mensaje_encriptado` solo lo podrá desencriptar quien tenga la clave privada asociada.

```
mensaje = b"Mensaje secreto de El Libro de Python"
encriptado = publica.encrypt(
    mensaje,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None))
```

El conocedor de la clave privada, puede desencriptar el mensaje.

```
desencriptado = privada.decrypt(
    encriptado,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
```

```
algorithm=hashes.SHA256(),
label=None)

print(mensaje_des)
# b'Mensaje secreto de El Libro de Python'
```

Gran parte de la criptografía actual funciona gracias a lo difícil que es factorizar números en sus primos. Técnicamente alguien podría desencriptar tu mensaje sin la clave privada, probando múltiples combinaciones. Sin embargo esto no es posible en la práctica, ya que se tardaría un tiempo infinito. A nivel práctico, lo consideramos imposible.

Cierto es que los ordenadores cuánticos pueden romper los esquemas tradicionales de encriptación, pero la criptografía post-cuántica resuelve esto. Aún quedan unos años para que esto sea una realidad.

### Ejercicios:

- Intenta desencriptar el mensaje con otra clave privada distinta y explica lo que sucede.

## 8.3 Determina si un número es primo

Veamos como determinar si un número es primo. Los números primos tienen propiedades matemáticas muy interesantes. Se dice que un número es primo si solo es divisible (sin resto) por uno y por sí mismo.

-  El 7 es primo. Solo es divisible entre 1 y 7.
-  El 8 no es primo. Es divisible entre 1, 2, 4 y 8.

Podemos determinar si un número es primo de la siguiente manera. Miramos si el número es divisible por otros números saltando el 1 y sí mismo. Si detectamos que lo es, decimos que no es primo.

```
def es_primo(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Aunque la anterior función es correcta existe una optimización. No es necesario probar a dividir todos los números. Basta con probar hasta  $\sqrt{n} + 1$ . En otras palabras, no hace falta probar si 10, 11, 12 dividen a 13. Ya sabes que no porque son mayores que 4. Sabiendo esto podemos optimizar el código.

```
def es_primo(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Ahora podemos usar nuestra función. Verificamos que el 41 es primo.

```
q = 41
print(f"{q} es primo? {es_primo(q)}")
```

💡 Ejercicios:

- Escribe una función que busque un número primo muy grande, con 8 dígitos.

## 8.4 Factoriza un número en primos

Veamos como factorizar un número en sus primos. Es decir, cualquier número que se te ocurra puede ser expresado como la multiplicación de varios números primos.

- El 8 es 2\*2\*2 .
- El 765 es 3\*3\*5\*17 .

Este tipo de factorización es muy interesante porque permitiría romper los algoritmos de criptografía actuales. Sin embargo para números muy grandes es tan complicado encontrar la solución que en la práctica es imposible.

Podemos definir la función de la siguiente manera. Intenta dividir nuestro número n por todos los números posibles, empezando por el 2 . Si es divisible por divisor , lo almacenamos como factor y continuamos con el siguiente.

```
def factorizar(n):
    factores = []

    for divisor in range(2, int(n**0.5) + 1):
        while n % divisor == 0:
            factores.append(divisor)
            n //= divisor

    if n > 1:
        factores.append(n)
    return factores
```

Veamos un ejemplo factorizando 765 .

```
numero = 765
factores = factorizar(numero)
print(f"{numero} = {'*'.join(map(str, factores))}")
# 765 = 3*3*5*17
```

---

Ahora, un ejemplo con un número mayor. Como puedes ver, a medida que el número es más grande, más tiempo se tarda en factorizar.

```
numero = 10000004400000259
factores = factorizar(numero)
print(f"{numero} = {'*'.join(map(str, factores))}")
# 10000004400000259 = 10000007*100000037
```

### Ejercicios:

- Busca un número no sea posible factorizar porque tarde demasiado tiempo.

## 8.5 Computación cuántica con qiskit

Veamos como usar un ordenador cuántico para generar una clave privada. Generar una clave privada es simplemente crear un montón de bits aleatorios. El problema es que existen muchas formas de obtener estos bits. Es lo que se conoce como fuente de entropía.

Una fuente de entropía puede ser algo tan cotidiano como el camino recorrido por tu gato durante un día. O algo tan complejo como medir el estado de un *qubit* en un ordenador cuántico. Pero en ambos casos, lo que obtenemos es lo mismo. Un número aleatorio.

Para nuestro ejemplo usaremos un ordenador cuántico. Las propiedades de la mecánica cuántica nos permiten obtener un valor aleatorio cuando medimos el estado de un *qubit*.

Los ordenadores cuánticos usan *qubits* en vez de bits. Estos pueden estar en superposición, lo que significa que están a la vez en el estado  y  con una probabilidad determinada. Un *qubit* es algo así como una moneda tirada al aire:

-  Mientras está en el aire se puede decir que está en ambos estados. Puede ser cara o cruz con 50% de probabilidad.

- 
- 🚧 Cuando toca el suelo es cuando se revela su estado. Cara o cruz.

Estamos lejos de poder tener un ordenador cuántico en casa pero la librería `qiskit` nos permite simular uno. También nos permite interactuar con [IBM Quantum](#), donde podemos usar un computador cuántico real en el *cloud*.

Puedes obtener el *API token* en <https://quantum.ibm.com> con el que podrás usar varios minutos al mes el ordenador cuántico de IBM. Obtén tu token ya que será necesario para este ejemplo.

Empezamos importando todo lo que necesitamos.

```
from qiskit import QuantumRegister, ClassicalRegister,  
    QuantumCircuit  
from qiskit import transpile  
from qiskit_ibm_runtime import QiskitRuntimeService
```

Ahora definimos un circuito para el ordenador cuántico. No olvides poner en `token` tu *token* de [IBM Quantum](#). Esto es lo que hace el código:

- Indicamos que queremos `64 qubits` y `64 bits`. Idealmente necesitaríamos `256` pero no hay ningún ordenador con tantos *qubits*.
- La `h` indica que usamos una puerta *Hadamard*. Esta hace que la probabilidad de los *qubits* de estar en `0` o `1` es la misma.
- Con `measure` decimos que queremos medir el estado de los `64 qubits`.
- Realizamos el proceso `4` veces como indica `shots`. Esto es para generar una clave con `256 bits` ya que es `64*4`.

```
q = QuantumRegister(64)  
c = ClassicalRegister(64)  
circ = QuantumCircuit(q, c)
```

---

```

circ.h(q)
circ.measure(q, c)

service = QiskitRuntimeService(channel="ibm_quantum",
    token="TU_TOKEN")
backend = service.backend('ibm_kyiv')
qc_basis = transpile(circ, backend)
job = backend.run(qc_basis, shots=4)

counts = job.result().to_dict()["results"][0]["data
    "]["counts"]
priv_key = ''.join(counts.keys()).replace('0x', '')
print(f"Clave privada: {priv_key}")

```

Una vez realizado esto ya tienes tu clave privada generada. Esta es un número aleatorio de 256 bits cuya entropía viene de un ordenador cuántico real.

Es importante notar que no debes generar claves privadas así. Al menos por ahora. Aunque la entropía de la clave pueda ser buena, no tienes ninguna garantía de que la clave es privada.

Si tienes el ordenador cuántico en tu casa, la cosa cambia. Pero es probable que no sea el caso.

## 8.6 Fuerza bruta con `itertools`

Veamos como escribir una función que pueda romper contraseñas probando con fuerza bruta todas las combinaciones posibles.

En c definimos todas las posibles letras que usaremos. Usamos sólo minúsculas y números para que sea más sencillo de romper. Cuantas más letras posibles, como mayúsculas o caracteres raros como ?, , más complicada de romper será.

El algoritmo es muy sencillo y prueba combinaciones desde longitud 1 hasta long\_mag . Estos serían alguno de los ejemplos probados.

- Para longitud 1 . Se prueba: a , b , c , ... hasta 9
- Para longitud 2 . Se prueba: aa , ab , ac , ... 9a , 9b , ... hasta 99 .
- Para longitud 3 . Se prueba: aaa , aab , ... 9aa , 9aab , ... hasta 999 .

Como puedes ver, las combinaciones crecen de forma exponencial. Cuantos más letras posibles y mayor longitud, más difícil es de romper.

```
import itertools

def fuerza_bruta(contrasenia, long_max=6):
    c = "abcdefghijklmnopqrstuvwxyz0123456789"

    for longitud in range(1, long_max + 1):
        print(f"Longitud {longitud}...")
        for intento in itertools.product(c, repeat=longitud):
            intento_s = ''.join(intento)
            if intento_s == contrasenia:
                return intento_s
    return None
```

Si nuestra contraseña solo tiene letras y números y es de longitud 6, puedes ver de la siguiente manera cómo la podemos romper en unos pocos segundos.

```
contrasenia = "pass67"
encontrada = fuerza_bruta(contrasenia, long_max=8)
print(f"Contraseña encontrada: {encontrada}")
```

### Ejercicios:

- Calcula las combinaciones posibles para una contraseña con números y letras de longitud 10. Intenta romperla con fuerza bruta.

## 8.7 Acortador de enlaces con flask

Veamos como implementar un servicio para acortar enlaces, como el que puedes ver en redes sociales o webs. Acortar enlaces permite ahorrar caracteres en tus mensajes y permite pasar:

- ↗ De una URL larga  
`https://www.ejemplo.com/con/url/muy/larga/y/muchos/parametros`
- 📄 A una URL corta `http://servidor:5000/6231aa1` .

Para que esto funcione es necesario un servicio que recuerde la relación entre ambas URL. Es decir, que cuando alguien intente acceder a la corta, se redirija a la larga.

- El servidor almacena la relación 📄 → ↗.

En este ejemplo vamos a implementar un servicio web con `flask` que nos permita acortar URL. Tendremos dos *endpoints*:

- `acorta` : Dada una URL larga, nos devuelve una corta y almacena la relación.
- `/` : Dada una URL corta, usando la relación almacenada, nos redirige a la larga.

```
from flask import Flask, request, redirect
import hashlib

app = Flask(__name__)
url_map = {}

@app.route('/acorta', methods=['POST'])
def shorten_url():
    url_larga = request.form['url']
    url_corta = acorta_url(url_larga)
    url_map[url_corta] = url_larga
    return f'Acortada: {url_larga} -> {url_corta}'
```

```
@app.route('/<url_corta>')
def corta_a_larga(url_corta):
    url_larga = url_map.get(url_corta)
    if url_larga:
        return redirect(url_larga)
    else:
        return 'No encontrada!', 404

def acorta_url(full_url):
    hashed = hashlib.sha256(full_url.encode()).hexdigest()
    return hashed[:7]

if __name__ == '__main__':
    app.run(debug=True)
```

Veamos el ejemplo en acción. Si instalas `curl` puedes hacer lo siguiente. Esto realiza una petición a tu aplicación indicando que quieres acortar la dirección de Google. Y almacena la URL corta `253d142`.

```
curl -X POST -d "url=http://www.google.com" http://
localhost:5000/acorta
# Acortada: http://www.google.com -> 253d142
```

Ahora en tu navegador si pones `http://localhost:5000/253d142` podrás ver como te redirige a Google.

💡 Ejercicios:

- Usa el paquete `sqlite3` para persistir la información de `url_map` si la aplicación se reinicia. Es decir, tal como está ahora si se reinicia la aplicación se perderán las URL acortadas. Persiste esa información en disco y cárgala al iniciar.

## 8.8 Construye una API con flask

Veamos como implementar un servicio en forma de API que realice conversiones de moneda entre € y \$.

- Definimos un ratio de conversión EUR\_A\_USD . En un caso real será variable.
- Definimos una endpoint to-usd que convierte los € a \$.

```
from flask import Flask, jsonify

app = Flask(__name__)

EUR_A_USD = 1.1

@app.route('/to-usd/<eur>', methods=['GET'])
def convert(eur):
    try:
        eur = float(eur)
        usd = eur * EUR_A_USD
        return jsonify({
            'EUR': eur,
            'USD': usd
        })
    except ValueError as e:
        return jsonify({'error': str(e)}), 400

if __name__ == '__main__':
    app.run(debug=True)
```

Prueba en tu navegador lo siguiente:

- http://127.0.0.1:5000/to-usd/100 : Convierte 100€ a \$.
- http://127.0.0.1:5000/to-usd/noesnumero : Devuelve un error, ya que no es un número.

💡 Ejercicios:

- Implementa una función to-eur que convierta de € a \$.

- 
- Añade un parámetro para que la URL sea `/to-usd?eur=100` .
  - Añade un nuevo *endpoint* que permita actualizar el ratio de conversión `EUR_A_USD` . Decide si es mejor utilizar *PUT* o *POST*.

## 8.9 Trabaja con ficheros con os

En este ejemplo vemos como trabajar con ficheros usando `os` . Veamos como:

- Crear ficheros
- Modificar el nombre de ficheros
- Eliminar ficheros y carpetas

Veamos como crear varios ficheros. Los creamos vacíos, pero puedes usar cambiar el `write` para añadir contenido si lo deseas.

```
def crea_ficheros(ruta, nombre, cantidad):  
    if not os.path.exists(ruta):  
        os.makedirs(ruta)  
    for i in range(cantidad):  
        archivo_ruta = os.path.join(ruta, f"{nombre}_{i}.txt")  
        with open(archivo_ruta, 'w') as fichero:  
            fichero.write('')  
    print(f"Creado: {archivo_ruta}")
```

Si llamamos a la función, crearemos 10 ficheros en la carpeta `ejemplo` con nombres desde `fichero_1.txt` hasta `fichero_9.txt` .

```
crea_ficheros("./ejemplo", "fichero", 10)
```

Veamos como modificar el nombre de los ficheros. Esta función añade un `prefijo` al nombre de cada fichero en la `ruta` . Es decir:

- Renombra `fichero.txt` a `prefijo_fichero.txt` .

```
def renombra_ficheros(ruta, prefijo):
    for contenido in os.listdir(ruta):
        archivo_ruta = os.path.join(ruta, contenido)

        if os.path.isfile(archivo_ruta):
            nuevo_nombre = prefijo + contenido
            nueva_ruta = os.path.join(ruta,
                                      nuevo_nombre)

            os.rename(archivo_ruta, nueva_ruta)
            print(f"{contenido} -> {nuevo_nombre}")
```

Podemos usar la función así.

```
renombrar_ficheros("./ejemplo", "prefijo_")
```

Veamos como eliminar ficheros y carpetas. Esta función elimina todos los ficheros de `ruta`.

```
def elimina_ficheros(ruta):
    for root, dirs, files in os.walk(ruta, topdown=False):
        for file in files:
            os.remove(os.path.join(root, file))
            print(f"Eliminado: {os.path.join(root, file)}")
    os.rmdir(ruta)
    print(f"Archivos y directorio eliminados: {ruta}")
```

Y la puedes llamar así.

```
elimina_ficheros("./ejemplo")
```

### Ejercicios:

- En vez de añadir `prefijo_` a todos los ficheros, modifica la función para añadir un número que indique su orden alfabético. Por ejemplo para `ejemplo.txt` y `archivo.txt` el prefijo a añadir sería `1_archivo.txt` y `2_ejemplo.txt`. La `a` va antes

---

que la `e` alfabéticamente.

## 8.10 Bases de datos con sqlite

Veamos como trabajar con bases de datos para persistir nuestra información en el disco y que pueda ser usada una vez nuestro código ha terminado. Sin algún tipo de persistencia, toda la información que tu programa sabe es olvidada una vez este acaba.

Hay muchas formas de hacerlo. Puedes almacenar la información en un simple fichero. O puedes usar bases de datos relacionales y lenguajes como SQL.

En este ejemplo usaremos `sqlite3` para almacenar nuestros gastos mensuales en una base de datos `gastos.db`. Se trata de un paquete muy sencillo de utilizar y a diferencia de otros no requiere un servidor externo.

Empezamos creando la base de datos y una tabla llamada `gastos` con dos campos:

-  `categoria` : Tipo de gasto almacenado. Es un texto `TEXT` .
-  `cantidad` : Cantidad del gasto. Es un número real `REAL` .

Como puedes ver exigimos que en ambos casos el campo no puede ser `NULL` . Es decir, que no lo puedes dejar vacío. Todo el contenido de nuestra base de datos se almacenará en el fichero `gastos.db` .

```
import sqlite3

def conecta_db():
    return sqlite3.connect('gastos.db')

def crea_tabla():
    with conecta_db() as conn:
        conn.execute('''


```

```
CREATE TABLE IF NOT EXISTS gastos (
    categoria TEXT NOT NULL,
    cantidad REAL NOT NULL
)'''
```

Ahora necesitamos una función para añadir gastos a nuestra tabla.

```
def add_gasto(categoría, cantidad):
    with conecta_db() as conn:
        conn.execute('''
            INSERT INTO gastos (categoría, cantidad)
            VALUES (?, ?)'', (categoría, cantidad))
```

También una función para obtener los gastos. Esta función consulta la base de datos y nos devuelve los gastos que han sido almacenados. También puedes usar `categoría` para filtrar los gastos por categoría. Si no lo usas, se devolverán todos los gastos.

```
def get_gastos(categoría=None):
    with conecta_db() as conn:
        cursor = conn.cursor()
        query = 'SELECT * FROM gastos WHERE '
        condiciones, parametros = [], []

    if categoría:
        condiciones.append("categoría = ?")
        parametros.append(categoría)

    query += " AND ".join(condiciones) if
    condiciones else "1=1"

    cursor.execute(query, parametros)
    return cursor.fetchall()
```

Ahora vamos a crear nuestros gastos. Creamos la tabla y usamos la función `add_gasto` para añadir nuestros gastos.

```
crea_tabla()
add_gasto("Transporte", 5)
add_gasto("Comida", 5)
```

```
add_gasto("Comida", 7)
add_gasto("Alquiler", 300)
```

Comprobamos que han sido almacenados correctamente. Deberíamos obtener los mismos que hemos introducido.

```
gastos = get_gastos()
for gasto in gastos:
    print(gasto)
```

Consultamos todos los gastos de una categoría concreta.

```
gastos = get_gastos(categoría="Comida")
for gasto in gastos:
    print(gasto)
```

Como puedes ver tus gastos se persisten en el disco duro aunque tu código haya terminado.

💡 Ejercicios:

- Añade a la tabla `gastos` de tu base de datos un campo llamado `fecha`, que almacene cuando se realizó el gasto.
- Añade a `get_gastos` la posibilidad de filtrar por fecha. Por ejemplo, los gastos de un mes concreto.
- Añade todas las operaciones CRUD sobre la base de datos.  
Las operaciones CRUD son *Create, Remove, Update y Delete*. Tenemos `add_gasto` y `get_gastos` . Añade `remove_gasto` y `update_gasto` .

## 8.11 Patrones con re

Veamos cómo validar si una dirección de correo está bien escrita. Aunque es algo que el ser humano ve en un golpe de vista, validar esto en programación es un poco más complicado.

- `esto@no.es@un.correo@valido` Dirección de correo no válida.
- `nombre@dominio.es` Dirección de correo válida.

Para resolver este problema usaremos las expresiones regulares o *regular expressions*. Estas nos permiten buscar patrones en texto. Este ejemplo nos permite detectar números en una cadena. Como puedes ver se detectó el `2` y el `3`.

```
import re
r1 = re.findall(r"\d+", "Tengo 2 gatos y 3 perros")
print(r1) # ['2', '3']
```

También podemos reemplazar texto. En este caso reemplazamos `2` y `3` por `dos` y `tres` respectivamente.

```
r2 = re.sub(r"2", "dos", "Tengo 2 gatos y 3 perros")
r2 = re.sub(r"3", "tres", r2)
print(r2)
# Tengo dos gatos y tres perros
```

Pero también validar que un texto cumple con unas normas. Por ejemplo, sabemos que un correo electrónico como `usuario@gmail.com` está compuesto por:

- Un nombre de `usuario`.
- Seguido de `@`.
- Un dominio como `gmail`.
- Un TLD como `.com`.

Podemos por tanto escribir una expresión que nos diga si una dirección de correo es válida o no de la siguiente manera.

```
def valida_email(email):
    usuario = r"[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+"
    arroba = r"@"
    dominio = r"[a-zA-Z0-9-]+"
    tld = r"(?:\.[a-zA-Z0-9-]+)*"
```

---

```

email_regex = rf"^{usuario}{arroba}{dominio}{tld}$"
"

return re.match(email_regex, email) is not None

print(valida_email("juan@gmail.com"))      # True
print(valida_email("juan@gmail.com.ar"))    # True
print(valida_email("653fsaasd"))            # False
print(valida_email("juan@gma@.dom.com"))   # False
print(valida_email("@@@"))                  # False

```

Puede parecer complicado, así que veamos parte por parte:

- `usuario` : Para el nombre de usuario se permiten números, minúsculas, mayúsculas y algunos caracteres raros. Por ejemplo `[a-zA-Z0-9]+` indica que se permiten números, minúsculas y mayúsculas. El `+` indica que buscamos uno o muchos caracteres de este tipo. El resto como `!` o `$` indica que estos caracteres también son permitidos.
- `arroba` : Buscamos que la `@` esté presente en el lugar adecuado.
- `dominio` : Requerimos un dominio con números, minúsculas, mayúsculas y guiones.
- `tld` : Requiere que empiece con un `.` y permitimos múltiples. Es decir, `.com` y `co.uk` son válidos.

