

La Biblia de Python
De Principiante a Desarrollador Senior
Creado por "Roberto Arce"

La Biblia de Python

© 2025 | QA sin filtros

Todos los derechos reservados.

Queda prohibida la reproducción total o parcial de esta obra por cualquier medio sin autorización expresa del autor.

Este libro está basado en experiencias reales y contiene opiniones sobre el ejercicio profesional de la calidad en proyectos de software.

Nombres de productos, empresas o situaciones reales se mencionan únicamente con fines educativos.

Primera edición: 2025

Diseño y estrategia editorial: QA sin filtros

Publicado por el autor a través de Amazon Kindle Direct Publishing (KDP)

www.amazon.com/kdp

La Biblia de Python

ÍNDICE GENERAL

Una guía completa para dominar Python desde los fundamentos hasta el desarrollo profesional, cubriendo buenas prácticas, automatización, testing, APIs, bases de datos, ciencia de datos y DevOps.

Prólogo

CAPÍTULO 1: Fundamentos sólidos de Python

- Historia de Python y filosofía *Zen*
- Instalación y configuración del entorno
- Instalación en Windows
- Verificar instalación
- Instalación en macOS
- Instalación en Linux (Debian, Ubuntu)
- `pip`: el gestor de paquetes
- Editores e IDEs recomendados
 - Visual Studio Code (VSCode)
 - PyCharm
 - JupyterLab y Notebooks
 - Recomendación final
- Tipos de datos y operadores
 - Tipos primitivos: números, booleanos, cadenas
 - Colecciones: listas, tuplas, sets, diccionarios
 - Operadores aritméticos, lógicos y de comparación
 - Operadores de identidad y pertenencia
 - Conversión de tipos (*type casting*)
 - Tipado dinámico vs. estático
- Control de flujo
 - Condicionales: `if`, `elif`, `else`
 - Bucles: `for`, `while`
 - Comprensiones de listas (*List Comprehensions*)
 - Cuándo usar cada estructura
- Manejo de errores y excepciones
 - `try`, `except`, `else`, `finally`
 - Tipos comunes de excepciones
 - Excepciones personalizadas
 - Buenas prácticas

La Biblia de Python

- Funciones y módulos
 - Definición, argumentos y retorno
 - `lambda` y funciones anónimas
 - `map`, `filter`, `reduce`
 - Docstrings* y anotaciones de tipo
 - Módulos y paquetes

- **Ejercicios recomendados**

1. Tipos de datos y operaciones básicas
2. Estructuras de datos
3. Control de flujo
4. Funciones y reutilización
5. Proyecto integrador — Parte 1

CAPÍTULO 2: Programación Orientada a Objetos

- Introducción a la POO
 - Qué es y por qué usarla
 - Analogías y ejemplos visuales
 - Características clave
- Clases, objetos, atributos y métodos
 - Ejemplo básico
 - Atributos de clase e instancia
 - Métodos y comportamiento
 - Ejemplo práctico completo
- Herencia, polimorfismo y encapsulamiento
 - Herencia simple y múltiple
 - Polimorfismo
 - Encapsulamiento y propiedades con `@property`
- Decoradores de clase
 - `@classmethod` y `@staticmethod`
 - Cuándo y cómo usarlos
- Patrones de diseño en Python
 - Patrón Singleton
 - Patrón Factory
 - Patrón Strategy
 - Cómo testear patrones de diseño

- **Ejercicios recomendados**

1. Clases, atributos y métodos
2. Herencia y polimorfismo
3. Encapsulamiento y propiedades

La Biblia de Python

4. Métodos estáticos y de clase
5. Patrones de diseño
6. Mini-proyecto integrador — POO

CAPÍTULO 3: Calidad, Testing y Buenas Prácticas

- Introducción: por qué importa la calidad
- Testing profesional con `unittest` y `pytest`
 - Pruebas unitarias y de integración
 - Fixtures*, *mocking* y estructura de carpetas
 - Cobertura de código y recomendaciones
- Desarrollo guiado por tests (TDD)
 - Ciclo de TDD
 - Caso práctico: calculadora
 - Cuándo y cómo aplicarlo
- Buenas prácticas y estilo
 - PEP8, *flake8*, *black*, *isort*
 - Integración de herramientas
 - Principios profesionales
- Principios SOLID y Clean Code
 - Los 5 principios clave (SRP, OCP, LSP, ISP, DIP)
 - Ejemplos aplicados y refactor completo
- **Ejercicios recomendados**
 1. Testing con `unittest` y `pytest`
 2. TDD (*Test Driven Development*)
 3. Estilo, *flake8*, *black*, *isort*
 4. Clean Code y SOLID

CAPÍTULO 4: Automatización de archivos, carpetas y procesos

- Trabajando con archivos y carpetas (`os`, `pathlib`)
 - Leer, escribir, mover y eliminar archivos
 - Automatización de tareas por lotes
- Ejecución de comandos del sistema (`subprocess`)
 - Captura de errores y ejemplos prácticos
- Automatización con emails, PDFs y Excel
 - Enviar correos con Gmail
 - Leer y combinar PDFs
 - Manipular Excel con `openpyxl` y `pandas`
 - Caso práctico: informe semanal automatizado

La Biblia de Python

- Web Scraping profesional
 - Requests + BeautifulSoup*
 - Selenium* y técnicas anti-bloqueo
 - User-Agent, headless*, buenas prácticas
- **Ejercicios recomendados**
 1. Automatización de archivos y carpetas
 2. Emails, PDF y Excel
 3. Web Scraping
 4. Selenium avanzado

CAPÍTULO 5: Manejo de Datos y Bases de Datos

- Bases de datos relacionales
 - SQLite, PostgreSQL, MySQL
 - CRUD con SQL y SQLAlchemy ORM
 - Relaciones, migraciones y buenas prácticas
- Bases NoSQL
 - MongoDB con PyMongo
 - Inserciones, consultas y actualizaciones
- Persistencia y serialización
 - JSON, Pickle, YAML
 - Cuándo usar cada uno
- **Ejercicios recomendados**
 1. SQLite / PostgreSQL / MySQL
 2. SQLAlchemy ORM
 3. MongoDB con PyMongo
 4. Persistencia y serialización

CAPÍTULO 6: APIs, Servicios Web y Backends

- APIs REST con Flask
 - Rutas, validaciones, middlewares
- APIs modernas con FastAPI
 - Tipado con Pydantic
 - Documentación automática (Swagger UI)
- Seguridad en APIs
 - Autenticación y JWT
 - Protección de rutas y roles

La Biblia de Python

- Consumo de APIs externas
Autenticación, errores y manejo de respuestas
- **Ejercicios recomendados**
 1. Flask — rutas y validaciones
 2. FastAPI — tipado y documentación
 3. Autenticación y autorización
 4. Consumo de APIs externas

CAPÍTULO 7: Programación Concurrente y Asíncrona

- Multithreading — ejecución concurrente con hilos
- Multiprocessing — paralelismo real
- Asyncio — asincronía moderna en Python
aiohttp, async/await, gather, create_task
- Casos prácticos
 - Web scraping concurrente
 - Automatización masiva
 - Microservicios asíncronos
- **Ejercicios recomendados**
 1. Hilos con `threading`
 2. Procesamiento con `multiprocessing`
 3. Asincronía con `asyncio`

CAPÍTULO 8: Python para Ciencia de Datos

- NumPy: vectores, matrices y operaciones
- Pandas: series, dataframes y limpieza
- Visualización: Matplotlib y Seaborn
- Machine Learning con Scikit-learn
 - Clasificación, regresión, evaluación de modelos
 - Preprocesamiento y *cross-validation*
- **Buenas prácticas profesionales**

CAPÍTULO 9: DevOps, CI/CD y Despliegue

- Docker para proyectos Python
Dockerfile, docker-compose, optimización

La Biblia de Python

- Despliegue en la nube
Heroku, AWS Lambda, Render
- CI/CD con GitHub Actions
- Tests automáticos y pipelines
- **Buenas prácticas y casos reales**

CONCLUSIONES Y PRÓXIMOS PASOS

- Cierre del ciclo y aprendizajes
- “¿Y ahora qué?”: cómo seguir creciendo
- Recordatorio final

ANEXOS

- **Anexo 1** — Soluciones Parte 1: Fundamentos
- **Anexo 2** — Soluciones Parte 2: POO
- **Anexo 3** — Soluciones Parte 3: Testing y Calidad
- **Anexo 4** — Soluciones Parte 4: Automatización y Scripting
- **Anexo 5** — Soluciones Parte 5: Bases de Datos y Serialización
- **Anexo 6** — Soluciones Parte 6: APIs y Backends
- **Anexo 7** — Soluciones Parte 7: Concurrencia y Asincronía

La Biblia de Python

Prólogo

“Dominar un lenguaje de programación no se trata solo de escribir código que funcione. Se trata de escribir código que impacte, que escale y que transforme ideas en productos reales.”