Ejercicios:

- Modifica la función para que solo permita dominios `.com` .
- Modifica la función para que no admita dominios `@google` .

## 8.12 Analítica de fútbol con seaborn

En este ejemplo vamos a representar un mapa de calor o *heatmap* desde donde se iniciaron más pases en el partido de fútbol España-

---

Inglaterra del 14 de julio de 2024.

- ⚽ Con `statsbombpy` obtenemos los datos del partido
- 📈 Con `seaborn` representamos la información gráficamente.

Primero obtenemos los datos usando el `match_id` de ese partido. Después filtramos los datos para obtener el origen de los pases que realizó Inglaterra a lo largo del partido.

```
from statsbombpy import sb
import seaborn as sns
import matplotlib.pyplot as plt

ev = sb.events(match_id=3943043)
ev = ev.loc[
    (ev['type'] == "Pass") &
    (ev['team'] == "England")]
```

Llegados aquí, en `ev` tenemos todos los eventos relativos a pases de Inglaterra. Pero lo más importante para el ejemplo es:

- 📺 El campo `location` : El punto de inicio de cada pase.
- 🏟 El campo `pass_end_location` : El punto donde finalizó el pase.

Ahora con `seaborn` representamos el *heatmap* de donde se iniciaron los pases. Esto nos permite hacernos una idea de los puntos calientes desde donde se inician los pases. Podemos ver que el portero inició muchos saques.

```
plt.figure(figsize=(8, 4))

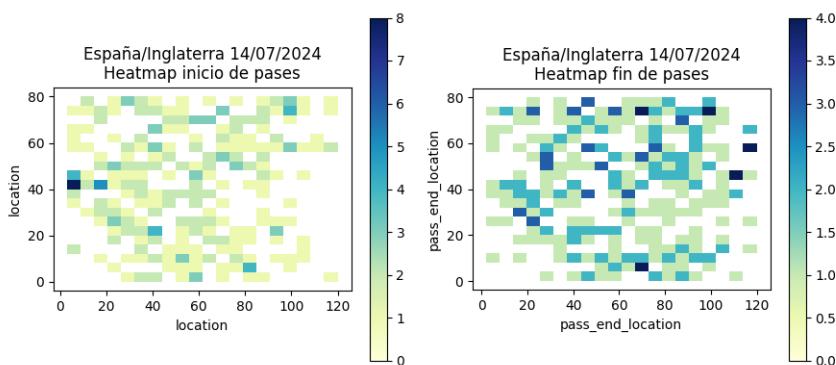
plt.subplot(1, 2, 1)
ax1 = sns.histplot(
    x=ev['location'].str[0],
    y=ev['location'].str[1],
    bins=(20, 20), cmap="YlGnBu", cbar=True)
ax1.set_aspect('equal', 'box')
```

```
plt.title('España/Inglaterra 14/07/2024\n' 'Heatmap inicio de pases')
```

Realizamos lo mismo con los puntos donde se finalizaron los pases. Como puedes ver usamos `str[0]` y `str[1]`. Estas son las coordenadas `x` e `y` en el campo.

```
plt.subplot(1, 2, 2)
ax2 = sns.histplot(
    x=ev['pass_end_location'].str[0],
    y=ev['pass_end_location'].str[1],
    bins=(20, 20), cmap="YlGnBu", cbar=True)
ax2.set_aspect('equal', 'box')
plt.title('España/Inglaterra 14/07/2024\n' 'Heatmap fin de pases')

plt.tight_layout()
plt.show()
```



### Ejercicios:

- Realiza la misma representación para España y compáralo con el *heatmap* que acabamos de ver de Inglaterra.

## 8.13 Programar tareas con schedule

Usando la librería `schedule` puedes programar tareas para que se ejecuten cada cierto intervalo. En este ejemplo ejecutamos `tarea` una vez cada 10 minutos.

```
import schedule
import time
from datetime import datetime

def tarea():
    print(f"{datetime.now()}: Tu tarea")

schedule.every(10).minutes.do(tarea)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Es importante notar el uso de `sleep(1)`. Al tener un bucle `while` infinito, sin este pequeño `sleep` nuestro programa estaría continuamente ejecutando el bucle y sería un desperdicio de recursos.

Por otro lado, aunque este ejemplo funciona perfectamente y es válido para la mayoría de los casos, es posible que quieras usar un *thread* distinto.

Con el ejemplo anterior, si `tarea` tarda en ejecutarse más de 10 minutos, estaría bloqueando la siguiente ejecución. Esto haría que tu tarea ya no se ejecutase cada `10` minutos. La tarea podría bloquear la siguiente ejecución.

Si quieres que tu tarea se ejecute exactamente cada 10 minutos sin importar que `tarea` tarde, por ejemplo, 11 minutos, puedes hacer lo siguiente.

```
import schedule
import time
```

```
import threading
from datetime import datetime

def tarea():
    print(f"{datetime.now()}: Tu tarea")

def tarea_threaded():
    task_thread = threading.Thread(target=tarea)
    task_thread.start()

schedule.every(10).minutes.do(tarea_threaded)

while True:
    schedule.run_pending()
    time.sleep(1)
```

### Ejercicios:

- En vez de ejecutar `tarea` cada cierto intervalo, usa `schedule` para ejecutarla todos los días a las 23:59.

## 8.14 Gráficas con matplotlib

En este ejemplo vamos a usar `matplotlib` para representar nuestros datos en gráficas de diferentes tipos. Este paquete es uno de los más usados en Python para representar datos.

Vamos a empezar definiendo unos datos aleatorios que representan la temperatura a lo largo de un año. Generamos `365` muestras con una distribución normal con media `20`.

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
temperatura = np.random.normal(20, 5, 365)
```

---

Ahora que tenemos en `temperatura` los datos, vamos a representarlos en diferentes gráficas:

- 📈 Una gráfica con la evolución de la temperatura con el tiempo.
- 📊 Un histograma con la distribución de las temperaturas.

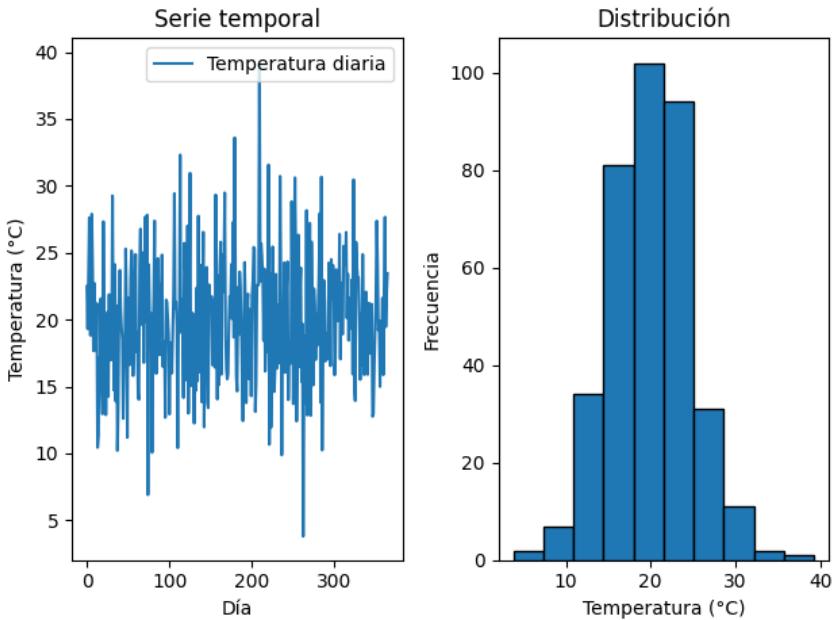
```
fig, ax = plt.subplots(1, 2)

ax[0].plot(temperatura, label="Temperatura diaria")
ax[0].set_xlabel("Día")
ax[0].set_ylabel("Temperatura (°C)")
ax[0].set_title("Serie temporal")
ax[0].legend()

ax[1].hist(temperatura, bins=10, edgecolor="black")
ax[1].set_xlabel("Temperatura (°C)")
ax[1].set_ylabel("Frecuencia")
ax[1].set_title("Distribución")

plt.tight_layout()
plt.show()
```

Es muy importante que no olvides el `plt.show()`. No es la primera vez que alguien pierde horas de tiempo por olvidárselo.



### 💡 Ejercicios:

- Modifica la gráfica del histograma para que en vez de dar la frecuencia en número de veces de cada valor de temperatura, la muestre en porcentaje.

## 8.15 Redes neuronales con tensorflow

En este ejemplo vemos como entrenar una red neuronal para reconocer números escritos a mano. Este modelo permite:

- 📸 Dada una entrada de una imagen con un número dibujado.
- 📤 Devuelve el número presente en esa imagen de `0` a `9`.

Empezamos importando todo lo que necesitamos. Ambos `keras` y `tensorflow` son los paquetes más usados para tareas relacionadas con *machine learning* e inteligencia artificial.

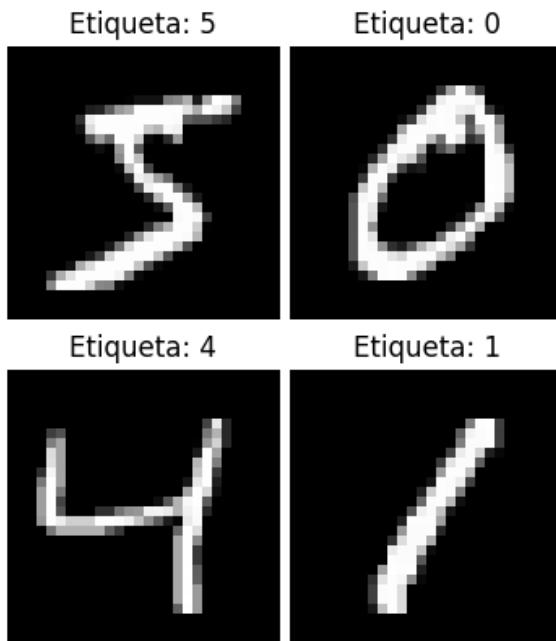
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.layers import Conv2D,
    MaxPooling2D
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
```

Ahora vamos a importar un conjunto de datos llamado MNIST. Este es un conjunto de 70.000 imágenes que contiene dígitos del 0 al 9 escritos a mano. Lo usaremos para entrenar nuestro modelo. Mostramos unos ejemplos de nuestro *dataset* para ver que pinta tienen.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
()

plt.figure()
for i in range(4):
    plt.subplot(2, 2, i + 1)
    plt.imshow(X_train[i], cmap='gray')
    plt.title(f"Etiqueta: {y_train[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Como puedes ver, todo son imágenes de 28x28 píxeles de dígitos escritos a mano. Tenemos un total de 70.000.



Para entender mejor nuestro *dataset*, puedes ver que tenemos dos tipos de variables.

- `x_`: Es la imagen. Un vector con 28x28 píxeles en blanco y negro.
- `y_`: Es la etiqueta de la imagen. Es decir, si es un `0`, un `1`, etc.

Esto que estamos haciendo se llama aprendizaje supervisado, ya que sabemos *a priori* el contenido de cada imagen. Estas etiquetas las ha puesto un humano.

Por otro lado tenemos otros dos tipos de variables:

- `train`: Son las imágenes que usaremos para entrenar nuestro modelo. De las 70.000 se toman 60.000 para entrenar.

- 
-  **test** : Son las imágenes usadas para verificar que nuestro modelo funciona correctamente. No se usan para entrenar el modelo. Permiten evaluar si el modelo es capaz de generalizar y clasificar correctamente imágenes no vistas anteriormente.

Ahora que tenemos nuestros datos de test y entrenamiento, tenemos que realizar un pequeño procesado.

- Normalizamos los píxeles. Es decir, hacemos que el nivel de negro varíe entre 0 y 1 en vez de 0 y 255.
- Redimensionamos los datos, añadiendo una dimensión extra. En realidad, no cambia nada, ya que nuestras imágenes tienen un único canal, blanco y negro.
- Convertimos las etiquetas. Por ejemplo, de **3** a **[0 0 0 1 0 0 0 0 0 0]**.  
Lo mismo pero expresado de otra forma.

```
X_train, X_test = X_train / 255.0, X_test / 255.0

X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Ahora definimos la arquitectura de nuestro modelo. Se trata de una arquitectura típica de CNN (*Convolutional Neural Network*).

```
cnn_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape
          =(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

---

Compilamos el modelo, indicando la función de *loss* que vamos a usar. Esta función de *loss* indica como evaluar lo bien o mal que el modelo realiza la predicción. Entrenar un modelo consiste en buscar minimizar ese error.

Es importante notar que hasta ahora el modelo “está vacío” y aún no sirve para nada.

```
cnn_model.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])
```

Ahora ya podemos entrenar el modelo con `fit`. Esto consiste en usar nuestros datos de entrenamiento para darle forma a nuestra red neuronal. Concretamente en calcular sus *weights*. Iterativamente se van buscando los *weights* que reducen el error, aplicando la función de *loss* indicada para mejorar el modelo que se adapta a nuestros datos.

```
cnn_model.fit(X_train, y_train,
               epochs=5, batch_size=32,
               validation_data=(X_test, y_test))
```

Después de unos minutos, tendremos nuestro modelo entrenado. Pero antes de usarlo, tenemos que evaluar como de bueno es. Imagina que entrena un modelo que falla el 50% de las veces. Serviría de poco.

Con `evaluate` podemos ver lo bueno que es nuestro modelo. Para ello usamos los datos de test, ya que son datos que el modelo no ha visto anteriormente. No sería justo usar datos ya conocidos por el modelo. Veamos que la *accuracy* es del 99.23%.

```
_, accuracy = cnn_model.evaluate(X_test, y_test,
                                 verbose=0)
print(f"La accuracy es: {accuracy * 100:.2f}%")
```

```
# La accuracy es: 99.23%
```

Por último, con `predict` puedes usar el modelo para clasificar tu imagen. Hemos tomado una del set de test, pero podrías usar tu propia imagen.

```
predicciones = cnn_model.predict(X_test[0:2])
esperado = y_test[0:2]
for p, e in zip(predicciones, esperado):
    prediccion = np.argmax(p)
    print(f"Esperado: {e}. Predicción: {prediccion}")

# Esperado: 7. Predicción: 7
# Esperado: 2. Predicción: 2
```

Un apunte importante es que `predict` no devuelve exactamente el número que hay en la imagen. Nos devuelve una lista con probabilidades. Nos dice que existe una probabilidad del:

- 0.0001 de que sea 0 .
- 0.003 de que sea 1 .
- 0.99 de que sea 2 .
- ...

Con `argmax` obtenemos la predicción que mayor probabilidad tiene. En este caso la mayor probabilidad es de que sea 2 . Por lo tanto aceptamos el 2 como resultado.

### Ejercicios:

- Almacena el modelo entrenado en disco para que puedas usarlo sin tener que entrenarlo cada vez. Usa ese modelo almacenado en el inicio de tu *script*.
- Junta los datos de test y entrenamiento y entrena el modelo usando solo un 1% de los datos para validación. Indica si esto mejora o empeora la *accuracy*. Pista, usa `validation_split` en

---

```
fit .
```

- Dibuja un número a mano en un papel. Haz una foto. Conviértele a 28x28 píxeles y usa `predict` sobre él.

## 8.16 Logging y verbosity con logging

El uso de *logs* es muy importante en cualquier código. En este ejemplo vamos a ver como establecer diferentes niveles de importancia en tus *logs* y configurarlo con un argumento por línea de comandos.

Los *logs* permiten observar el código desde el exterior, viendo por donde pasa la ejecución. Son de vital importancia cuando las cosas no salen como uno espera, ya que ayudan a entender que pasó. Es importante poder controlar el nivel de detalle de los *logs*, ya que no siempre queremos verlo todo.

Para pequeños *scripts* tal vez basta con usar `print`. Pero en programas complejos que corren 24/7 necesitamos herramientas más profesionales como `logging`, lo que permite tener diferentes niveles de *logs* según su importancia.

- `INFO` : Para información importante, se usa `logger.info()` .
- `DEBUG` : Para información mas detallada, se usa `logger.debug()` .

En el siguiente ejemplo vemos como usar `logging` con `argparse` para que puedas configurar el nivel de detalle de tus *logs* desde la línea de comandos.

```
# programa.py
import argparse
import logging

parser = argparse.ArgumentParser(description="Nivel de
    verbosity")
parser.add_argument('--log-level', type=str,
```

```
        default='INFO',
        choices=['DEBUG', 'INFO',
                  'WARNING', 'ERROR'])

args = parser.parse_args()
logging.basicConfig(level=args.log_level, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger('logger')

logger.info('Ejemplo info')
logger.debug('Ejemplo debug')
logger.warning('Ejemplo warning')
```

Si ejecutamos nuestro código con `INFO` solo se muestran los *logs* hasta `INFO`.

```
python3 programa.py --log-level=INFO
# 2024-10-30 12:51:06,991 - INFO - Ejemplo info
# 2024-10-30 12:51:06,991 - WARNING - Ejemplo warning
```

Si usamos `DEBUG` se muestran los logs hasta `DEBUG`.

```
python3 programa.py --log-level=DEBUG
# 2024-10-30 12:50:37,421 - INFO - Ejemplo info
# 2024-10-30 12:50:37,421 - DEBUG - Ejemplo debug
# 2024-10-30 12:50:37,421 - WARNING - Ejemplo warning
```

Estos son los diferentes niveles que ofrece `logging`.

```
DEBUG < INFO < WARNING < ERROR < CRITICAL
```

Como podemos ver `log-level` permite configurar el detalle de los *logs*. Puedes usar `INFO` para información general y `DEBUG` para el máximo detalle.

💡 Ejercicios:

- Modifica el código para que acepte el nivel de *log* como `INFO` e `info` sin importar las mayúsculas.

- 
- Escribe una función donde uses `logger.info` y `logger.debug` para *logs* con diferente importancia.

## 8.17 Cálculo simbólico con sympy

En este ejemplo vemos como derivar e integrar expresiones matemáticas con `sympy`. Este paquete permite cálculo simbólico en Python. Nos permite trabajar con expresiones matemáticas de manera algebraica, en vez de solo con números como estamos acostumbrados.

Vamos a definir una expresión matemática con la `x` como variable independiente. Definimos `x` como variable simbólica y creamos una expresión `p`. Vemos el polinomio resultante.

```
import sympy as sp

x = sp.symbols('x')
p = x**3 + 3*x + 10

print(p)
# x**3 + 3*x + 10
```

Usando `diff` e `integrate` podemos derivar e integrar nuestra expresión con respecto a la variable simbólica `x`.

```
print("Derivada:", sp.diff(p, x))
# Derivada: 3*x**2 + 3

print("Integral:", sp.integrate(p, x))
# Integral: x**4/4 + 3*x**2/2 + 10*x
```

### 💡 Ejercicios:

- Dado el sistema de ecuaciones  $x+7y=10$  y  $3x-2y=7$  usa `sympy` para encontrar la solución de `x` e `y` que hace que se verifique la igualdad.

---

## 8.18 Creando un juego con pygame

Gracias a `pygame` podemos crear juegos en Python de manera muy sencilla. Permite realizar animaciones interactivas 2D, que reciben eventos de teclado ratón o joystick. También permite detectar colisiones y manejar sonidos.

En este ejemplo vamos a hacer un pequeño juego que permite al usuario mover con las teclas del teclado un rectángulo.

Empezamos por crear una pantalla con unas dimensiones y creamos nuestro jugador, un pequeño rectángulo.

```
import pygame
import sys

pygame.init()
ancho, alto = 400, 300
pantalla = pygame.display.set_mode((ancho, alto))

jugador = pygame.Rect(50, 50, 20, 20)
velocidad = 5
```

Ahora usamos un bucle infinito donde continuamente actualizamos la posición del jugador. El uso de `keys` nos permite detectar la tecla que se ha pulsado. Basándonos en la tecla pulsada, actualizamos las coordenadas de nuestro jugador.

```
while True:
    pantalla.fill((0, 0, 0))
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    keys = pygame.key.get_pressed()
    if keys[pygame.K_LEFT]: jugador.x -= velocidad
    if keys[pygame.K_RIGHT]: jugador.x += velocidad
    if keys[pygame.K_UP]: jugador.y -= velocidad
```

---

```
if keys[pygame.K_DOWN]: jugador.y += velocidad

pygame.draw.rect(pantalla, (0, 255, 0), jugador)

pygame.display.flip()
pygame.time.delay(30)
```

💡 Ejercicios:

- Evita que el `jugador` pueda salirse de la pantalla, poniendo un límite.
- Implementa el juego *snake* con `pygame`.

## 8.19 Scrapping web con beautifulsoup

El *scrapping* web consiste en acceder al contenido de páginas web de manera automatizada. Consiste en descargarse su contenido, que normalmente es HTML, para procesarlo y extraer la información que nos interesa.

En este ejemplo vamos a ver como hacer *scrapping* la siguiente página de Wikipedia:

- [https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

Pongamos que el contenido que nos interesa es la siguiente tabla. Nuestro objetivo es escribir un *script* que permita acceder a este contenido desde Python.

## Python

	
Paradigm	Multi-paradigm: object-oriented,[1] procedural (imperative), functional, structured, reflective
Designed by	Guido van Rossum
Developer	Python Software Foundation

Empezamos importando `requests` para obtener el código HTML y `bs4` para buscar en el HTML.

```
import requests
from bs4 import BeautifulSoup

url = "https://en.wikipedia.org/wiki/Python_(
       programming_language)"
tabla = {}
```

Con el siguiente código nos descargamos la web y buscamos la tabla de la clase `infobox vevent` que es la que almacena lo que nos interesa. Para localizar las etiquetas HTML del código que te interesa, desde tu navegador haz click derecho e “Inspeccionar”.

```
try:
    response = requests.get(url)
    response.raise_for_status()
```

```
soup = BeautifulSoup(response.text, 'lxml')

infobox = soup.find('table', class_='infobox
vevent')
if infobox:
    tabla = {
        fila.find('th').get_text(strip=True): fila
            .find('td').get_text(strip=True)
        for fila in infobox.find_all('tr')
        if fila.find('th') and fila.find('td')
    }

except requests.exceptions.RequestException as e:
    print(f"Error: {e}")
```

Esto almacena en `tabla` todo el contenido de la tabla de esa página de la Wikipedia. La *key* es la primera columna y el *value* la segunda. Podemos acceder a la entrada de `Developer` así. Como puedes ver, el contenido corresponde al de la web.

```
print(tabla["Developer"])
# Python Software Foundation
```

Aunque en este ejemplo hemos hecho *scrapping* de un campo muy concreto, el hecho de ser automático permite escalarlo mas allá. Por ejemplo, podrías hacer *scrapping* todos los días del precio de diferentes productos de Amazon y guardarlo en una base de datos para al final del año, ver la evolución del precio.

### Ejercicios:

- Crea una función que vaya a la sección de referencias y obtenga todos los enlaces que existen y los meta en una lista. Solo enlaces.

---

## 8.20 Unir pdfs con pypdf

En este ejemplo vemos como juntar dos pdf con `pypdf`. Este paquete permite trabajar en Python con ficheros `.pdf`. Los puedes unir, eliminar páginas, añadir marcas de agua, encriptar, desencriptar o extraer contenido de páginas concretas.

Vamos a empezar creando dos ficheros pdf vacíos.

```
from pypdf import PdfWriter  
  
pdf_writer = PdfWriter()  
pdf_writer.add_blank_page(595.28, 841.89)  
  
with open("pdf1.pdf", "wb") as f:  
    pdf_writer.write(f)  
  
with open("pdf2.pdf", "wb") as f:  
    pdf_writer.write(f)
```

Y ahora lo interesante. Veamos como juntar dos pdf `pdf1.pdf` y `pdf2.pdf` en un fichero nuevo llamado `unidos.pdf`.

```
pdf_merger = PdfWriter()  
  
with open("pdf1.pdf", 'rb') as file1:  
    pdf_merger.append(file1)  
  
with open("pdf2.pdf", 'rb') as file2:  
    pdf_merger.append(file2)  
  
with open("unidos.pdf", 'wb') as output_file:  
    pdf_merger.write(output_file)
```

En apenas 6 líneas de código lo hemos realizado. Así que la próxima vez que quieras unir dos pdf, plantéate hacerlo usando Python en vez de recurrir a servicios online de dudosa reputación y privacidad.

 Ejercicios:

- 
- Escribe una función que elimine la última página de un pdf.
  - Escribe una función que una todos los pdf que existan en una carpeta.

## 8.21 Crear excels con pyexcel

En este ejemplo, vemos cómo crear un fichero Excel con Python, utilizando `pyexcel`. Este paquete permite crear archivos Excel, añadiendo contenido y fórmulas.

Partiendo de unos gastos en euros, vamos a crear un fichero excel que los liste en una columna y los sume todos. Empezamos importando todo y con unos datos de ejemplo.

```
from openpyxl import Workbook
from openpyxl.styles import Font

datos = [
    ["Alquiler", 800],
    ["Electricidad", 100],
    ["Agua", 50],
    ["Internet", 60],
    ["Supermercado", 300]
]
```

Ahora añadimos todo el contenido:

- 📈 Le damos nombre a nuestras dos columnas `A1` y `B1`.
- 🎨 También damos un estilo al texto, usando **negrita** para la fuente.
- 🔍 Iteramos nuestros datos `concepto` y `gasto` y los vamos metiendo en las filas consecutivas, empezando por la `2`.
- ✚ Añadimos una última fila con la fórmula `SUM` para sumar todos los gastos.

```

wb = Workbook()
ws = wb.active

ws['A1'] = "Concepto"
ws['B1'] = "Gasto €()"
bold_font = Font(bold=True)
ws['A1'].font = bold_font
ws['B1'].font = bold_font

for i, (concepto, gasto) in enumerate(datos, start=2):
    ws[f'A{i}'] = concepto
    ws[f'B{i}'] = gasto

ultima = len(datos) + 1

ws[f'A{ultima + 1}'] = "Gastos Totales"
ws[f'B{ultima + 1}'] = f"=SUM(B2:B{ultima})"

wb.save("gastos.xlsx")

```

Si ejecutas este programa, crearás el fichero Excel `gastos.xlsx`. Ábrelo y podrás ver lo siguiente.

	A	B
1	Concepto	Gasto (€)
2	Alquiler	800
3	Electricidad	100
4	Agua	50
5	Internet	60
6	Supermercado	300
7	Gastos Totales	1310
8		

### 💡 Ejercicios:

- Añade una nueva columna que se llame “Porcentaje (%)” y que

---

calcule el porcentaje de un gasto sobre el total. Por ejemplo, el alquiler (800€) representa el 61% del total (1310 €).

## 8.22 Benchmark con timeit

En este ejemplo vemos como medir la eficiencia de tu código Python. Existen dos métricas muy importantes que reflejan la eficiencia de nuestro código:

- ⚡ Velocidad: El tiempo que tarda en ejecutarse, medido en segundos.
- 📈 Memoria: La memoria RAM que consume, medido en MB.

Python no es el lenguaje más eficiente, pero existen formas de mejorarlo. Siempre que queramos optimizar nuestro código, es necesario medir estas métricas y comparar el resultado antes y después.

Nos centraremos en la primera, la velocidad. Ahora supongamos que tenemos dos funciones que realizan lo mismo:

- `fun_a` : Multiplica una lista por un número usando *list comprehension*.
- `fun_b` : Multiplica una lista por un número usando `numpy` .

Ambas realizan la misma tarea y no estamos seguros de cual usar. Podemos por tanto medir la velocidad de ambas y decidirnos por la más rápida.

```
from timeit import timeit
import numpy as np

def fun_a(data, scalar):
    return [x * scalar for x in data]

def fun_b(data, scalar):
    return data * scalar
```

---

En muchas ocasiones los problemas de eficiencia solo se notan cuando trabajamos con muchos datos. Para ello vamos a definir una lista con un millón de elementos.

Después medimos con `timeit` el tiempo que tarda en ejecutarse cada una. Promediamos 10 ejecuciones. Podemos ver que `func_b` es mucho más rápida.

```
lista = list(range(10**6))
array = np.array(lista)

t_a = timeit(lambda: fun_a(lista, 5.55), number=10)
print(f"fun_a: {t_a / 10:.4f} segundos")

t_b = timeit(lambda: fun_b(array, 5.55), number=10)
print(f"fun_b: {t_b / 10:.4f} segundos")

# fun_a: 0.0360 segundos
# fun_b: 0.0010 segundos
```

Siempre que quieras optimizar un código empieza por aquí. Primero mide lo que tarda para entender el punto de partida. Y después intenta optimizar.

Otro apunte importante es el consumo de memoria. Tal vez una solución sea más rápida, pero consuma más memoria. En este caso, tendrás que decidir qué es lo importante para tu programa.

### Ejercicios:

- Encuentra otra dos formas de realizar la misma tarea en Python donde una de ellas sea más rápida.
- Explora como medir el consumo de memoria. Esta es junto con la velocidad otra métrica importante.

---

## 8.23 Rotar imagen con scikit-image

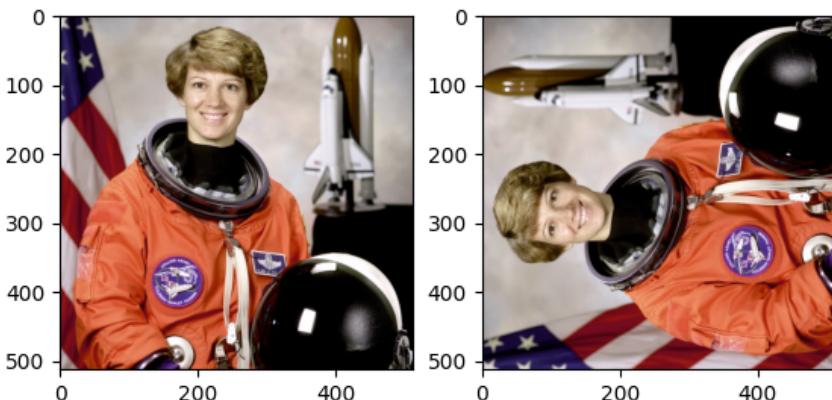
En este ejemplo vamos a ver como rotar una imagen con `scikit-image`. Este paquete permite realizar procesamiento de imágenes de todo tipo.

Usamos una imagen por defecto, `astronaut` y la cargamos en `imagen`. Después con `rotate` la rotamos 90 grados. Por último con `imshow` representamos ambas imágenes, la que ha sido rotada y la que no.

```
from skimage import data
from skimage.transform import rotate
import matplotlib.pyplot as plt

imagen = data.astronaut()
imagen_rotada = rotate(imagen, angle=90, resize=True)

fig, ax = plt.subplots(1, 2)
ax[0].imshow(imagen)
ax[1].imshow(imagen_rotada)
plt.show()
```



---

### Ejercicios:

- Convierte la imagen a blanco y negro.
- Escribe un código para guardar la imagen a un fichero `.png`.

## 8.24 Simulando parada de bus con `simpy`

En este ejemplo vamos a simular una parada de bus donde llegan personas y autobuses, usando `simpy`. Este paquete permite simular componentes como vehículos o personas y su interacción entre ellos a través de eventos.

Veamos un ejemplo simulando la parada de un bus. Tenemos dos procesos:

-  Personas: Personas que llegan a la parada aleatoriamente.
-  Bus: Bus que llega a la parada y se lleva a gente a intervalos fijos con cierta aleatoriedad.

Este tipo de simulaciones puede ser interesante para dimensionar el tamaño de los autobuses y frecuencias de los mismos, buscando que la gente espere lo menos posible y sin derrochar recursos.

Si tienes `10` personas que llegan por minuto y un bus que pasa cada `1` hora con `20` plazas, con total seguridad en la parada se acumulará gente y tus usuarios se enfadarán. No queremos que pase esto.

Empezamos importando lo que necesitamos.

```
import simpy
import random
```

Ahora definimos una `Parada` de bus con dos procesos. Uno simula personas llegando a la parada de bus y otro simula el bus llegando a la parada y llevándose a la gente.

```

class Parada:
    def __init__(self, env, capacidad_bus):
        self.personas = 0
        self.env = env
        self.capacidad_bus = capacidad_bus

    def llega_persona(self):
        while True:
            llegan = random.randint(0, 10)
            print(f"[PERSONA] Llegan {llegan} en el
                  minuto {self.env.now}")
            self.personas += llegan
            yield self.env.timeout(1)

    def llega_bus(self):
        while True:
            se_lleva = min(self.capacidad_bus, self.
                           personas)
            self.personas -= se_lleva
            print(f"[BUS] Se lleva: {se_lleva} y
                  quedan {self.personas}")
            yield self.env.timeout(random.uniform(5,
                                                   10))

```

Veamos paso por paso:

- `llega_persona` : Cada 1 segundo llegan a la parada un número de personas aleatorias entre 0 y 10 .
- `llega_bus` : Un bus llega una vez aleatoriamente entre 5 y 10 minutos, con una distribución normal. Cada vez que llega se lleva `capacidad_bus` personas.

Ahora creamos el entorno `env` y ejecutamos la simulación durante 120 minutos.

```

env = simpy.Environment()
parada = Parada(env, capacidad_bus=50)

env.process(parada.llega_persona())

```

```
env.process(parada.llega_bus())
env.run(until=120)
```

Puedes ver como con estos parámetros casi nunca queda gente esperando por el bus. Sin embargo si bajamos la `capacidad_bus` a 30 podremos ver como el bus no da abasto y cada vez se acumulan más personas esperando en la parada.

Este tipo de simulaciones son muy útiles para dimensionar líneas de bus o metro. Ante una demanda determinada podemos estimar:

- El tamaño del bus medido en capacidad de personas.
- La frecuencia del bus medida en tiempo.

Esto es también aplicable a simulaciones sobre propagación de virus, mercados financieros o tráfico. Te animamos a que explores todo lo que `simpy` ofrece.

Ejercicios:

- Usando `matplotlib` representa en una gráfica el número de personas que están en la parada esperando a lo largo del tiempo. Realiza modificaciones en `Parada` si hiciera falta.

## 8.25 Coordenadas y distancia con `geopy`

En este ejemplo vemos como calcular la distancia entre dos coordenadas geográficas con `geopandas`. En concreto entre las ciudades de Madrid y Buenos Aires.

Definimos las coordenadas, latitud y longitud. Y con `geodesic` obtenemos la distancia entre ambos puntos, que es de 10020.32 kilómetros.

```
from geopy.distance import geodesic
```

```
coord_madrid = (40.4168, -3.7038)
coord_buenos_aires = (-34.6037, -58.3816)

distance_km = geodesic(coord_madrid,
    coord_buenos_aires).kilometers

print(f"Distancia: {distance_km:.2f} km.")
# Distancia: 10020.32 km
```

### Ejercicios:

- El modelo que emplea `geodesic` por defecto para calcular la distancia es el WGS-84. Es el más usado pero existen otros. Experimenta con otros modelos de elipsoide viendo la diferencia entre las distancias que da cada uno.

## 8.26 Genética y ADN con biopython

En este ejemplo vemos como trabajar con secuencias de ADN usando `biopython`. Este paquete permite trabajar con secuencias de ADN, ARN, proteínas, y además ofrece herramientas para convertir entre formatos, realizar análisis de secuencias y trabajar con datos bioinformáticos en general.

Vamos a crear un programa que encuentre la secuencia complementaria de una secuencia de ADN. Como contexto, el ADN está definida por un conjunto de cuatro bases nitrogenadas:

-  Adenina (A)
-  Timina (T)
-  Citosina (C)
-  Guanina (G)

Con `seq` podemos crear una pequeña secuencia de ADN. Esta clase nos permite almacenar la secuencia en un objeto, y se nos ofrecen

---

diferentes operaciones. Por ejemplo `complement` permite encontrar la secuencia complementaria.

```
from Bio.Seq import Seq

adn = Seq("ATGGCCATTGTAATGGC")
print(f"ADN: {adn}")
# ADN: ATGGCCATTGTAATGGC

adn_compl = adn.complement()
print(f"Complementaria: {adn_compl}")
# Complementaria: TACCGGTAAACATTACCG
```

También podemos usar `transcribe` para transcribir el ADN a ARN y `translate` para traducir a una secuencia de proteínas.

```
arn = adn.transcribe()
print(f"Secuencia ARN: {arn}")
# ARN: AUGGCCAUUGUAAUGGGC

proteina = arn.translate()
print(f"Proteina: {proteina}")
# Proteina: MAIVMG
```

### Ejercicios:

- Sabiendo que la Adenina (A) siempre se empareja con Timina (T) y la Citosina (C) con Guanina (G) escribe la función `complement` en Python sin usar `biopython`. Esta debe mostrar la secuencia de ADN complementaria.

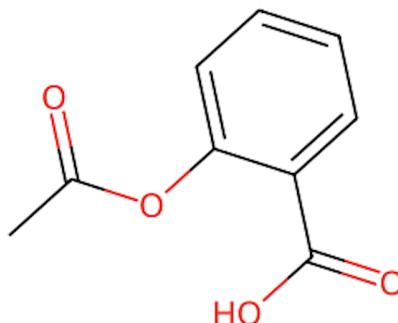
## 8.27 Visualizar moléculas con rdkit

En este ejemplo vemos como visualizar una molécula con `rdkit` . Este paquete es usado en química informática. Permite crear y manipular moléculas, calcular sus propiedades, visualizarlas, compararlas y realizar reacciones químicas.

Por ejemplo, la aspirina (ácido acetilsalicílico), cuya fórmula simplificada es `c9H8O4`, se puede expresar con notación SMILES de la siguiente manera. Usando `MolToImage`, la podemos representar gráficamente.

```
from rdkit import Chem
from rdkit.Chem import Descriptors
from rdkit.Chem import Draw

smiles = 'CC(=O)OC1=CC=CC=C1C(=O)O'
molecula = Chem.MolFromSmiles(smiles)
Draw.MolToImage(molecula).show()
```



Con `MolWt` podemos calcular la masa molecular de la aspirina, expresada en gramos por mol.

```
peso_molecular = Descriptors.MolWt(molecula)
print(f"Massa: {peso_molecular:.3f} g/mol")
# Massa: 180.159 g/mol
```

💡 Ejercicios:

- Usando `rdkit` busca una forma de convertir de notación SMILES a la fórmula, es decir de `CC(=O)OC1=CC=CC=C1C(=O)O` a

---

C9H8O4 para la aspirina o O a H2O para el agua.

## 8.28 Polinomios con numpy

En este ejemplo vemos como realizar operaciones con polinomios. Lo haremos de dos formas:

- Primero de forma manual, implementando nosotros las operaciones.
- Despues usando las funciones que nos ofrece `numpy` .

En muchas ocasiones Python nos ofrece todo lo que necesitamos sin tener que implementarlo nosotros. Pero es muy interesante entender como funcionan las cosas, intentando implementarlo nosotros aunque luego usemos un paquete externo.

Empecemos viendo que es un polinomio. Un polinomio tiene la siguiente forma:  $5x^3 + 3x^2 + 10x + 4$  . Tiene los siguientes componentes:

- La variable independiente. En nuestro caso usamos `x` como es común.
- Unos coeficientes. En nuestro caso `5` , `3` , `10` , `4` . Como puedes ver cada uno acompaña a una potencia de `x` . Un coeficiente puede ser cero.
- El grado. Se refiere al máximo exponente. En nuestro caso el grado es `3` .

Podemos expresar nuestro polinomio en Python de la siguiente manera. Una simple `list` con los coeficientes. Los ordenamos de menor a mayor grado.

```
p = [4, 10, 3, 5]
```

---

Es decir, asumimos que el primer índice  $p[0]$  corresponde al mínimo grado, es decir, al término  $x^0$ .

Ahora que sabemos que es un polinomio y cómo representarlo en Python, podemos definir las siguientes operaciones:

- $+$  Sumar. Suma dos polinomios  $p$  y  $q$ .
- $-$  Restar. Resta dos polinomios  $p$  y  $q$ .
- $\times$  Multiplicar. Multiplica dos polinomios  $p$  y  $q$ .
- Evaluar en un punto. Reemplaza la  $x$  de  $p$  por un valor concreto.

Empecemos con la suma y la resta.

```
from itertools import zip_longest

def suma(p, q):
    return [pp+qq for pp, qq in zip_longest(p, q,
                                              fillvalue=0)]

def resta(p, q):
    return [pp-qq for pp, qq in zip_longest(p, q,
                                              fillvalue=0)]
```

Es importante notar que usamos `zip_longest` en vez de `zip` ya que el segundo asume que ambas `list` tienen la misma longitud. En el caso de los polinomios esto no es el caso. Se pueden sumar polinomios con diferente grado.

La multiplicación tiene algo mas de miga. Consiste en tomar cada término del primero y multiplicarlo por todos los términos del segundo, sumando el resultado.

```
def multiplica(p, q):
    grado = len(p) + len(q) - 2
    resultado = [0] * (grado + 1)

    for i, pp in enumerate(p):
```

```
for j, qq in enumerate(q):
    resultado[i + j] += pp * qq

return resultado
```

Evaluar un polinomio en un punto es simplemente reemplazar  $x$  por un valor determinado y hacer las operaciones. Hay diferentes formas de hacerlo. Esto es el método de Horner, bastante eficiente.

```
def evalua(p, x):
    resultado = 0
    for coef in reversed(p):
        resultado = resultado * x + coef
    return resultado
```

Ahora usemos nuestras funciones. Definimos dos polinomios:

- $p$  : Con el valor  $x^3 + 3$  .
- $q$  : Con el valor  $5x^2 + x + 2$  .

```
p = [3, 0, 1]
q = [2, 1, 5]
```

⊕ Los sumamos. El resultado se interpreta como  $6 + x + 5x^2$  .

```
print(suma(p, q))
# [6, 1, 5]
```

⊖ Los restamos.

```
print(resta(p, q))
# [-4, -1, 1]
```

✗ Los multiplicamos

```
print(multiplica(p, q))
# [5, 1, 17, 3, 6]
```

█ Y evaluamos  $p$  en el punto  $5$  .

```
print(evalua(p, 5))  
# 76
```

Aunque como decimos es importante entender como funcionan las cosas, en la práctica puede ser mejor usar paquetes que ofrecen estas funciones ya implementadas. Y en muchos casos optimizadas.

Para esto podemos usar `numpy` y sus funciones. No hay necesidad de reinventar la rueda. Como puedes ver ofrece funciones para hacer todo. El resultado es el mismo.