Hace más de 15 años comencé mi carrera en el mundo del desarrollo de software. En aquel entonces, Python era solo una promesa silenciosa, eclipsada por gigantes como Java o C++. Hoy, Python no solo es el lenguaje más popular del mundo: es el motor detrás de aplicaciones web, automatización, ciencia de datos, inteligencia artificial, ciberseguridad y mucho más.

Pero con esa popularidad también ha llegado una avalancha de tutoriales superficiales, cursos inconexos y documentación que confunde más de lo que enseña. Muchos programadores aprenden a usar `print()`, `for`, o `if`, pero no saben cómo estructurar una aplicación profesional, desplegarla en producción o escribir pruebas automatizadas. En otras palabras: **no están listos para un entorno real.**

Este libro nace para cambiar eso.

La Biblia de Python no es un simple manual. Es una guía diseñada para acompañarte en tu evolución desde lo más esencial hasta el nivel profesional que exigen las empresas y proyectos modernos. Aquí no encontrarás recetas vacías ni fragmentos de código sin contexto. Encontrarás fundamentos sólidos, buenas prácticas, proyectos reales, y una mentalidad de calidad y excelencia técnica.

He trabajado en múltiples roles a lo largo de mi carrera: programador, analista, QA, manager y arquitecto de soluciones. Esa visión integral es la que he querido volcar en estas páginas. Porque entender Python no es suficiente: hay que entender **cómo se usa Python en el mundo real** para crear soluciones robustas, seguras y escalables.

Si eres principiante, este libro te dará la base que te hubiera gustado tener desde el día uno. Si ya tienes experiencia, aquí encontrarás ese salto que te separa del siguiente nivel: el enfoque senior.

Y si vienes de otro lenguaje, prepárate para enamorarte del equilibrio entre simplicidad y poder que solo Python puede ofrecer.

Así que abre tu editor, clona el repositorio, y comienza este viaje.

Porque no estás solo.

Estás a punto de leer una biblia.

Y las biblias no se hojean. Se estudian. Se viven. Se aplican.

La Biblia de Python

Capítulo 1: Fundamentos sólidos de Python

Historia de Python y filosofía Zen

Breve historia del lenguaje

Python fue creado a finales de los años 80 por **Guido van Rossum**, un programador neerlandés que trabajaba en el Centrum Wiskunde & Informatica (CWI) en Ámsterdam. Su intención era diseñar un lenguaje que fuese **intuitivo, legible y poderoso**, una alternativa a los lenguajes complicados que dominaban la época, como C o Perl.

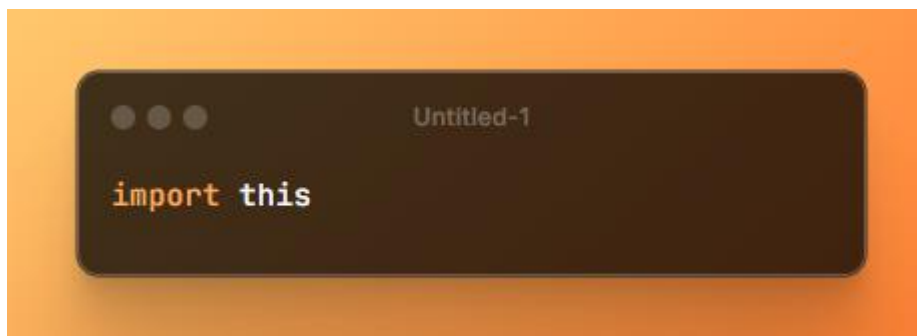
La primera versión pública apareció en 1991 y desde entonces ha evolucionado para convertirse en uno de los lenguajes más populares del mundo. Hoy en día, **Python es utilizado por gigantes tecnológicos como Google, Netflix, Facebook, NASA y Spotify**, y es uno de los pilares del desarrollo en **IA, ciencia de datos, desarrollo web y automatización**.