```
import numpy as np  
  
p = [1, 0, 3]  
q = [5, 1, 2]  
  
print(np.polynomial.polynomial.polyadd(p, q))  
print(np.polynomial.polynomial.polysub(p, q))  
print(np.polynomial.polynomial.polymul(p, q))  
print(np.polynomial.polynomial.polyval(5, p))
```

💡 Ejercicios:

- Escriba una función que dados dos polinomios `p` y `q` realice su división, indicando su cociente y residuo.

## 8.29 Simulando apuestas con `numpy`

Veamos como simular una apuesta usando `numpy`. Simularemos una apuesta con las siguiente características:

- 🎲 Tenemos una moneda con `cara` y `cruz`.
- 💰 Se empieza con un `dinero_inicial`.
- ⚡ Se apuesta un número de veces `num_apuestas`.
- 🤞 Existe una probabilidad de ganar `prob_ganar`. Si se gana obtenemos `retorno_ganar`.

- 📉 Existe una probabilidad de perder `1-prob_ganar` . Si se pierde perdemos `retorno_perder` .

```
import numpy as np

def apuesta(dinero_inicial, prob_ganar,
            retorno_ganar, retorno_perder,
            num_apuestas):

    dinero = [dinero_inicial]
    p = [prob_ganar, 1 - prob_ganar]

    for _ in range(num_apuestas):
        if np.random.choice(['cara', 'cruz'], p=p) == 'cara':
            dinero.append(dinero[-1] * (1 + retorno_ganar))
        else:
            dinero.append(dinero[-1] * (1 + retorno_perder))

    return dinero
```

Como puedes ver si la probabilidad de ganar es `prob_ganar` , la de perder es `1-prob_ganar` . Esto es así porque ambas tienen que sumar `1` .

Por otro lado, usamos `choice` para generar el evento aleatorio de tirar la moneda. En una moneda normal no sesgada, `p=0.5` . Es decir, misma probabilidad de cara que de cruz.

Ahora podemos usar la función con los siguientes parámetros. El resultado de la simulación es la evolución del dinero a lo largo de todas las jugadas. Mostramos el dinero tras las `20` apuestas. Dado que se trata de un proceso aleatorio, no obtendrás el mismo valor.

```
simulacion = apuesta(
    dinero_inicial=1000,
    prob_ganar=0.5,           # 50% probabilidad ganar
```

```
retorno_ganar=0.8,      # +80% si ganas
retorno_perder=-0.5,   # -50% si pierdes
num_apuestas=20)
print(f"Dinero final €(): {simulacion[-1]:.0f}")
# Dinero final €(): 349
```

### Ejercicios:

- Representa gráficamente (eje-y dinero, eje-x número de apuesta) la evolución de la apuesta.
- Representa gráficamente (eje-y dinero, eje-x número de apuesta) la evolución de 1000 personas realizando la misma apuesta. Una línea de cada color por persona.
- Modifica la función `apuesta` para cambiar la estrategia. En vez de apostar todo cada vez, añade un parámetro `no_apostar` que indica el porcentaje que no se apuesta. Observa como cambia y explora la relación con el criterio de Kelly.

## 8.30 Simulación Monte Carlo con numpy

En este ejemplo realizamos una simulación de Monte Carlo sobre un portfolio de inversión usando `numpy`. Es común realizar este tipo de simulaciones en portfolios de inversión, donde existen los siguientes parámetros:

- 💰 Una cantidad de dinero inicial.
- 📈 Un retorno anual esperado pero no fijo. Varía de acuerdo a una volatilidad.
- 🤔 Una volatilidad. Esta volatilidad hace variar al retorno anual.
- ⏳ Un tiempo de varios años.

El objetivo es calcular la cantidad de dinero final después del tiempo en años considerado. Si la volatilidad fuera 0 y el retorno anual

---

fueras fijos cada año, el cálculo sería bien sencillo. Pero los mercados financieros están expuestos a incertidumbre.

Esta incertidumbre la podemos modelar con métodos de Monte Carlo, viendo los diferentes escenarios posibles. Nos dará una idea del mejor y del peor caso.

Podemos simular esta volatilidad con una distribución normal usando `normal`. Una vez lo tenemos, calculamos el `valor_portfolio` a lo largo de los años. Por último lo representamos en una gráfica.

```
import numpy as np
import matplotlib.pyplot as plt

def simula_portfolio(
    inicial, retorno_anual,
    volatilidad_anual, tiempo_anios,
    simulaciones):

    np.random.seed(42)
    retornos = np.random.normal(retorno_anual,
        volatilidad_anual, (tiempo_anios, simulaciones))
    valor_portfolio = inicial * (1 + retornos).cumprod(
        axis=0)

    plt.figure()
    plt.plot(valor_portfolio, alpha=0.5)
    plt.title("Simulación Monte Carlo de Portfolio")
    plt.xlabel("Año")
    plt.ylabel("Valor Portfolio €()")
    plt.grid(True)

    mejor_caso = valor_portfolio.max(axis=1)[-1]
    peor_caso = valor_portfolio.min(axis=1)[-1]
    plt.text(tiempo_anios / 2, mejor_caso / 2,
        f'Mejor caso: {mejor_caso:.0f€}'+f'\nPeor
            caso: {peor_caso:.0f€}',

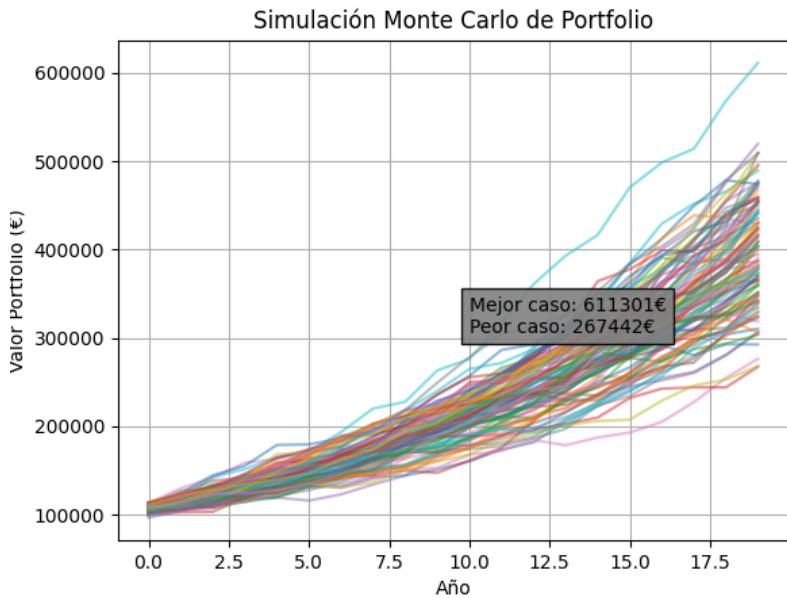
bbox=dict(facecolor='grey', alpha=0.9))
```

```
plt.show()
```

Ahora podemos usar la función con los siguientes parámetros. Realizamos 100 simulaciones con 20 años como horizonte temporal. Esperamos un retorno anual del 7% con una volatilidad del 4% y usamos 100.000 Euros.

```
simula_portfolio(  
    inicial=100_000,  
    retorno_anual=0.07,  
    volatilidad_anual=0.04,  
    tiempo_anios=20,  
    simulaciones=100)
```

Podemos ver los diferentes resultados que obtendría este portfolio. Esto nos da una idea del mejor y del peor caso. Como puedes ver, en ninguna de nuestras simulaciones se ha perdido dinero después de los 20 años.



### Ejercicios:

- Realiza la simulación usando diferentes combinaciones de `retorno_anual` y `volatilidad_anual`. Busca una combinación que de lugar a pérdidas en algunas situaciones.
- Añade una nueva representación que muestre la media de todos los escenarios posibles en una sola línea.

## 8.31 Análisis financiero con yfinance

En este ejemplo analizamos la acción de Apple e implementamos un sistema básico de indicadores para predecir los futuros movimientos.

Gracias al paquete `yfinance` nos podemos descargar información sobre diferentes mercados financieros. Seleccionamos `AAPL` que es el

---

*ticker* de Apple y las fechas que queremos. Con `rolling` calculamos la media móvil de 9 y 20 días.

```
import yfinance as yf
import numpy as np
import matplotlib.pyplot as plt

apple = yf.download('AAPL', start='2020-01-01', end
                   ='2021-01-01')

apple['9_MA'] = apple['Close'].rolling(window=9).mean
()
apple['20_MA'] = apple['Close'].rolling(window=20).
    mean()
```

Existen diferentes indicadores financieros que los analistas usan para intentar predecir si una acción subirá o bajará de precio. Uno muy usado es el cruce de medias. Este indicador dice lo siguiente:

- 📈 Si la media rápida (9 días) cruza hacia arriba la lenta (20 días) es común que la acción suba ⚡ a corto plazo.
- 📉 Si la media rápida (9 días) cruza hacia abajo la lenta (20 días) es común que la acción baje 🙏 a corto plazo.

Esto es precisamente lo que expresamos en el siguiente código. Con `diff` buscamos el cambio de cuando una media pasa a estar por encima o debajo de la otra.

```
apple['Signal'] = np.where(apple['9_MA'] > apple['20
        _MA'], 1.0, 0.0)
apple['Position'] = apple['Signal'].diff()
```

Y ahora que tenemos todos los datos podemos representarlo. Incluimos en la gráfica el precio de la acción de Apple, las medias y los indicadores de compra y venta que hemos calculado.

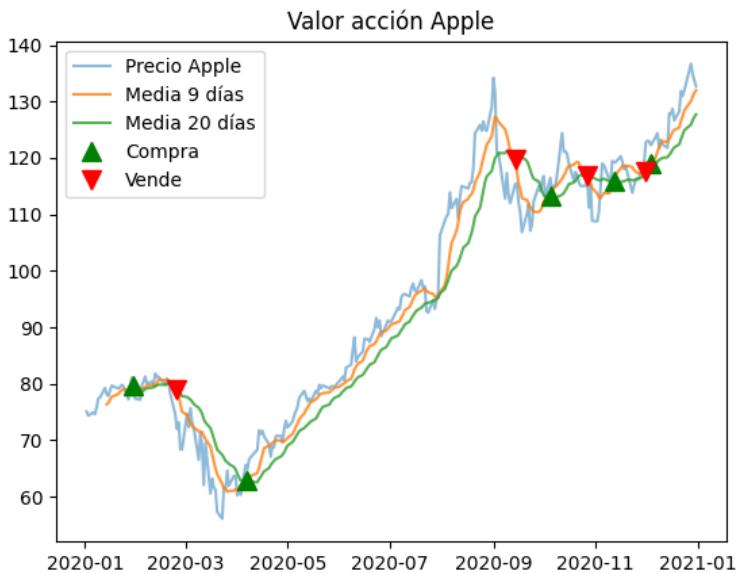
```
plt.figure()
```

```
plt.plot(apple['Close'], label='Precio Apple', alpha=0.5)
plt.plot(apple['9_MA'], label='Media 9 días', alpha=0.8)
plt.plot(apple['20_MA'], label='Media 20 días', alpha=0.8)

plt.plot(apple.index,
         np.where(apple['Position'] == 1, apple['9_MA'],
                  np.nan),
         '^', markersize=10, color='g', lw=0, label='Compra')

plt.plot(apple.index,
         np.where(apple['Position'] == -1, apple['9_MA'],
                  np.nan),
         'v', markersize=10, color='r', lw=0, label='Vende')

plt.title('Valor acción Apple')
plt.legend(loc='best')
plt.show()
```



### Ejercicios:

- Añade una nueva gráfica que muestre el retorno diario en porcentaje de Apple. Es decir, cuánto subió o bajó cada día.

## 8.32 Programación asíncrona con aiohttp

En este ejemplo vemos como realizar peticiones de manera asíncrona a una API externa. En concreto veremos como usar la API de Github para obtener el número de seguidores de diferentes cuentas. Este tipo de peticiones se puede realizar de dos formas:

- Síncrona: Enviamos una petición y no enviamos la siguiente hasta que la anterior no ha acabado. Suele ser lo más lento ya que mientras el servidor nos contesta nos quedamos parados sin hacer nada.

- 
- 🐛 Asíncrona: Enviamos todas las peticiones a la vez y esperamos a que nos devuelven respuesta. Suele ser más rápido ya que realizamos el procesado en paralelo.

🐌 La forma más sencilla es hacerlo de manera síncrona usando `requests`. Simplemente hacemos una petición HTTP a la API y tomamos el campo `followers`. Hemos simplificado la gestión de errores. Dependiendo de tu caso tal vez quieras manejar las excepciones.

```
import requests

url_base = "https://api.github.com/users/{}"

def get_seguidores(usuario):
    url = url_base.format(usuario)
    r = requests.get(url)
    return usuario, r.json()["followers"]
```

Y ahora podemos obtener los followers de tres usuarios distintos. Puedes de hecho acceder al mismo contenido si pones lo siguiente en tu navegador:

- `api.github.com/users/python`
- `api.github.com/users/google`
- `api.github.com/users.firebaseio`

```
for usuario in ["python", "google", "firebase"]:
    usuario, followers = get_seguidores(usuario)
    print(f"Followers de {usuario}: {followers}")

# Followers de python: 22840
# Followers de google: 46406
# Followers de firebase: 4113
```

Pero esta solución hace todas las peticiones de manera secuencial. Hace petición, espera, siguiente petición, espera, etc. Esto hace que no sea eficiente. Con cientos de usuarios sería muy lento.

---

💡 Para mejorarlo podemos hacerlo de manera asíncrona con `asyncio` y `aiohttp`. De esta manera podemos lanzar varias peticiones a la vez y después esperar. Esto es mas eficiente porque las peticiones se realizan en paralelo.

```
import asyncio
import aiohttp

url_base = "https://api.github.com/users/{}"
usuarios = ["python", "google", "firebase"]

async def get_seguidores(session, usuario):
    url = url_base.format(usuario)
    async with session.get(url) as r:
        return usuario, int((await r.json())["followers"])

async def get.todos(usuarios):
    async with aiohttp.ClientSession() as s:
        tasks = [get_seguidores(s, u) for u in usuarios]
        return await asyncio.gather(*tasks)

r = asyncio.run(get.todos(usuarios))
for usuario, rr in r:
    print(f"Followers de {usuario}: {rr}")
```

Si pruebas de las dos maneras veras como la forma asíncrona es mas rápida. Y cuantos mas grande sea `usuarios` mas se notará la diferencia.

📝 Ejercicios:

- Modifica `fetch_all` para que procese un máximo de `MAX_TASKS` a la vez. Esto puede ser necesario ya que si tenemos millones de usuarios no es buena idea lanzar millones de funciones asíncronas a la vez.
- Gestiona todas las excepciones que puedan ocurrir en el caso

---

asíncrono. Si por ejemplo una petición falla, gestíonalo de tal forma que no se eche todo a perder.

### 8.33 Crear baraja de Poker con `itertools`

En este ejemplo vemos como crear una baraja de Poker usando `itertools`. Aunque la podríamos crear definiendo una a una las 52 cartas de la baraja de Poker, podemos hacerlo de manera más sencilla sabiendo lo siguiente:

- ♠ ♥ ♣ ♦ Hay 4 palos : Picas, corazones, tréboles y diamantes.
- 🎭 Hay 13 números : El A , números del 2 al 10 y J , Q , K .

Las 52 cartas resultantes son simplemente las combinaciones de dos elementos de ambos conjuntos. Esto se conoce también como producto cartesiano. Lo podemos hacer de la siguiente manera.

```
import itertools

palos = ['Picas', 'Corazones', 'Tréboles', 'Diamantes']
numeros = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']

baraja = list(itertools.product(numeros, palos))

for carta in baraja:
    print(f"{carta[0]} de {carta[1]}

# A de
# A de
# A de
# A de
# ...
```

---

Como puedes ver en `baraja` ahora tenemos las `52` cartas, generadas como combinaciones de los números y palos.

💡 Ejercicios:

- Escribe una función `producto` que tenga el mismo comportamiento que `product` pero que no use el paquete `itertools`.
- Modifica el código para generar la baraja española. Esta tiene números del `1` al `10` y oros, copas, espadas y bastos como palos.

### 8.34 Barajar cartas con shuffle

En este ejemplo vemos como barajar una `list` que contiene las cartas `52` de Poker. Entendemos por barajar el mezclarlas de forma aleatoria

Empezamos definiendo la `baraja` de Poker. El uso de `product` nos permite generar todas las cartas pero están ordenadas de `A` a `K` y de `Picas` a `Diamantes`.

```
import itertools
import random

palos = ['Picas', 'Corazones', 'Tréboles', 'Diamantes']
numeros = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']
baraja = list(itertools.product(numeros, palos))
```

Vamos entonces a definir una función `baraja_cartas` que las baraje. Hay muchas formas de hacer esto, siendo una de ellas el algoritmo de *Fisher-Yates*. Sin embargo este tiene muchas variantes dependiendo de la eficiencia, si baraja la propia entrada o crea una nueva, etc.

En nuestro caso vamos a implementarlo de la manera mas sencilla:

- 
-  Iteramos nuestra baraja de fin a principio.
  -  Para cada carta tiramos un dado para generar un número aleatorio entre `0` y la carta en la que estamos.
  -  Cambiamos la carta de posición.

```
def baraja_cartas(b):  
    n = len(b)  
    for i in range(n - 1, 0, -1):  
        j = random.randint(0, i)  
        b[i], b[j] = b[j], b[i]
```

Y la podemos usar de la siguiente manera. Es importante notar que nuestra función modifica la propia `baraja`. Es decir, no crea una nueva variable barajada.

```
baraja_cartas(baraja)  
  
for carta in baraja:  
    print(f"{carta[0]} de {carta[1]}")
```

Aunque es importante saber como funcionan las cosas, si quieres barajar lo más práctico es que uses la función `shuffle` que viene con el paquete `random`.

```
random.shuffle(baraja)
```

### Ejercicios:

- Escribe un programa que realice cientos de simulaciones usando la función `baraja_cartas` sobre la baraja ordenada. Mide como de probable es que una carta acabe en diferentes posiciones. Concluye si la función es verdaderamente aleatoria.

---

### 8.35 Ordenar con bubble sort

A continuación vemos como ordenar una lista de menor a mayor. Como siempre decimos, en la práctica es mejor que uses una función ya hecha. Sin embargo es útil saber lo básico sobre los algoritmos de ordenación. Esto te ayudará a saber cual elegir.

Simplificando las cosas, los algoritmos de ordenación se reducen a dos cosas:

- ⌚ Velocidad: El tiempo que tardan en ejecutarse.
- 🧠 Memoria: La memoria que consumen.

Algo también muy importante es lo que se conoce como *Big O Notation*. Esto mide como se comportan los algoritmos (en términos de velocidad y memoria) a medida que se usan con entradas mas grandes. Algunos ejemplo para entenderlo:

- 🥇  $O(1)$  El tiempo en ejecutarse es el mismo sin importar el tamaño de la entrada.
- 🥈  $O(\log n)$  El tiempo crece lentamente, con el logaritmo.
- 🥉  $O(n)$  El tiempo crece proporcionalmente.
- 😢  $O(n^2)$  El tiempo crece de manera cuadrática.

Veamos algunos ejemplos para entender esto. Imagina un algoritmo que:

- ⌚ Tarda  $1$  segundo en ejecutarse.
- ➡ Para una entrada de tamaño  $10$ .

Imagina ahora que usamos una entrada del doble de tamaño. La entrada pasa de  $10$  □  $20$ . Dependiendo del  $O()$  de nuestro algoritmo, el tiempo se verá incrementado de forma diferente:

- Si es  $O(1)$  pasamos de  $1$  segundo a  $1$  segundo. El mismo tiempo.

- Si es  $O(\log n)$  pasamos de 1 segundo a 1.3 segundos. No está mal.
- Si es  $O(n)$  pasamos de 1 segundo a 2 segundos. Sigue sin estar mal.
- Si es  $O(n^2)$  pasamos de 1 segundo a 4 segundos. El tiempo empieza a incrementar notablemente.

Por esto es tan importante saber de que tipo es nuestro algoritmo. Dicho esto, vamos a implementar un algoritmo de ordenación conocido como *bubble sort*. Este es muy sencillo de implementar y tiene un  $O(n^2)$  en el peor de los casos. No es el mejor, pero si el mas sencillo y muy didáctico. Su funcionamiento es sencillo.:

- 🔔 Toma cada elemento y lo compara con el resto.
- ➡️ Si es mayor se desplaza al siguiente.
- 🔄 En cada nueva iteración, iteramos un elemento menos, ya que los del final van quedando ordenados.
- 📈 Tras realizar todas las iteraciones, tenemos 1 ordenado.

```
def bubble_sort(l):
    n = len(l)
    for i in range(n):
        for j in range(0, n-i-1):
            if l[j] > l[j+1]:
                l[j], l[j+1] = l[j+1], l[j]
                # Pista: Puede optimizarse
    return l
```

Lo podemos usar de la siguiente forma.

```
l = [64, 34, 25, 12, 22, 11, 90]
l_ordenado = bubble_sort(l)
print("Ordenado:", l_ordenado)
# Ordenado: [11, 12, 22, 25, 34, 64, 90]
```

 Ejercicios:

- 
- En la función `bubble_sort` hemos dejado una pista. Este algoritmo puede optimizarse de manera muy sencilla para evitar hacer trabajo redundante. Lo puedes hacer en dos líneas de código. Optimízalo y explícalo.
  - Cambia la función `bubble_sort` para que ordene `[1]` de mayor a menor.

### 8.36 Convertir binario a decimal

Veamos como convertir un número de representación binaria de decimal. Veamos antes un ejemplo de dicha conversión:

- 🤖 El número `11011` en binario.
- ~~100~~ Es el número `27` en decimal.

Esta conversión se realiza sumando potencias de dos. En este caso  $2^4 + 2^3 + 2^1 + 2^0$ . La potencia  $2^2$  nos la saltamos porque hay un `0` en esa posición.

Si lo piensas, es muy similar a la representación decimal, con la diferencia de que la base es `2` en vez de `10`. Es decir, el número `27` es en realidad  $2 \cdot 10^1 + 7 \cdot 10^0$ , o lo que es lo mismo, `27`.

Podemos crear nuestra función de la siguiente manera. Simplemente vamos sumando `2` elevado a la potencia que toque usando el `digito` que es `0` o `1`.

```
def binario_a_decimal(binario):  
    decimal = 0  
    potencia = 0  
    for digito in reversed(binario):  
        if digito not in {'0', '1'}:  
            raise ValueError("Error")  
        decimal += int(digito) * (2 ** potencia)  
        potencia += 1
```

```
    return decimal
```

Y podemos usar la función de la siguiente manera.

```
binario = "11011"
decimal = binario_a_decimal(binario)
print(f"{binario} -> {decimal}")
# 11011 -> 27
```

💡 Ejercicios:

- Escribe una función `decimal_a_binario` que realice la conversión inversa, de decimal a binario.

### 8.37 Usar código C con cffi

En este ejemplo vemos como usar código C desde Python, gracias a `cffi`. Este paquete permite a Python actuar como *wrapper* de código escrito en C.

Por ejemplo, podemos escribir una función en C y exponerla para que pueda ser llamada desde Python. Esto puede ser útil cuando queramos la velocidad de C pero la simplicidad de Python. Hay de hecho muchos paquetes de Python que aunque tu veas código Python, por debajo el código es C. Por ejemplo, en `numpy`.

Por un lado definimos un código en C. Este es el código que queremos llamar desde Python. Creamos una función para determinar si un número es primo o no.

```
// mylib.c
#include <stdbool.h>
#include <math.h>

bool es_primo(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
```

```
if (n % 2 == 0 || n % 3 == 0) return false;

int limit = sqrt(n);
for (int i = 5; i <= limit; i += 6) {
    if (n % i == 0 || n % (i + 2) == 0) {
        return false;
    }
}
return true;
}
```

Ahora usamos `cffi`. Primero debemos compilar nuestro código en C para que pueda ser usado por Python. Esto solo hay que hacerlo una vez. Fíjate en el `compile`.

```
import cffi
import os

if not os.path.exists("mylib.o"):
    ffi = cffi.FFI()
    ffi.cdef("bool es_primo(int n);")
    ffi.set_source(
        "_mylib",
        '#include "mylib.c"')
    ffi.compile()
```

Una vez lo tenemos, podemos importar la librería y usar `es_primo`. El código que se ejecutará por debajo será el código C compilado. Pero en realidad lo que estás usando es código Python.

```
import _mylib
numero = 777
primo = _mylib.lib.es_primo(numero)
print(f"El {numero}{'' if primo else ' no'} es primo")
```

### Ejercicios:

- Crea una función en C `son_primos` que dada una lista de números devuelva otra lista de `bool` indicando si cada uno de los

---

números introducidos es primo o no. Llámala desde Python.

### 8.38 Martingala y apuestas con random

A continuación vemos como simular con `numpy` la martingala. Esta es una estrategia de apuestas que se popularizó en la ruleta y tiene la siguiente estructura:

- ⚡ Se comienza con una apuesta inicial. Se apuesta siempre al ●.
- ● Si se gana, se vuelve a la apuesta inicial.
- ● Si se pierde, se dobla la apuesta.

La esencia de esta estrategia es que cuando perdemos, doblamos la siguiente apuesta para intentar recuperar la cantidad perdida. En la teoría, suena bien.

Podemos implementar esta estrategia de la siguiente forma. Se define una cantidad `inicial`, una `apuesta`, una probabilidad de ganar `prob_ganar` y una cantidad `objetivo` después de la cual se para la simulación.

```
import random

def martingala(inicial, apuesta, prob_ganar, objetivo):
    :
    balance = inicial
    bet = apuesta

    while balance > 0 and balance < inicial + objetivo
        :
            rojo = random.random() < prob_ganar
            if rojo:
                balance -= bet
                bet *= 2
            else:
                balance += bet
```

```
    bet = apuesta

    if bet > balance:
        break

    return balance
```

Como puedes ver el código usa `random` para generar el evento. El `while` hace que continuamente se apueste mientras haya `balance` y no hayamos llegado al objetivo. Puedes ver los dos escenarios claramente definidos:

- ● Si sale `rojo` perdemos. Doblamos la apuesta `bet`.
- ● Si se gana `negro` ganamos. Volvemos a la apuesta inicial.

Ahora podemos usar nuestra función con unos parámetros concretos. Usamos un `balance` inicial de `1000` Euros, usando una apuesta inicial de `10` Euros, con una probabilidad de ganar del `48\%` y un objetivo de `2000` Euros. Vemos el resultado.

```
print(martingala(inicial=1000,
                  apuesta=10,
                  prob_ganar=0.48,
                  objetivo=2000))
```

Como puedes comprobar ejecutando la simulación múltiples veces, en la mayoría de las ocasiones se pierde. Esto es debido a que una racha de mala suerte hace que la apuesta crezca exponencialmente, y llegue un punto donde no se puede afrontar.

 Ejercicios:

- Modifica los valores para buscar una estrategia que reduzca el riesgo de bancarrota.

---

## 8.39 Temporizador con time

En algunas ocasiones realizamos una tarea que tarda un tiempo en completarse. En estos casos es importante notificar al usuario del progreso, indicando el tiempo que falta. Podemos definir una barra de progreso de la siguiente manera.

```
import time
import sys

def cuenta_atras(segundos):
    for queda in range(segundos, 0, -1):
        progreso = int((segundos - queda) / segundos * 30)
        barra = f"[{'=' * progreso}{' ' * (30 - progreso)}]"
        sys.stdout.write(f"\r{barra} {queda}s restantes...")
        sys.stdout.flush()
        time.sleep(1)
    print("\r[=====] Completado!")

cuenta_atras(10)
```

 Ejercicios:

- Modifica la función `cuenta_atras` para que en vez de actualizarse cada `1` segundos con el progreso se actualice mas frecuentemente.

## 8.40 Calcular impuestos

A continuación vemos como calcular los impuestos en un sistema por tramos. En muchos países los impuestos se definen por tramos, donde a cada uno aplica un impuesto diferente. Podemos definir los tramos en Python de la siguiente forma.

```
TRAMOS = [
    (12000, 0.19),
    (20000, 0.24),
    (30000, 0.30),
    (60000, 0.37),
    (float('inf'), 0.45)]
```

Esto significa que dependiendo del tramo, se aplicarán unos impuestos diferentes. Por ejemplo:

- 🌱 Para 10.000 sólo aplica el 19%. Total, 1.900 de impuestos.
- 🐾 Para 15.000 aplica el 19% sobre 12.000 y el 24% sobre 3.000. Total, 3.000 de impuestos.
- 💧 Para 100.000 aplican todos los tramos. 19% sobre 12000, 24% sobre 8000, 30% sobre 10000, 37% sobre 30000 y 45% sobre 40000. Total 36.300 de impuestos.

Ahora definamos una función para calcular el impuesto.

```
def impuesto(ingreso):
    impuesto = 0

    for i, (hasta, tax) in enumerate(TRAMOS):
        desde = TRAMOS[i - 1][0] if i > 0 else 0
        tramo = min(ingreso, hasta) - desde
        if tramo > 0:
            impuesto += tramo * tax
        if ingreso <= hasta:
            break

    return impuesto
```

Y la usamos.

```
total_impuesto = impuesto(ingreso = 100000)
print(f"El impuesto es {total_impuesto:.2f}")
# El impuesto es 36300.00
```

---

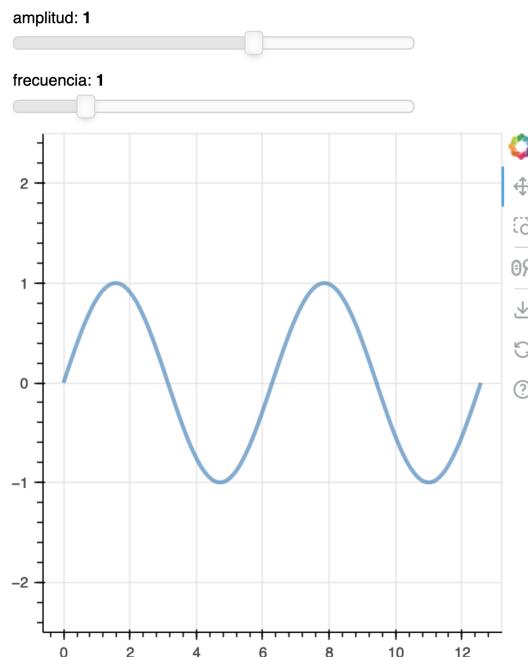
## Ejercicios:

- Modifica la función `impuesto` para que devuelva además del impuesto en Euros a pagar, el porcentaje que este supone del sueldo.

### 8.41 Plot interactivo con bokeh

Con `bokeh` podemos crear plots interactivos que se sirven como una página web. Es muy útil cuando quieres mostrar datos a un usuario como gráficas, y permitir al usuario que interactúe con ellas.

A continuación vemos como implementar lo siguiente. Un diagrama con la representación de una onda sinusoidal y dos *widgets* que permiten modificar los parámetros de la misma.



---

Importamos lo que necesitamos.

```
import numpy as np
from bokeh.io import curdoc
from bokeh.layouts import column, row
from bokeh.models import ColumnDataSource, Slider
from bokeh.plotting import figure
```

Definimos la onda sinusoidal. Con `linspace` podemos definir `N` puntos entre `0` y `2*PI`.

```
N = 200
x = np.linspace(0, 4*np.pi, N)
y = np.sin(x)
source = ColumnDataSource(data=dict(x=x, y=y))

plot = figure(height=400, width=400, y_range=[-2.5,
    2.5])
plot.line('x', 'y', source=source, line_width=3,
    line_alpha=0.6)
```

Creamos dos `Slider`. Estos permitirán al usuario modificar la amplitud y frecuencia interactivamente. El `on_change` nos permite decir que hacer cuando la `amplitud` o `frecuencia` cambien.

```
amplitud = Slider(title="amplitud", value=1.0, start
    =-5.0, end=5.0, step=0.1)
frecuencia = Slider(title="frecuencia", value=1.0,
    start=0.1, end=5.1, step=0.1)

def update_data(attrname, old, new):
    a = amplitud.value
    k = frecuencia.value

    x = np.linspace(0, 4*np.pi, N)
    y = a*np.sin(k*x)

    source.data = dict(x=x, y=y)

for w in [amplitud, frecuencia]:
```

```
w.on_change('value', update_data)
```

Creamos el plot con los dos `Slider` y el `plot`.

```
curdoc().add_root(  
    column(amplitud,  
           frecuencia,  
           plot, width=800))
```

Y con el siguiente comando podemos ejecutar. El resultado se podrá ver en la siguiente dirección:

- [🔗 http://localhost:5006](http://localhost:5006)

```
bokeh serve interactivo_bokeh.py
```

💡 Ejercicios:

- Implemente un botón de `reset` que al ser pulsado devuelva al `sin` a sus valores originales de `amplitud` y `frecuencia`.
- Modifica los rangos para que se pueda seleccionar una `amplitud` y `frecuencia` mayores en los `Slider`.

## 8.42 Simula hipotecas con matplotlib

A continuación vemos como calcular la cuota mensual y los intereses de una hipoteca, dadas las siguientes entradas:

- 💰 Cantidad de dinero prestada.
- 📈 Porcentaje de interés anual.
- 📆 Años de la hipoteca.

El objetivo final es saber cuanto tendremos que pagar al mes durante los años que dure la hipoteca y los intereses de la misma.

Definimos una función que calcula la cuota y los intereses totales. Esta función simplemente implementa la fórmula de la amortización.

---

```

def cuota_interes(prestamo, interes, anios):
    int_mensual = interes / 1200
    num_cuotas = anios * 12
    cuota = (prestamo * int_mensual) / (1 - (1 +
        int_mensual) ** -num_cuotas)
    intereses = ((cuota * num_cuotas - prestamo) /
        prestamo) * 100
    return (cuota, intereses)

```

Ahora la podemos usar con los siguientes parámetros:

- ⚡ Se quieren tomar prestados 100000 Euros.
- 📈 Con un 2% de interés anual.
- 🕰️ Durante 10 años.

Con esto podemos ver que la cuota mensual de 920 Euros y los intereses son del 10%. Es decir, después de los 10 años habremos pagado un 10% de lo que pedimos prestado.

```

cuota, intereses = hipoteca(100000, 2, 10)
print(f"Cuota mensual: {cuota:.2f} €")
print(f"Intereses: {intereses:.2f} %")
# Cuota mensual: 920.13 €
# Intereses: 10.42 %

```

Podemos realizar el cálculo para varios plazos. Desde 5 años hasta 30.

```

prestashop = 100000
interes = 2
plazos = range(5, 31, 5)

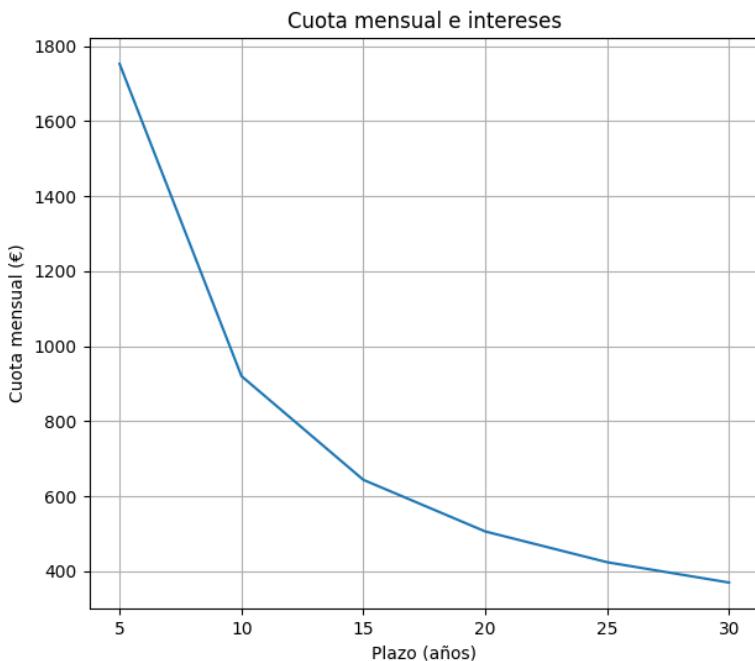
cuotas = [hipoteca(prestado, interes, plazo)[0] for
          plazo in plazos]

fig, ax1 = plt.subplots(figsize=(7, 6))
ax1.set_xlabel("Plazo (años)")
ax1.set_ylabel("Cuota mensual €()")
ax1.plot(plazos, cuotas, label="Cuota mensual €()")

```

```
plt.title("Cuota mensual e intereses")
plt.grid(True)
plt.show()
```

Y vemos la cuota mensual que nos quedaría en cada caso.



### Ejercicios:

- Como podemos ver, la cuota mensual se reduce considerablemente cuando aumentamos el plazo de la hipoteca. Sin embargo esto tiene un efecto en los intereses que se pagan. Añade una gráfica donde se represente los intereses pagados y el plazo en años.

---

## 8.43 Palabras El Quijote con wordcloud

A continuación vemos como crear una nube de palabras o *wordcloud* con las palabras de uno de los libros más leídos mundialmente. Don Quijote de la Mancha, escrito por Miguel de Cervantes en 1605.

Una nube de palabras representa:

- 💬 Las palabras presentes en un texto.
- 🌟 Donde las palabras más usadas se representan de manera más grande.

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

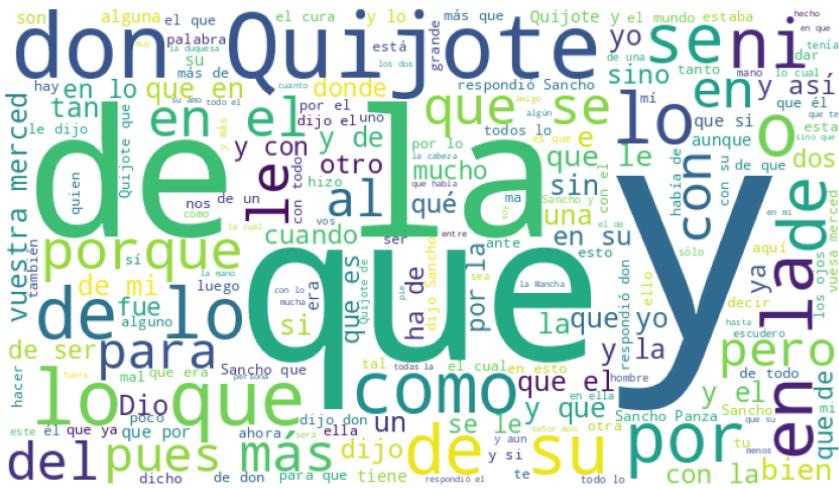
with open("quijote_wordcloud.txt", "r", encoding="utf-8") as file:
    quijote = file.read()

wc = WordCloud(background_color="white", width=700,
               height=400)
wc.generate(quijote)

plt.figure(figsize=(10, 5))
plt.imshow(wc)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Como podemos ver en la siguiente imagen, las palabras mas comunes son preposiciones, conjunciones o artículos, pero también tenemos algunas interesantes:

- Qujote
- Vuestra merced
- Sancho Panza



## Ejercicios:

- Como puedes observar, las palabras más comunes son `y`, `que` o `la`. Esto es común en cualquier texto ya que las conjunciones, preposiciones y artículos son usadas en cualquier. Pero no dicen mucho. Usa el argumento `stopwords` para ignorar estas palabras.

## 8.44 Programas ejecutables con pyinstaller

Uno de los problemas de Python es que para ejecutar un código necesitas tener instalado Python. Y no sólo eso. También necesitas instalar las dependencias como `numpy`, `pandas` o las que uses.

Siquieres distribuir tu código a un usuario final que no sepa de Python ni de programación, obligarle a instalar todo esto añade mucha fricción.

Por suerte `pyinstaller` nos permite generar programas ejecutables.

---

Es decir, que puedan ser ejecutados sin tener Python instalado. Veamos como usarlo.

Vamos a empezar creando un ejemplo muy sencillo. Este código pide al usuario su nombre y tras introducirlo muestra un `print` por pantalla.

```
# ejecutables_pyinstaller.py
name = input("Dime tu nombre: ")
print(f"Hola {name}")
```

Usando `pyinstaller` puedes crear un ejecutable.

- ⚠ Es importante tener en cuenta que si usas Windows, el ejecutable será para Windows. Si usas Mac, lo mismo. Desafortunadamente `pyinstaller` no permite la *cross-compilation*, es decir, generar un ejecutable de Windows desde Mac.

Puedes generar el ejecutable de la siguiente manera.

```
pyinstaller --onefile --clean ejecutables_pyinstaller.py
```

Y ya tienes el programa. Este puede ser ejecutado sin tener Python. Lo puedes ejecutar de dos formas:

- 💻 Desde el terminal con `./dist/ejecutables_pyinstaller`. Para usuarios avanzados.
- 🖱 Como cualquier otro programa, haciendo doble click en el. Para usuarios sin conocimientos de programación.

### Ejercicios:

- Para hacerlo más profesional, añade un ícono a tu ejecutable.

## 8.45 Calcular interés compuesto

El interés compuesto nos permite determinar la evolución del valor de algo a lo largo del tiempo, asumiendo un incremento anual determinado. Es decir, tenemos:

- 💰 Un **principal**. El valor inicial. Este puede ser un dinero.
- 📈 Un **interés** medido en **\%** anual.
- ⏳ Un **tiempo** medido en **años**.

```
def interes_compuesto(principal, interes, tiempo):  
    return principal * (1 + interes) ** tiempo
```

Con los siguientes parámetros, puedes ver que **1000** Euros al **5\%** anual en **5** años se convierten en **1276**.

```
principal = 1000  
interes = 0.05  
tiempo = 5  
  
interes_final = interes_compuesto(principal, interes,  
                                   tiempo)  
print(f"{principal} -> {interes_final:.0f} en {tiempo}  
      años")  
# 1000 -> 1276 en 5 años
```

💡 Ejercicios:

- Crea otra función **interés\_compuesto** que acepte un parámetro de entrada que sea **interés** pero en vez de ser fijo, es un **list** de **n** valores, siendo **n** el **tiempo**. Es decir, si **tiempo** es **5** entonces **interés** será una **list** con **5** elementos donde el primero indicará el interés del primer año, el segundo del segundo, etc.

---

## 8.46 Busca el número que falta

A continuación vamos a escribir una función que permite buscar los números que faltan en una lista.

- Si tienes una lista con 0236 .
- Se buscan los números que faltan, 145 .

Podemos definir la función de la siguiente manera. Se crea la secuencia completa entre el menor y el mayor número, indicando cuales son todos los esperados. Después vemos si cada uno de ellos está en nums . Si no está, lo añadimos a faltan .

```
def encuentra_numeros(nums):  
    min_num = min(nums)  
    max_num = max(nums)  
    faltan = []  
    for i in range(min_num, max_num + 1):  
        if i not in nums:  
            faltan.append(i)  
    return faltan
```

Ahora podemos usarla de la siguiente manera.

```
nums = [0, 2, 3, 6]  
faltan = encuentra_numeros(nums)  
print(faltan)  
# [1, 4, 5]
```

Ejercicios:

- Esta implementación es sencilla pero muy poco eficiente. Busca optimizarla para que tarde menos en buscar los números que faltan.

## 8.47 Comprimir información

Veamos como escribir un algoritmo sencillo que permite comprimir información. Imagina que tienes unos datos:

-  AAAAETTTTH : Tenemos una secuencia de datos.
-  A4E1T4H1 : Que puede ser comprimida de esta forma. Por ejemplo en vez de AAAA podemos comprimir a A4 .

Veamos como implementarlo en `comprime` . Iteramos toda nuestra secuencia `s` . Si el siguiente elemento `s[i+1]` es igual al actual, contamos cuantos iguales hay. Cuando ya no es igual, guardamos la información en `comprimido` .

```
def comprime(s):
    comprimido = []
    cuenta = 1

    for i in range(len(s)):
        if i + 1 < len(s) and s[i] == s[i + 1]:
            cuenta += 1
        else:
            comprimido.append((s[i], cuenta))
            cuenta = 1

    return comprimido
```

Podemos usar la función de la siguiente manera, y este es el resultado.

```
comprimido = comprime("AAAAETTTTH")
print(comprimido)
# [('A', 4), ('E', 1), ('T', 4), ('H', 1)]
```

Es importante notar que este algoritmo funciona bien cuando hay mucha información repetida. En el caso contrario, puede ser hasta peor como en el siguiente ejemplo:

- 
-  `ABCDEF` : La secuencia no tiene nada de redundancia, es decir, no se repite nada.
  -  `A1B1C1D1E1F1` : Nuestro algoritmo de compresión da una salida que es en realidad mayor. No nos sirve.

### Ejercicios:

- Modifica la función `comprime` para que en vez de devolver una lista de `tuple` devuelva un `str` con la misma información.
- Implementa una función `descomprime` que realice lo contrario. Esto es, dada una secuencia comprimida, la descomprima a la original.

## 8.48 Interfaz de usuario con PyQt

En algunas ocasiones es importante añadir un interfaz de usuario a nuestros programas, para que puedan ser usado por un público no necesariamente técnico.

Podemos hacer esto con `pyqt` . En el siguiente ejemplo vemos como crear un interfaz sencillo que tiene:

-  Un contador
-  Un botón que al pulsarlo incrementa el contador en `1` .
-  Una sorpresa cuando el contador llega a `100` .

Vamos a usar los siguiente componentes:

- `QHBoxLayout` nos permite organizar los elementos en la pantalla.
- `QLabel` nos permite añadir un texto con una etiqueta al interfaz.
- `QPushButton` permite crear un botón con un comportamiento definido con `clicked.connect` .

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget,
    QHBoxLayout, QPushButton, QLabel

class AppContador(QWidget):
    def __init__(self):
        super().__init__()
        self.contador = 0

        self.setWindowTitle("Contador")
        self.setGeometry(100, 100, 300, 200)

        layout = QHBoxLayout()

        self.label = QLabel("0", self)
        self.label.setStyleSheet("font-size: 30px;")

        self.button = QPushButton("Incrementar", self)
        self.button.setStyleSheet("""
            font-size: 20px;
            border-radius: 20px;
            background-color: #4CAF50;
            color: white;
            padding: 10px;
            border: 2px solid #3E8E41;
        """)
        self.button.clicked.connect(self.suma_contador)

        layout.addWidget(self.label)
        layout.addWidget(self.button)

        self.setLayout(layout)

    def suma_contador(self):
        self.contador += 1
        if self.contador == 100:
            self.label.setText("")
            self.contador = 0
        else:
```

```
self.label.setText(str(self.contador))
```

Una vez tenemos la aplicación, la podemos ejecutar de la siguiente forma.

```
if __name__ == "__main__":
    app = QApplication(sys.argv)

    w = AppContador()
    w.show()

    sys.exit(app.exec())
```

### 💡 Ejercicios:

- Usa `pyinstaller` para poder distribuir este aplicación a usuarios que no tengan Python, pudiendo ejecutarlo sin instalarlo.

## 8.49 Dashboard con streamlit

Con `streamlit` podemos crear paneles de mando o *dashboards* con diferentes representaciones gráficas. En este ejemplo usamos datos de la esperanza de vida en diferentes países, y permitimos al usuario seleccionar diferentes países para comparar su esperanza de vida.

Como puedes ver el código es bien sencillo:

- 📁 Primero cargamos los datos usando `pandas`. Realizamos un pequeño procesado de las columnas.
- 📈 Usamos `multiselect` para permitir al usuario seleccionar diferentes países.
- 📈 Usamos `line_chart` para representar gráficamente la esperanza de vida de los países seleccionados.

```
import streamlit as st
import pandas as pd
```

---

```

df = pd.read_csv("esperanza_vida.csv", skiprows=3)

st.title (" Esperanza de Vida Mundial")

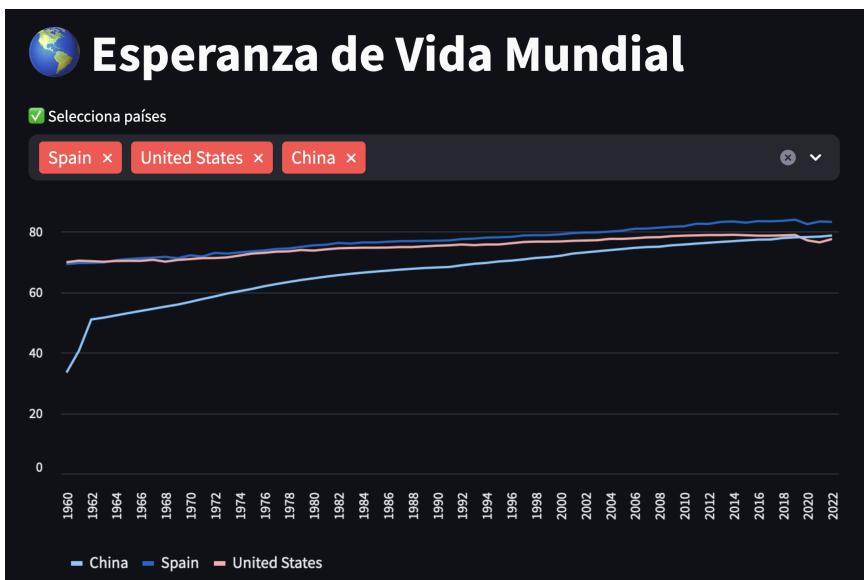
df = df[['Country Name'] + [str(year) for year in
    range(1960, 2023)]]
df.set_index('Country Name', inplace=True)

p = st.multiselect (" Selecciona países", df.index)
st.line_chart(df.loc[p].T)

```

Y podemos ejecutar el código de esta manera.

```
streamlit run dashboard_streamlit.py
```



Si en tu navegador introduces la siguiente dirección, podrás ver el resultado:

- [🔗 http://localhost:8501/](http://localhost:8501/)

---

## Ejercicios:

- Representa el histograma de la esperanza de vida de todos los países y como evoluciona a lo largo de los años.

### 8.50 Problema de la mochila

A continuación vemos como resolver el problema de la suma de subconjuntos. Este problema consiste en lo siguiente.

- 🎒 Tienes una mochila de un determinado tamaño.
- 💻 📝 📕 🎃 💡 💦 Y un conjunto de objetos donde cada uno ocupa un determinado volumen.

El problema que tenemos es que no podemos meter todo en la mochila. Queremos por tanto saber que opciones tenemos para optimizar la mochila al máximo.

Este algoritmo nos permite buscar que objetos puedes meter en tu mochila sin que quede espacio libre, dandonos todas las combinaciones posibles.

Existen múltiples variantes de este algoritmo. Vemos la más didáctica aunque no la más eficiente.

```
from itertools import combinations

def obtiene_combinaciones(objetos, mochila):
    result = []
    pesos = list(objetos.values())
    nombres = list(objetos.keys())

    for r in range(1, len(pesos) + 1):
        for s_i in combinations(range(len(pesos)), r):
            s_p = [pesos[i] for i in s_i]
            if sum(s_p) == mochila.get(''):
                s_n = [nombres[i] for i in s_i]
```

```
    result.append(s_n)
return result
```

Ahora damos unos valores a nuestros objetos. Estos indican lo que ocupa cada objeto en la mochila:

- Ordenador: 5
- Bolígrafo: 1
- Libreta: 5
- Peluche: 3
- Linterna: 2
- Chanclas: 4

Por otro lado tenemos la mochila, que tiene la siguiente capacidad.

- Mochila: 14

Como puedes ver no entra todo en la mochila. Si sumamos lo que ocupan todos los objetos tendríamos 20 pero en la mochila solo entra 14 .

Usando nuestra función `obtiene_combinaciones` obtenemos todas las combinaciones posibles de lo que podríamos llevar.

```
objetos = {'ordenador': 5,
           'boli': 1,
           'libreta': 5,
           'peluche': 3,
           'linterna': 2,
           'chanclas': 4}

mochila = 14

combinaciones = obtiene_combinaciones(objetos, mochila
)
for i in combinaciones:
    print(i)
```

---

Estas son las combinaciones resultantes. Puedes comprobar que todas suman 14 , la capacidad de la mochila:

- 
- 
- 
- 

### Ejercicios:

- Como puedes ver nuestro algoritmo es muy ineficiente. Busca una forma de optimizarlo. Dejamos para los usuarios más avanzados la opción de usar programación dinámica.
- Existe un problema parecido más completo llamado *knapsack*. Este nos permite tener en cuenta no sólo el tamaño de nuestros objetos sino su importancia. Si vamos a la playa, tal vez las chanclas sean más importantes. Este algoritmo nos permite introducir este concepto de importancia, asignando un peso a cada elemento. Implementa una nueva función que asigne una importancia a cada elemento y usa el algoritmo de *knapsack* para resolverlo.

---

## 9 Top 100 Errores Comunes



A continuación vemos una recopilación de los errores más comunes en Python. Decimos que hay un error cuando:

- **Bug**: Existe un fallo de lógica o vulnerabilidad.
- **Pythonic**: Existe una forma de hacerlo más propia de Python.
- **Velocidad**: Se puede escribir siendo su tiempo de ejecución menor.
- **Simplicidad**: Se puede simplificar la lógica.

Pero no existe una verdad absoluta. Establece tu propio criterio en función a tu caso. Lo mas importante es que el código funcione y resuelva tu problema. En los casos donde sea opinable la mejor forma de hacerlo, es mejor centrarse en que funciona que en interminables luchas alejadas de resolver el problema real.

## 9.1 Listas: Acceder último elemento

✗ Aunque se puede acceder al último elemento así.

```
lista = [1, 2, 3, 4]
print(lista[len(lista)-1]) # 4
```

✓ Es mejor hacerlo así.

```
print(lista[-1]) # 4
```

## 9.2 Listas: Append devuelve None

✗ La función `append` no devuelve nada, es decir `None`. Es común pensar que devuelve la lista con el nuevo elemento, pero no.

```
lista = [1, 2, 3]
lista = lista.append(4)
print(lista) # None
```

✓ Accede a `lista` para verla con el nuevo elemento añadido.

```
lista = [1, 2, 3]
lista.append(4)
print(lista) # [1, 2, 3, 4]
```

## 9.3 Listas: Copia referencia

✗ Ten cuidado con lo siguiente. Al modificar un valor de nuestra matriz, se modifican dos en realidad. Esto es porque no son realmente una copia, sino referencias a lo mismo.

```
matriz = [[0, 0]] * 2
print(matriz) # [[0, 0], [0, 0]]
```

```
matriz[0][0] = 99  
print(matriz) # [[99, 0], [99, 0]]
```

- ✓ Crea tu matriz de la siguiente forma. Así te aseguras de que son elementos independientes y no referencias de lo mismo.

```
matriz = [[0, 0], [0, 0]]  
matriz[0][0] = 99  
print(matriz) # [[99, 0], [0, 0]]
```

## 9.4 Listas: Confundir `append` y `extend`

- ✗ No confundas el `append` con el `extend`. El siguiente código da error porque `extend` está pensado para usarse con listas.

```
x = [1, 2, 3]  
x.extend(4)  
print(x) # TypeError
```

- ✓ Si deseas añadir un solo elemento, lo que necesitas es `append`.

```
x = [1, 2, 3]  
x.append(4)  
print(x) # x = [1, 2, 3]
```

## 9.5 Tuplas: De un valor

- ✗ Siquieres definir una `tuple` de un único valor, esto no funciona. Se define un `str`.

```
a = ("negro")  
print(type(a)) # <class 'str'>
```

- ✓ Deberás incluir una `,`. En este caso si tendremos una `tuple` de un solo valor.

```
a = ("negro",)
print(type(a)) # <class 'tuple'>
```

## 9.6 Diccionarios: Sustituto if

✗ En algunos casos una estructura `if` y `else` se puede reemplazar por un diccionario.

```
def saludo(idioma):
    if idioma == 'en':
        return 'Hello'
    elif idioma == 'es':
        return 'Hola'
    elif idioma == 'fr':
        return 'Bonjour'
    else:
        return 'Lenguaje no válido'

print(saludo('es')) # Hola
```

✓ De la siguiente manera.

```
def saludo(idioma):
    return {'en': 'Hello',
            'es': 'Hola',
            'fr': 'Bonjour'}
    }.get(idioma, 'Lenguaje no válido')

print(saludo('es')) # Hola
```

## 9.7 Diccionarios: Unir diccionarios

✗ Si quieres unir dos diccionarios, el operador `+` no está definido.

```
dict1 = {'a': 1}
dict2 = {'b': 2}
```

```
print(dict1 + dict2) # TypeError: unsupported operand  
                      type(s)
```

✓ Pero los puedes unir con `|`.

```
dict1 = {'a': 1}  
dict2 = {'b': 2}  
print(dict1 | dict2) # {'a': 1, 'b': 2}
```

## 9.8 If: Orden de operadores

✗ Ten cuidado con el orden de los operadores. Este ejemplo será siempre `True` y seguramente no sea lo que buscas.

```
if x == 5 or 6:  
    print("Es 5 o 6")
```

✓ Lo que buscas es esto.

```
if x == 5 or x == 6:  
    print("Es 5 o 6")
```

## 9.9 If: Paréntesis innecesario

✗ No hace falta definir dos tramos con paréntesis para verificar si la edad está en el rango `(12, 19)`.

```
edad = 15  
if (edad > 12) and (edad < 19):  
    print("Es un adolescente")
```

✓ Podemos reescribirlo de la siguiente manera, más fácil de leer.

```
edad = 15  
if 12 < edad < 19:  
    print("Es un adolescente")
```

---

## 9.10 If: Condición con booleano

✗ Si tenemos un `valor` que queremos usar como condición es habitual hacer lo siguiente, viendo si es `True` para actuar en consecuencia.

```
valor = True
if valor == True:
    print("valor es True")
```

✓ Pero lo podemos simplificar a lo siguiente. Lo mismo pero más sencillo.

```
valor = True
if valor:
    print("valor es True")
```

## 9.11 If: Condición con lista

✗ Si queremos evaluar si la lista está vacía podemos usar `len()` y ver si es `0`.

```
mi_lista = []
if len(mi_lista) == 0:
    print("La lista está vacía")
```

✓ Sin embargo no es necesario, podemos hacer lo siguiente. Es más *Pythonic*.

```
mi_lista = []
if not mi_lista:
    print("La lista está vacía")
```

---

## 9.12 If: Usar el operador ternario

✗ Si queremos escribir una función que devuelve si un número es par o no, lo podemos hacer así.

```
def es_par(a):
    if a % 2 == 0:
        return "Par"
    else:
        return "Impar"
```

✓ Pero podemos hacer uso del operador ternario para resumirlo todo en una línea. No siempre es mejor, pero a tener en cuenta.

```
def es_par(a):
    return "Par" if a % 2 == 0 else "Impar"
```

## 9.13 If: Ordenar checks

✗ Imagina que tienes dos condiciones. Una más rápida de verificar y la otra más lenta. Entraremos en el `if` si ambas condiciones se cumplen.

```
def check_barato():
    print("Ejecuta check_barato")
    return False

def check_caro():
    print("Ejecuta check_caro")
    return True

if check_caro() and check_barato():
    print("Entra")

# Ejecuta check_caro
# Ejecuta check_barato
```

- 
- ✓ Es mejor ejecutar antes el `check_barato`. Si este es `False` Python ya no ejecutará el otro `check`, y nos ahorraremos ejecutar el `check_caro`.

```
if check_barato() and check_caro():
    print("Entra")
# Ejecuta check_barato
```

## 9.14 If: Operador walrus

- ✗ Si tenemos un `resultado` y queremos usarlo como condición, lo podemos hacer de la siguiente manera.

```
resultado = mi_funcion()
if resultado:
    print(resultado)
```

- ✓ Pero gracias al operador `:=`, el operador *walrus* de Python nos podemos ahorrar una línea. *Walrus* significa morsa, y es lo que `:=` simboliza siendo los ojos y dientes. No siempre es mejor así, pero un recurso interesante.

```
if resultado := mi_funcion():
    print(resultado)
```

## 9.15 Match: Usa match

- ✗ Si tienes múltiples condiciones en un `if` donde varias tienen el mismo resultado, no hagas esto.

```
def permisos(rol):
    if rol == 1:
        return "todos"
    elif rol == 2:
        return "todos"
```

```
    elif rol == 3:
        return "alguno"
    elif rol == 4:
        return "ninguno"
    else:
        return "ninguno"

print(permisos(1)) # todos
print(permisos(2)) # todos
```

✓ Mejor aprovecha el potencial de `match` .

```
def permisos(rol):
    match rol:
        case 1 | 2:
            return "todos"
        case 3:
            return "alguno"
        case _:
            return "ninguno"
```

## 9.16 Operadores: Confundir orden

✗ Al igual que en matemáticas el operador `*` se evalúa primero.

```
print(6 + 6 * 2) # 18
```

✓ Si quiere que la suma vaya primero, usa paréntesis.

```
print((6 + 6) * 2) # 24
```

## 9.17 Operadores: Confundir and

✗ No confundas el operador `&` con el `and` . Si buscas el operador lógico, no uses `&` , ya que este es el operador *bitwise*. Esto no es lo que buscas.

```
if True & False:  
    print("Entra")
```

✓ Para el operador lógico usa `and`.

```
if True and False:  
    print("Entra")
```

## 9.18 Operadores: Usa `in`

✗ Si queremos saber si una letra es vocal podemos ver si es cualquiera de las vocales.

```
def es_vocal(letra):  
    if letra == 'a' or letra == 'e' or letra == 'i' or  
        letra == 'o' or letra == 'u':  
            return True  
    return False  
  
print(es_vocal("a")) # True  
print(es_vocal("z")) # False
```

✓ Pero podemos simplificarlo de la siguiente manera. Dado que una cadena como `aeiou` es iterable, podemos usar `in`.

```
def es_vocal(letra):  
    if letra in 'aeiou':  
        return True  
    return False
```

## 9.19 Bucles: Iterar elementos

✗ Siquieres iterar una lista para acceder a todos sus elementos, no lo hagas como se hace en otros lenguajes de programación.

```
nombr es = ['Ana', 'Juan', 'Pedro']
for i in range(len(nombr es)):
    print(nombr es[i])

# Ana, Juan, Pedro
```

✓ Es mejor hacerlo con iteradores. Es más *Pythonic*.

```
nombr es = ['Ana', 'Juan', 'Pedro']
for nombre in nombr es:
    print(nombre)

# Ana, Juan, Pedro
```

## 9.20 Bucles: Iterar índices y elementos

✗ Si quieres acceder a cada elemento y además a su índice, no lo hagas como en otros lenguajes de programación.

```
nombr es = ['Ana', 'Juan', 'Pedro']
indice = 0
for nombre in nombr es:
    print(f"indice={indice} nombre={nombre}")
    indice += 1
```

✓ Usa `enumerate` para ahorrarte la variable `indice`. Más sencillo, menos propenso a errores y como no, más Pythonic.

```
nombr es = ['Ana', 'Juan', 'Pedro']
for indice, nombre in enumerate(nombr es):
    print(f"indice={indice} nombre={nombre}")

# indice=0 nombre=Ana
# indice=1 nombre=Juan
# indice=2 nombre=Pedro
```

---

## 9.21 Bucles: Iterar al revés

✗ Si quieres iterar una lista al revés, no hagas esto.

```
colores = ["rojo", "verde", "azul"]

for i in range(len(colores) - 1, -1, -1):
    print(colores[i])
```

✓ Usa `reversed`.

```
colores = ["rojo", "verde", "azul"]

for color in reversed(colores):
    print(color)
```

## 9.22 Bucles: Iterar dos listas

✗ Si quieres iterar dos listas a la vez, no lo hagas así.

```
nombres = ['Ana', 'Beatriz', 'Carlos']
edades = [25, 30, 35]
for i in range(len(nombres)):
    print(f"{nombres[i]} tiene {edades[i]} años")
```

✓ Usa `zip()`. Es más *Pythonic*. No olvides que si no son del mismo tamaño, se usará la más corta.

```
nombres = ['Ana', 'Beatriz', 'Carlos']
edades = [25, 30, 35]
for nombre, edad in zip(nombres, edades):
    print(f"{nombre} tiene {edad} años")

# Ana tiene 25 años
# Beatriz tiene 30 años
# Carlos tiene 35 años
```

---

## 9.23 Bucles: Iterar y modificar Listas

✗ No modifiques una lista mientras la iteras. Pasan cosas raras. Si eliminás un elemento los índices cambian y puedes tener comportamientos indeseados y difíciles de detectar.

```
lista = [1, 2, 3, 4, 4, 4, 4, 5]
for x in lista:
    if x == 4:
        lista.remove(x)

print(lista) # [1, 2, 3, 4, 5]
```

✓ Si quierés filtrar elementos, hazlo así. En este caso filtramos el 4 correctamente.

```
lista = [1, 2, 3, 4, 4, 4, 4, 5]
print([i for i in lista if i != 4])
# [1, 2, 3, 5]
```

## 9.24 Bucles: Iterar diccionarios

✗ Si quierés iterar los *key-value* de un diccionario no hagas esto.

```
mi_diccionario = {'a': 1, 'b': 2, 'c': 3}
for clave in mi_diccionario.keys():
    print(clave, mi_diccionario[clave])
```

✓ Usa `items` para acceder a la `clave` y `valor`.

```
mi_diccionario = {'a': 1, 'b': 2, 'c': 3}
for clave, valor in mi_diccionario.items():
    print(clave, valor)
```

## 9.25 Bucles: Uso else en for

✗ Si buscas un número en una lista y quieres saber si se ha encontrado o no, lo podrías hacer de la siguiente manera.

```
numbers = [1, 2, 3, 4, 5]
found = False

for num in numbers:
    if num == 7:
        print("Se encontró!")
        found = True
        break

if not found:
    print("No se encontró!")
```

✓ Pero puedes ahorrar alguna línea usando `for-else`. La sección `else` se ejecutará si el bucle termina sin encontrar el número. Como nota, en este caso es mejor usar `in`.

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 7:
        print("Se encontró!")
        break
else:
    print("No se encontró!")
```

## 9.26 Comprehension: Generar listas secuenciales

✗ Si quieres crear una lista con números consecutivos, puedes reducir esto.

```
numeros = []
for i in range(1, 4):
```

```
    numeros.append(i)
```

```
print(numeros) # [1, 2, 3]
```

- ✓ A una línea con *list comprehension*. Es más *Pythonic*.

```
numeros = [i for i in range(1, 4)]  
print(numeros) # [1, 2, 3]
```

## 9.27 Comprehensions: Modificar con list comprehensions

- ✗ Para crear una lista con los cuadrados de los números del 1 al 3, puedes hacerlo así.

```
cuadrados = []  
for i in range(1, 4):  
    cuadrados.append(i**2)  
  
print(cuadrados) # [1, 4, 9]
```

- ✓ Pero es más *Pythonic* usando las *list comprehension*.

```
cuadrados = [n**2 for n in range(1, 4)]  
print(cuadrados) # [1, 4, 9]
```

## 9.28 Comprehensions: Filtrar con list comprehensions

- ✗ Si quieres crear una lista de un conjunto de valores donde cada uno es transformado y filtrado según un criterio, puedes simplificar esto.

```
cuadrados_pares = []  
for i in range(1, 10):  
    if not i % 2:  
        cuadrados_pares.append(i**2)
```

```
print(cuadradores_pares) # [4, 16, 36, 64]
```

- ✓ En esto. En ambos casos obtenemos el cuadrado de los primeros 9 números pares. No olvides que a veces la forma anterior es más legible y puede llegar a ser mejor.

```
cuadradores_pares = [i**2 for i in range(1, 10) if not
                     i % 2]
print(cuadradores_pares) # [4, 16, 36, 64]
```

## 9.29 Funciones: Usar lambda

- ✗ Si tienes una función muy sencilla para usar con `sorted` no es necesario definir una función.

```
l = ['b', 'a', 'Z']
def a_lower(x):
    return x.lower()
print(sorted(l, key=a_lower)) # ['a', 'b', 'Z']
```

- ✓ Te puedes ahorrar alguna línea de código usando funciones lambda.

```
print(sorted(l, key=lambda x: x.lower())) # ['a', 'b',
                                             'Z']
```

## 9.30 Funciones: Uso de early return

- ✗ Definimos una función que busca el primer positivo en la lista. Sin embargo es poco eficiente, ya que aunque lo haya encontrado, sigue iterando todos los números.

```
def busca(numeros):
    encontrado = None
    for numero in numeros:
```

```
        if encontrado is None and numero > 0:
            encontrado = numero
    return encontrado

positivo = busca([-3, -2, 0, 5, 10])
print(positivo) # 5
```

- ✓ Haz uso del *early return*. Cuando hayas encontrado lo que buscas, deja de iterar y devuelve lo que estabas buscando.

```
def busca(numeros):
    for numero in numeros:
        if numero > 0:
            return numero
    return None

positivo = busca([-3, -2, 0, 5, 10])
print(positivo) # 5
```

### 9.31 Funciones: Función sin implementar

- ✗ Si tienes pendiente por implementar una función y la definas usando `pass`, puede ser peligroso. Alguien podría usarla sin saber que está sin implementar.

```
def funcion_no_implementada():
    pass
```

- ✓ Es mejor lanzar una excepción, para que si alguien la usa se dé cuenta de que no está implementada.

```
def funcion_no_implementada():
    raise NotImplementedError("No implementada")
```

---

## 9.32 Funciones: Evitar uso de global

✗ Evita el uso de variables globales, sobre todo si las modificas. Esto produce lo que se conoce como *side effects* y causa muchísimos problemas.

```
contador = 0

def incrementar():
    global contador
    contador += 1
```

✓ Mejor pasa el argumento, modifícalo y devuélvelo.

```
def incrementar(contador):
    return contador + 1

contador = 0
contador = incrementar(contador)
```

## 9.33 Funciones: Retorno de None

✗ No olvides que cualquier función devuelve `None` por defecto. En el siguiente caso si usamos un *roll* no tenido en cuenta, obtendremos `None` y esto podría causar problemas.

```
def puede_entrar(rol):
    if rol == "policia":
        return True
    elif rol == "visitante":
        return False

print(puede_entrar("rol_no_definido")) # None
```

✓ Mejor ser explícito y manejar todos los casos.

```
def puede_entrar(rol):
```

```
if rol == "policia":  
    return True  
elif rol == "visitante":  
    return False  
return False  
  
print(puede_entrar("rol_no_definido")) # False
```

### 9.34 Funciones: No usar la función main

✗ Aunque para *scripts* rápidos no se suele respetar, el siguiente código no es del todo correcto.

```
def main():  
    # tu código aquí  
    print("Hola Mundo")  
  
main()
```

✓ Ejecuta tu `main` sólo si se está en el `__main__`.

```
def main():  
    # tu código aquí  
    print("Hola Mundo")  
  
if __name__ == "__main__":  
    main()
```

### 9.35 Funciones: Anotaciones

✗ Para ejemplos sencillos el siguiente código es correcto.

```
def suma(a, b):  
    return a + b
```

✓ Pero si deseas hacer tu código más profesional, utiliza *type annotations* para indicar el tipo de cada argumento de entrada y salida.

---

Esto actúa como documentación y un *linter* puede utilizarlo para protegerte.

```
def suma(a: int, b: int) -> int:  
    return a + b
```

## 9.36 Generadores: Uso de sum

✗ Siquieres sumar todos estos elementos, te puedes ahorrar convertir la secuencia en una lista.

```
suma_pares = sum([x for x in range(1000000) if not x %  
                  2])  
print(suma_pares) # 249999500000
```

Puedes usar `sum` directamente sin el `[` y `]`.

```
suma_pares = sum(x for x in range(1000000) if not x %  
                  2)  
print(suma_pares) # 249999500000
```

## 9.37 Generadores: Confundir generadores con listas

✗ No confundas el `range`. Este apenas ocupa memoria.

```
lista = range(5)  
print(lista) # range(0, 5)
```

Con una lista. En este la `lista` si que ocupa memoria.

```
lista = range(5)  
print(list(lista)) # [0, 1, 2, 3, 4]
```

---

### 9.38 Generadores: Usar generador terminado

✗ Si creamos un generador, solo podemos iterarlo una vez. La segunda vez, como ya se ha llegado al final, no quedará nada por recorrer.

```
def cuenta_cuatro():
    for i in range(5):
        yield i

gen = cuenta_cuatro()
for i in gen:
    print(i) # 0 1 2 3 4

for i in gen:
    # No se imprime.
    # Nada que iterar.
    print(i)
```

✓ Deberás crear un generador nuevo.

```
for i in cuenta_cuatro():
    print(i) # 0 1 2 3 4

for i in cuenta_cuatro():
    print(i) # 0 1 2 3 4
```

### 9.39 Clases: Usar métodos estáticos

✗ Si tienes métodos genéricos que no actúan sobre el objeto.

```
class Utilidades:
    def generar_id():
        # Lógica para generar ID...
        return "XYZ123"
```

✓ Es posible que sea mejor definirlo como estático.

```
class Utilidades:  
    @staticmethod  
    def generar_id():  
        # Lógica para generar ID...  
        return "XYZ123"
```

## 9.40 Clases: Comparar objetos

✗ Ten cuidado si comparas dos objetos sin haber definido el criterio a utilizar.

```
class MiClase:  
    def __init__(self, valor):  
        self.valor = valor  
  
a, b = MiClase(1), MiClase(1)  
print(a == b) # False
```

✓ Si quieres poder comparar dos objetos tienes que definir `__eq__`. Establecemos el criterio de que si `valor` es igual, entonces los objetos son iguales.

```
class MiClase:  
    def __init__(self, valor):  
        self.valor = valor  
    def __eq__(self, otro):  
        if isinstance(otro, MiClase):  
            return self.valor == otro.valor  
        return False  
  
a, b = MiClase(1), MiClase(1)  
print(a == b) # True
```

## 9.41 Clases: No definir str

✗ Cuando definas una clase, si no defines el método `__str__` cuando lo muestres por pantalla obtendrás lo siguiente.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
c = Punto(1, 2)  
print(c) # # <__main__.Punto object at 0x100f3ff40>
```

✓ Define `__str__` para obtener información más significativa.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def __str__(self):  
        return f"Punto: x={self.x} y={self.y}"  
  
c = Punto(1, 2)  
print(c) # Punto: x=1 y=2
```

## 9.42 Excepciones: Manejar excepciones

✗ Cuidado cuando uses `input`. Esto permite al usuario introducir una entrada. Nunca puedes fiarte de lo que viene del exterior. Si en vez de un número introduces `texto` obtendrás una excepción que rompe el programa.

```
edad = int(input("Indica tu edad: "))  
if edad >= 18:  
    print("Mayor de edad")  
else:  
    print("Menor de edad")
```

```
# ValueError: invalid literal for int() with base 10:  
'texto'
```

- ✓ Siempre que conviertas entre tipos usa un bloque `try` y maneja posibles excepciones. De esta manera el programa no terminará de forma abrupta.

```
try:  
    edad = int(input("Indica tu edad: "))  
    if edad >= 18:  
        print("Mayor de edad")  
    else:  
        print("Menor de edad")  
except ValueError as e:  
    print("La entrada no es una edad correcta")
```

## 9.43 Excepciones: Manejar excepciones genéricas

- ✗ Es posible usar `Exception` para manejar excepciones, pero es demasiado genérico.

```
a, b = 10, 0  
try:  
    print(a/b)  
except Exception as e:  
    print("Error:", e)
```

- ✓ Suele ser recomendable manejar cada excepción por separado, ya que es posible que quieras manejarlas de forma distinta.

```
a, b = 10, "0"  
try:  
    print(a/b)  
except ZeroDivisionError as e:  
    print("Error ZeroDivisionError:", e)  
except TypeError as e:  
    print("Error TypeError:", e)
```

---

## 9.44 Excepciones: Ignorar excepciones

✗ Es muy peligroso ignorar excepciones con `continue`. Esta excepción queda silenciada y nadie sabrá si pasó o no. Pasa lo mismo si usas `pass`.

```
try:  
    print(10/0)  
except Exception:  
    continue
```

✓ Como poco muestra un *log* de la excepción para que no quede en el olvido.

```
try:  
    print(10/0)  
except Exception as e:  
    print(f"Error: {e}")  
# Error: division by zero
```

## 9.45 Excepciones: Usar context managers

✗ Si quieres abrir un fichero y mostrar su contenido lo puedes hacer así. Aunque no es recomendable porque este código es propenso a que te olvides o no se ejecute el `close`. Y es muy importante cerrar el fichero después de terminar.

```
archivo = open('archivo.txt', 'r')  
contenido = archivo.read()  
print(contenido)  
archivo.close()
```

✓ Usa *context managers*. Gracias a ellos el fichero se cierra automáticamente sin tener que indicarlo expresamente.

```
with open('archivo.txt', 'r') as archivo:
```

```
contenido = archivo.read()
print(contenido)
```

## 9.46 Paquetes: Nombrar fichero como paquete

- ✗ No llames a tus ficheros igual que otros paquetes que uses. Por ejemplo no uses `numpy.py` ni `math.py`. Esto producirá conflictos.
- ✓ Usa nombres para tus paquetes y módulos distintos.

## 9.47 Paquetes: Incluye tu licencia

- ✗ Si publicas tu código o paquete en Internet sin especificar ninguna licencia, estará por defecto protegido por derechos de autor. Esto limita el uso que le pueden dar otras personas.
- ✓ Es mejor incluir una licencia que permita a otros usar tu código de manera libre. Igual que tú te beneficias del código de otras personas, permite que otros se beneficien del tuyo. Algunos ejemplos de licencias libres son *MIT*, *Apache* o *GPL*.

## 9.48 Paquetes: Usar fichero requirements.txt

- ✗ Si distribuyes tu código para que otra gente lo use, es importante que incluyas los paquetes externos que necesita. Puedes hacer esto con `pip freeze > requirements.txt`. Pero no debe faltar la versión.

```
numpy
pandas
requests
```

- ✓ Incluye siempre la versión de la dependencia exacta. De esta manera garantizas que el código funcionará.

---

```
numpy==2.1.2
pandas==2.2.3
requests==2.32.3
```

## 9.49 Comentarios: Redundantes

✗ Evita usar comentarios redundantes que no aporten nada.

```
# Inicializamos la variable x con el valor 5.
x = 5
```

✓ Si no aporta nada, no pongas comentario. Y si pones un comentario, que aporte valor.

```
x = 5
```

## 9.50 Comentarios: Confundir con docstrings

✗ Si quieres añadir un comentario que se extienda en varias líneas, no uses las triples comillas salvo que sea un *docstring*.

```
"""
Esto es un comentario de
varias líneas. Pero consume memoria.
"""
```

✓ Si es un comentario normal, usa `##`.

```
# Haz esto mejor si quieras tener
# un comentario de varias líneas.
```

## 9.51 Comentarios: Documentar funciones

✗ Es mejor no documentar así las funciones.

```
def calcula_area(ancho, largo):
    # Calcula el área de un rectángulo
    return ancho * largo
```

✓ Usa *docstrings* con triple `"` e indicando las entradas y salidas.

```
def calcula_area(ancho, largo):
    """
    Calcula el área de un rectángulo

    Parameters:
    ancho (int or float): El ancho del rectángulo.
    largo (int or float): El largo del rectángulo.

    Returns:
    int or float: El área del rectángulo
    """
    return ancho * largo
```

## 9.52 General: Evita cálculos innecesarios

✗ Imagina una función que convierte de grados a radianes. Aunque funciona perfectamente estamos calculando `pi/180` cada vez. Y esto en realidad es una constante.

```
import math

def grados_a_radianes(grados):
    radianes = grados * (math.pi / 180)
    return radianes

print(grados_a_radianes(45))
# 0.7853981633974483
```

✓ Es mejor calcular el valor `pi/180` y usarlo directamente. Nos ahorraremos algunos cálculos.

```
FACTOR_GRADOS_RAD = 0.017453292519943295
def grados_a_radianes(grados):
    radianes = grados * FACTOR_GRADOS_RAD
    return radianes

print(grados_a_radianes(45))
# 0.7853981633974483
```

## 9.53 General: Medir tiempo una vez

✗ Si quieres medir el tiempo que tarda en ejecutarse un fragmento de código no lo hagas una sola vez. Es mejor medirlo muchas veces y tomar una media.

```
import time

start_time = time.time()
sum(x * x for x in range(10**6))
print(f"tiempo={time.time() - start_time}")
```

✓ Para ello puedes usar `timeit` .

```
from timeit import timeit

tiempo = timeit('sum(x * x for x in range(10**6))',
                 number=50)/50
print(f"tiempo={tiempo}")
```

## 9.54 General: Escapar comillas

✗ Si quieres incluir comillas dentro de una cadena no lo puedes hacer de la siguiente forma ya que obtendrás un `SyntaxError` .

```
s = "La palabra "feedback" es inglesa"
print(s)
# SyntaxError: invalid syntax
```

- 
- ✓ Puedes hacerlo de varias formas. Usando comillas simples para la cadena o usand `\\"`.

```
s1 = 'La palabra "feedback" es inglesa'  
s2 = "La palabra \"feedback\" es inglesa"
```

## 9.55 General: Uso incorrecto de sorted

- ✗ Ten cuidado cuando ordenes una lista con `sorted`. Al usar el código ASCII, la `z` se considera que va antes de la `a`.

```
l = ['a', 'b', 'c', 'Z']  
print(sorted(l))  
# ['Z', 'a', 'b', 'c']
```

- ✓ Para usar `sorted` define tu lógica de ordenado.

```
l = ['a', 'b', 'c', 'Z']  
print(sorted(l, key=lambda x: x.lower()))  
# ['a', 'b', 'c', 'Z']
```

## 9.56 General: Confundir identidad con igualdad

- ✗ No es lo mismo identidad que igualdad. El operador `is` nos dice si ambas variables son la misma. Se entiende por ser la misma si ambas hacen referencia a la misma posición de memoria.

```
a = [1, 2, 3]  
b = [1, 2, 3]  
  
print(a is b) # False
```

- ✓ Si queremos ver si son iguales, deberemos usar el operador `==` para comparar.

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b) # True
```

## 9.57 General: Nombrar variables mal

✗ El PEP8 establece una convención de cómo nombrar variables en Python. Los siguientes ejemplos son incorrectos.

```
def unaFuncionAsiEstaMal(a, b):
    return a + b

class una_clase:
    pass
```

✓ La forma correcta de nombrar a una función es con `snake_case` y clase con `CamelCase`.

```
def una_funcion_así_esta_bien(a, b):
    return a + b

class UnaClase:
    pass
```

## 9.58 General: Uso de asserts en producción

✗ Ten cuidado con el uso de `assert`. Imagina que solo queremos que los usuarios autorizados ejecuten una tarea. Esto puede parecer que funciona bien, pero si ejecutamos el código con `python -O codigo.py` el `-O` elimina los `assert` y el `usuario3` no autorizado puede entrar.

```
AUTORIZADOS = ("usuario1", "usuario2")
```

---

```
def tarea_restringida(usuario):
    assert usuario in AUTORIZADOS
    print("Ejecutar tarea restringida")

tarea_restringida("usuario3")
```

- Mejor realiza las comprobaciones de la siguiente manera.

```
def tarea_restringida(usuario):
    if usuario in AUTORIZADOS
        print("Ejecutar tarea restringida")
    else:
        raise Exception(f'{usuario} no autorizado')
```

## 9.59 General: Código específico por OS

✗ No escribas código que solo funcione en una plataforma. Si quieres crear una ruta a un fichero, si lo haces así no funcionará en Windows. Algunos sistemas usan `/`, pero otros como Windows usa `\`.

```
ruta = "carpeta" + "/" + "Documentos" + "/" + "fichero.txt"
```

- Usa la función `join` que se adaptará dependiendo del sistema operativo.

```
import os
ruta = os.path.join("carpeta", "Documentos", "fichero.txt")
```

## 9.60 General: Uso de magic numbers

✗ Evita el uso de *magic numbers*. Esto son números que aparecen en el código sin saber muy bien de donde vienen. Para un lector puede ser confuso ver ese `0.3048` sin saber de donde viene.

```
altura_pies = 3006
altura_metros = altura_pies * 0.3048
print(altura_metros) # 916.2
```

- ✓ Asigna mejor ese número a una variable constante. Usa también un comentario.

```
# Factor de conversión de pies a metros
FACTOR_PIES_METROS = 0.3048
altura_metros = altura_pies * FACTOR_PIES_METROS
```

## 9.61 General: Rellena tu `.gitignore`

- ✗ Si trabajas con *Git* no tengas el `.gitignore` vacío.
- ✓ Dile que ignore ficheros que no son necesarios como `__pycache__`. Esto hará que tu repositorio público de código no tenga ficheros innecesarios. Y a veces privados como el `.env`.

```
# Contenido .gitignore
__pycache__/
*.py[cod]
*$py.class
env/
.env/
*.pyc
```

## 9.62 General: Precisión en `float`

- ✗ Los `float` no tienen precisión infinita. Aunque las matemáticas nos digan que `0.1` sumado `10` veces es `1`, esto no es así en Python.

```
suma = sum(0.1 for i in range(10))
print(suma == 1.0) # False
```

- 
- ✓ En algunos casos cuando compares `float` puede ser mejor usar `isclose`.

```
import math
suma = sum(0.1 for i in range(10))
print(math.isclose(suma, 1.0)) # True
```

## 9.63 General: Usa alias con import

- ✗ Si quieres importar `pandas`, `numpy` o `matplotlib` puedes hacerlo así.

```
import pandas
import numpy
import matplotlib.pyplot
```

- ✓ Pero es común asignar un alias más breve. Verás que todo el mundo hace.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

## 9.64 General: Usar f-strings

- ✗ Si quieres unir una variable a un texto, puedes hacerlo así.

```
nombre = "Mundo"
saludo = "Hola " + nombre + ", bienvenido."
```

- ✓ Pero es mas *Pythonic* hacerlo así, usando los *f-strings*.

```
nombre = "Mundo"
saludo = f"Hola {nombre}, bienvenido."
```

---

## 9.65 General: Uso de print en producción

✗ Aunque el uso de `print` es muy usado, no es recomendable usarlo en entornos de producción, ya que no ofrece ninguna granularidad.

```
print("Mensaje de log")
```

✓ Es preferible utilizar `logging` con `info` o `debug` según la importancia del contenido que se está mostrando. Si se configura en modo `info`, solo se mostrarán los mensajes `logging.info`.

```
import logging

logging.basicConfig(level=logging.INFO, format='%(levelname)s - %(message)s')
logging.info("Mensaje nivel información")
logging.debug("Mensaje nivel debug")
```

## 9.66 General: Intercambiar variables

✗ Si quieres intercambiar el valor de dos variables, poner `a` en `b` y viceversa, no hace falta que uses una variable temporal.

```
a, b = 0, 10

temp = a
a = b
b = temp

print(a, b) # 10 0
```

✓ Lo puedes hacer en una línea de código.

```
a, b = b, a
```

---

## 9.67 General: Extraer tuplas a variables

✗ Si tienes una tupla con varios valores, no es necesario que los asignes a una variable uno a uno.

```
datos = (50, "Python")
edad = datos[0]
lenguaje = datos[1]
```

✓ Lo puedes hacer de la siguiente forma.

```
datos = (50, "Python")
edad, lenguaje = datos
```

## 9.68 General: Uso de all

✗ Si tienes una lista de `bool` y quieres ver si todos son `True` no lo hagas así.

```
v = [True, True, False]
if v[0] and v[1] and v[2]:
    print("Todos son verdaderos")
```

✓ Usar la función `all`. Si quieres ver si al menos uno es `True` puedes usar la función `any`.

```
v = [True, True, False]
if all(v):
    print("Todos son verdaderos")
```

## 9.69 General: No repeter código 1

✗ Si quieres ejecutar una función se cumpla o no una condición, puedes evitar el código duplicado.

```
if x:  
    hacer_a()  
    hacer_b()  
else:  
    hacer_b()
```

✓ Como `hacer_b` se ejecuta en ambos casos, lo puedes simplificar.

```
if x:  
    hacer_a()  
hacer_b()
```

## 9.70 General: No repetir código 2

✗ Imagina que tienes un texto al que quieras quitar los signos de exclamación e interrogación. Puedes usar `replace` repetidas veces.

```
def procesar_texto(texto):  
    texto = texto.replace('?', '')  
    texto = texto.replace('?', '')  
    texto = texto.replace('!', '')  
    texto = texto.replace('¡', '')  
    return texto
```

✓ Pero podría quedar mejor hacerlo de la siguiente manera.

```
def procesar_texto(texto):  
    for char in ['?', '?', '!', '¡']:  
        texto = texto.replace(char, '')  
    return texto
```

## 9.71 General: Uso de tuplas con constantes

✗ Si quieres almacenar un conjunto de valores constantes en tu código, es mejor que no uses listas. Una lista puede ser modificada de forma accidental.

---

```
http_status = [200, 404, 403, 500]
```

- Usa tuplas, ya que son elementos inmutables. Una vez declarado no se puede cambiar.

```
http_status = (200, 404, 403, 500)
```

## 9.72 General: env son strings

- Ten cuidado si usas variables de entorno. Su contenido es de tipo `str` por lo que lo siguiente nunca pasará. No es lo mismo `6` que `"6"`.

```
import os

if os.getenv("MYVAR") == 6:
    print("Es 6")
```

- Si quieres comparar con `6` que es un `int`, no olvides convertirlo primero.

```
import os

if int(os.getenv("MYVAR")) == 6:
    print("its 5")
```

## 9.73 General: Confunde cuadrado

- No confundas el operador `^` y el `**`. Si quieres elevar el cuadrado, no hagas esto. El operador `^` no es lo que buscas, se trata del operador XOR. Dado que `5` es `101` en binario y `2` es `010` el resultado es `111`, o lo que es lo mismo en decimal, `7`.

```
print(5^2) # 7
```

- ✓ Lo que buscas para elevar al cuadrado es `**`.

```
print(5**2) # 25
```

## 9.74 General: Sigue el formato de PEP8

- ✗ Formatea tu código de acuerdo a la recomendación PEP8. Por ejemplo, añade espacios entre números y operadores.

```
resultado = 2+3*4
```

- ✓ Este código si sigue la PEP8.

```
resultado = 2 + 3 * 4
```

## 9.75 General: Saturar un valor

- ✗ Cuando quieras saturar un valor, es decir, usar el propio `valor` si es menor que un límite o saturar al límite si el mayor, no hagas lo siguiente.

```
def saturar(valor, valor_max):
    saturado = valor_max
    if valor < valor_max:
        saturado = valor
    return saturado

print(saturar(30, 15)) # 15
print(saturar(13, 15)) # 13
```

- ✓ Se puede simplificar de la siguiente manera.

```
def saturar(valor, valor_max):
```

```
    return min(valor, valor_max)

print(saturar(30, 15)) # 15
print(saturar(13, 15)) # 13
```

## 9.76 Módulos: Importe circular

✗ Si tienes un módulo `mod_a` que importa `mod_b` y a su vez ese módulo `mod_b` importa `mod_a` tendrás un problema de *circular import* `ImportError`.

```
# mod_a.py
from mod_b import funcion_b
def funcion_a():
    pass
```

Por otro lado `mod_b`.

```
# mod_b.py
from mod_a import funcion_a
def funcion_b():
    pass
```

✓ Para evitar tener un *circular import* es recomendable que definas un tercer módulo que contenga esas funciones comunes a importar por ambos módulos.

## 9.77 Módulos: Importe explícito

✗ No uses `*` para importar todo el contenido del módulo.

```
from tu_modulo import *
```

✓ Es mejor ser explícito e indicar exactamente lo que quieras importar.

---

```
from tu_modulo import funcion_a, funcion_b
```

## 9.78 Numpy: Mezcla de tipos

✗ Si defines un `array` de `numpy` con diferentes tipos, Python cambiará los tipos para acomodar a todos. En este caso convierte `1` y `2` en `str`.

```
import numpy as np
arr = np.array([1, 2, '3'])
print(arr) # ['1' '2' '3']
```

✓ Siquieres forzar a que se utilice un tipo concreto, usa `dtype`. Ten cuidado cuando mezclas tipos.

```
arr = np.array([1, 2, '3'], dtype=int)
print(arr) # [1 2 3]
```

## 9.79 Numpy: Usar operaciones vectorizadas

✗ Si tienes un `array` yquieres sumar `10` a cada elemento, no hace falta iterarlo todo.

```
import numpy as np
arr = np.array([1, 2, 3])
for i in range(len(arr)):
    arr[i] = arr[i] + 10

print(arr) # [11 12 13]
```

✓ Te puedes beneficiar de las operaciones vectorizadas de `numpy`. Más fácil.

```
print(arr + 10) # [11 12 13]
```

## 9.80 Numpy: Uso de mask

✗ Si tienes un array y quieres quedarte con los elementos `>0`, no lo hagas así.

```
import numpy as np
arr = np.array([-3, 3, -5, 10, 20, 1])

filtrado = []
for i in range(len(arr)):
    if arr[i] > 0:
        filtrado.append(arr[i])

print(np.array(filtrado)) # [ 3 10 20 1]
```

✓ Aprovecha el *boolean indexing* o *masking* de `numpy` y hazlo así. El resultado es el mismo y se beneficia de la vectorización, que lo hace más rápido.

```
import numpy as np
print(arr[arr > 0]) # [ 3 10 20 1]
```

## 9.81 Numpy: Confundir multiplicación

✗ Si quieres multiplicar dos matrices como se haría en álgebra, no uses el operador `*`, ya que este las multiplica elemento a elemento.

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
print(arr1 * arr2)
# [[ 5 12]
#  [21 32]]
```

✓ Para multiplicar dos matrices usa `@`.

```
print(arr1 @ arr2)
# [[19 22]
#  [43 50]]
```

## 9.82 Numpy: Uso de append

✗ Si conoces de antemano el tamaño de tu array `numpy` no hagas esto. Es muy ineficiente, ya que cada `append` tiene que asignar memoria nueva y en algunos casos copiar otra vez todo lo anterior.

```
import numpy as np

arr = np.array([])
for i in range(1000):
    arr = np.append(arr, i)
```

✓ Es mejor definir *a priori* el tamaño. La diferencia en eficiencia es muy grande.

```
size = 1000
arr = np.empty(size)
for i in range(size):
    arr[i] = i
```

## 9.83 Numpy: Tamaño y overflow

✗ En el siguiente ejemplo tenemos un *overflow*. Python asigna un máximo de 6 letras a cada registro por lo que si usas un nombre mayor, se verá cortado. El 6 viene de `Alicia`. Esto es peligroso. Perdemos información.

```
import numpy as np

a = np.rec.fromrecords(
```

```
[('Alicia', 25), ('Bob', 30)],  
names=['nombre', 'edad'])  
  
a["nombre"][0] = "NuevoNombreLargo"  
print(a["nombre"]) # ['NuevoN' 'Bob']
```

- ✓ Puede ser mejor ser explícito con el `dtype` que queremos. En este caso asignamos `20` letras.

```
a = np.rec.fromrecords(  
[('Alicia', 25), ('Bob', 30)],  
dtype=[('nombre', 'U20'), ('edad', 'i4')])  
  
a["nombre"][0] = "NuevoNombreLargo"  
print(a["nombre"]) # ['NuevoNombreLargo' 'Bob']
```

## 9.84 Numpy: Optimizar columnas

- ✗ En `numpy` cuando almacenamos un número como la edad se usa `int64` por defecto. En casos donde almacenamos números pequeños, podemos optimizar esto.

```
arr1 = np.rec.fromrecords(  
[('Alicia', 25), ('Bob', 30)],  
names=['nombre', 'edad'])  
  
print(f"{arr1.nbytes} bytes") # 64 bytes
```

- ✓ Podemos usar `uint8` lo que nos permite reducir la memoria que ocupa. Puede que 22 bytes no parezca mucho, pero si almacenamos millones de valores, ahorraremos muchos GB.

```
arr1 = np.rec.fromrecords(  
[('Alicia', 25), ('Bob', 30)],  
dtype=[('name', 'U5'), ('age', np.uint8)])  
  
print(f"{arr1.nbytes} bytes") # 42 bytes
```

---

## 9.85 Pandas: Usa chunksize

✗ Si trabajas con ficheros enormes de varios GB, es posible que no puedas cargar todo el fichero en un `DataFrame` para procesarlo.

```
df = pd.read_csv('fichero_enorme.csv')  
# Procesa
```

✓ Si tu fichero es demasiado grande, considera procesarlo en trozos o *chunks*. De esta manera no cargarás todo en memoria, sino los trozos.

```
for chunk in pd.read_csv('fichero_enorme.csv',  
                         chunksize=1000):  
    # Procesa de 1000 en 1000
```

## 9.86 Pandas: Renombrar columnas

✗ Si tienes quieres cambiar el nombre de varias columnas, como eliminar espacios y usar minúsculas, no es necesario cambiar el nombre uno a uno.

```
import pandas as pd  
  
data = {'Nombre': [1, 2],  
        'EDAD': [4, 5],  
        'ciudaD': [6, 7]}  
df = pd.DataFrame(data)  
df = df.rename(columns={'Nombre': 'nombre',  
                       'EDAD': 'edad',  
                       'ciudaD': 'ciudad'})
```

✓ Puedes usar `rename` para quitar los espacios (`strip()`) y usar minúsculas (`lower()`).

```
df = df.rename(columns=lambda x: x.strip().lower())
```

```
print(df)
```

## 9.87 Pandas: Usar category

✗ Si quieres almacenar un `DataFrame` cuyos elementos pueden tomar unos valores determinados conocidos, hacerlo así no es muy eficiente.

```
import pandas as pd

df = pd.DataFrame({
    'productos': ['A', 'B', 'C'] * 100000
})

print("Memoria", df.memory_usage(deep=True).sum())
# Memoria 17400128
```

✓ Considera usar el tipo `category` lo que reduce el consumo de memoria notablemente.

```
df['productos'] = df['productos'].astype('category')
print("Memoria", df.memory_usage(deep=True).sum())
# Memoria 300410
```

## 9.88 Reinventar la rueda: 1

✗ Si quieres eliminar los duplicados de una lista, no es necesario escribirlo de cero.

```
l = [1, 2, 2, 3, 4, 4, 5]
sin_duplicados = []
for elemento in l:
    if elemento not in sin_duplicados:
        sin_duplicados.append(elemento)

# [1, 2, 3, 4, 5]
```

- 
- ✓ Puedes usar un set para eliminar los duplicados y convertir a lista otra vez.

```
l = [1, 2, 2, 3, 4, 4, 5]
sin_duplicados = list(set(l))
# [1, 2, 3, 4, 5]
```

## 9.89 Reinventar la rueda: 2

- ✗ Si quieres contar los elementos de una lista, no es necesario que lo escribas desde cero.

```
elementos = ['a', 'b', 'c', 'a', 'b', 'b']
conteo = {}
for e in elementos:
    if e in conteo:
        conteo[e] += 1
    else:
        conteo[e] = 1

print(conteo)
# {'a': 2, 'b': 3, 'c': 1}
```

- ✓ Puedes usar `Count`.

```
from collections import Counter
elementos = ['a', 'b', 'c', 'a', 'b', 'b']
conteo = Counter(elementos)

print(conteo)
# Counter({'b': 3, 'a': 2, 'c': 1})
```

## 9.90 Reinventar la rueda: 3

- ✗ Si quieres saber si una cadena empieza por algo en concreto.

```
x = "El Libro de Python"  
print(x[0:2] == "El") # True
```

✓ Es mejor hacerlo con `startswith` .

```
x = "El Libro de Python"  
print(x.startswith("El")) # True
```

## 9.91 Reinventar la rueda: 4

✗ Si quieres ver si un elemento está contenido en una lista lo puedes hacer así.

```
def contiene(nombres, nombre):  
    encontrado = False  
    for n in nombres:  
        if n == nombre:  
            encontrado = True  
            break  
    return encontrado  
  
nombres = ['Ana', 'Juan', 'Pedro']  
print(contiene(nombres, 'Juan')) # True
```

✓ Pero es más fácil usar `in` .

```
print('Juan' in nombres) # True
```

## 9.92 Reinventar la rueda: 5

✗ Si quieres unir los elementos de una lista por `,` lo puedes hacer así.

```
nombres = ['Ana', 'Juan', 'Pedro']  
resultado = ''  
for index, nombre in enumerate(nombres):
```

```
if index < len(nombres) - 1:  
    resultado += nombre + ', '  
else:  
    resultado += nombre  
  
print(resultado) # Ana, Juan, Pedro
```

✓ Pero es más fácil usar `join`.

```
resultado = ', '.join(nombres)  
print(resultado) # Ana, Juan, Pedro
```

## 9.93 Reinventar la rueda: 6

✗ Si quieres leer un csv puedes crear tu propia función.

```
def lee_csv(ruta):  
    with open(ruta, 'r') as fichero:  
        lineas = fichero.readlines()  
        datos = [linea.strip().split(',') for linea in  
                 lineas]  
    return datos
```

✓ Pero es mejor usar alguna librería externa como `numpy`.

```
import numpy as np  
datos = np.genfromtxt('datos.csv', delimiter=',',  
                      skip_header=1)
```

## 9.94 Reinventar la rueda: 7

✗ Si quieres iterar una lista muy grande en trozos, en vez de definir tu el código.

```
def chunks(lst, n):  
    for i in range(0, len(lst), n):  
        yield lst[i:i + n]
```

---

```
for chunk in chunks(lista_enorme, 100):
    print(chunk)
```

Puedes hacer uso de `batched`.

```
import itertools as it
for chunk in it.batched(lista_enorme, 100):
    print(chunk)
```

## 9.95 Vulnerabilidad: Usar pickle

No uses la librería `pickle` para serializar y deserializar. Permite la ejecución de código no autorizado. Si conseguimos que un usuario abra nuestro `datos_maliciosos` podremos ejecutar código en su máquina. En este ejemplo solo listamos con `ls` sus ficheros, pero se puede hacer mucho más.

```
import pickle
import os
class Malicioso:
    def __reduce__(self):
        return (os.system, ('ls',))

datos_maliciosos = pickle.dumps(Malicioso())
pickle.loads(datos_maliciosos)
```

Usa otras herramientas para serializar, como `json`.

## 9.96 Vulnerabilidad: Inyección SQL

Si tienes una base de datos con una función que permite obtener los `usuarios` no lo hagas así. Este código es vulnerable a inyección SQL. Si la entrada es `"admin' OR '1'='1"` un atacante podrá obtener todos tus datos.

```
import sqlite3

def get_usuario(usuario):
    conn = sqlite3.connect('ejemplo.db')
    cursor = conn.cursor()

    query = f"SELECT * FROM usuarios WHERE usuario = \
        '{usuario}'"
    cursor.execute(query)

    result = cursor.fetchall()
    conn.close()

    return result
```

✓ Mejor hazlo así. Este código ya no es vulnerable.

```
def get_usuario(usuario):
    conn = sqlite3.connect('ejemplo.db')
    cursor = conn.cursor()

    query = "SELECT * FROM usuarios WHERE usuario = ?"
    cursor.execute(query, (usuario,))

    result = cursor.fetchall()
    conn.close()

    return result
```

## 9.97 Vulnerabilidad: No usar eval y exec

✗ Ten mucho cuidado con `eval` y `exec`. Permite ejecutar código arbitrario, lo que es la puerta de entrada perfecta para cualquier atacante. Si a este programa le pasas como entrada `__import__('os').system("ls")` puedes saber el contenido de la carpeta. Podrías incluso acceder a ficheros, borrarlos, etc.

```
entrada = input("comando: ")
eval(entrada)
```

- ✓ A menos que sepas lo que haces, no uses `eval` o `exec`. Y si los usas, al menos limita lo que se puede ejecutar.

## 9.98 Vulnerabilidad: Usar pip sin saber que hay

✗ Ten mucho cuidado con tus dependencias. Que un paquete esté publicado en `pip` no significa nada. Puede tener errores. Puede tener código malicioso.

- ✓ Asegúrate de que tus dependencias son de fiar. Busca si son paquetes conocidos, si hay otra gente que los usa, quien hay detrás.

## 9.99 Vulnerabilidad: Request sin timeout

✗ Puedes realizar una petición a un servidor externo de la siguiente forma. Ahora imagina que ese servidor no te responde, y te deja esperando para siempre.

```
import requests
response = requests.get('https://ejemplo.com')
print(response.status_code)
```

- ✓ Para prevenir este tipo de ataque es recomendable usar un `timeout`. Esto es el tiempo que esperaremos. Pasado ese tiempo, si no hemos recibido respuesta, daremos error y a otra cosa.

```
import requests

response = requests.get('https://example.com', timeout
=5)
print(response.status_code)
```

---

## 9.100 Vulnerabilidad: Cuidado con random

✗ Si quieres generar una clave privada, no uses `random`. Este paquete está bien cuando queremos un número aleatorio para pruebas, pero no es seguro para generar claves privadas. Digamos que no es todo lo aleatorio que debe.

```
import random
clave_privada = random.getrandbits(256)
```

✓ Usa mejor el módulo `secrets` que proporciona mayor aleatoriedad o entropía.

```
import secrets
clave_privada = secrets.randbits(256)
```

---

Enhorabuena, has llegado al final del libro. Estoy seguro de que ya no ves Python de la misma manera. Tampoco tengo dudas de que aunque este libro haya resuelto tus dudas iniciales, te ha generado muchas más. No pasa nada, es normal.

Ahora es el momento de que pongas en práctica todo lo que has aprendido. Busca una idea, un problema real que te inspire, y úsalo como excusa para practicar Python. A programar se aprende programando.

No dudes en usar todos los recursos a tu alcance. Este libro, Google, StackOverflow, YouTube, o LLMs como ChatGPT. Pero no lo olvides. Nunca uses código que no entiendas.

Recuerda también que aunque lo importante es que el código funcione, hay una serie de buenas prácticas que debes seguir para que tu código sea fácil de entender, mantener y escalar. No olvides también escribir tests.

Si has construido algo con Python que quieras compartir, estamos encantados de que nos lo hagas saber. También si tienes sugerencias o correcciones. Nos puedes contactar a [ellibrodepython@gmail.com](mailto:ellibrodepython@gmail.com)