La elección del nombre “Python” no fue por la serpiente, sino en honor al grupo de comedia británico **Monty Python**, de donde también proviene el tono humorístico de algunos ejemplos en la documentación oficial.

La filosofía Zen de Python

Una de las claves del éxito de Python está en su filosofía. En 2004, Tim Peters definió una serie de principios conocidos como **"The Zen of Python"**, que reflejan la mentalidad que debe seguir un desarrollador que quiera escribir código pythonic (es decir, natural y elegante en Python).

Puedes visualizar estos principios ejecutando en una consola de Python:



Verás algo como lo siguiente:

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.
```

La Biblia de Python

Flat is better than nested.
Sparse is better than dense.
Readability counts.

Algunos principios clave que aplicaremos en este libro:

- **La legibilidad importa:** El código debería ser tan fácil de leer como un texto bien escrito.
- **Lo explícito es mejor que lo implícito:** Evita la magia. El código debe decir claramente lo que hace.
- **Preferir lo simple a lo complejo:** Resuelve los problemas con la solución más clara y directa posible.

Este enfoque es lo que convierte a Python en una herramienta poderosa no solo para desarrolladores avanzados, sino también para quienes empiezan.

Instalación y configuración del entorno

Trabajar con Python requiere una instalación correcta del intérprete y una estructura mínima para comenzar a desarrollar de forma profesional. A continuación explicamos cómo instalarlo y verificarlo en los tres principales sistemas operativos.

Instalación en Windows

1. Ve a <https://www.python.org/downloads>
2. Descarga el instalador correspondiente a tu versión de Windows.
3. Al iniciar la instalación, **marca la casilla que dice "Add Python to PATH"** antes de hacer clic en "Install Now".
4. Espera unos segundos hasta que finalice la instalación.

Verificar instalación

Abre la terminal de Windows (CMD) y escribe:

```
cmd
```

```
python --version
```

Si todo está bien, verás una salida como:

```
nginx
```

```
Python 3.11.4
```

La Biblia de Python

Instalación en macOS

La forma más cómoda es usar [Homebrew](#), el gestor de paquetes para macOS:

```
bash
```

```
brew install python
```

Verifica la instalación:

```
bash
```

```
python3 --version
```

En macOS es común que el comando `python` esté reservado para la versión 2.x que viene preinstalada. Por eso, se recomienda usar siempre `python3`.

Instalación en Linux (Debian, Ubuntu)

Abre una terminal y ejecuta:

```
bash
```

```
sudo apt update
```

```
sudo apt install python3 python3-pip
```

Verifica:

```
bash
```

```
python3 --version
```

[pip: el gestor de paquetes](#)

Python incluye un sistema de gestión de librerías externas llamado `pip`. Con él puedes instalar cualquier paquete publicado en el [Python Package Index \(PyPI\)](#).

Ejemplo:

```
bash
```

```
pip install requests
```

La Biblia de Python

Editores e IDEs recomendados para trabajar con Python

Uno de los factores clave para desarrollarse profesionalmente como programador Python es dominar un entorno de desarrollo eficiente. Un buen editor de código o un entorno integrado (IDE) facilita la productividad, reduce errores y permite aprovechar al máximo las herramientas del lenguaje.

Visual Studio Code (VSCode)

VSCode es uno de los editores más utilizados para Python a nivel mundial. Es gratuito, de código abierto, ligero, multiplataforma y altamente extensible.

Ventajas:

- Integración con terminal, Git y depuración.
- Recomendado para desarrollo web, automatización y proyectos pequeños o medianos.
- Compatible con entornos virtuales, Docker y Jupyter Notebooks.

Extensiones recomendadas:

- **Python (de Microsoft):** permite autocompletado, debugging y linting.
- **Pylance:** motor de análisis rápido y con tipado estático.
- **Jupyter:** para abrir y ejecutar notebooks directamente en el editor.
- **Black Formatter / isort:** para mantener el código limpio y ordenado.

Ejemplo de configuración básica en `settings.json`:

```
json
{
  "python.formatting.provider": "black",
  "editor.formatOnSave": true,
  "python.linting.enabled": true,
  "python.linting.flake8Enabled": true
}
```

PyCharm

La Biblia de Python

PyCharm, desarrollado por JetBrains, es un IDE completo y robusto específicamente diseñado para desarrollo en Python. Existen dos versiones: **Community** (gratuita) y **Professional** (de pago).

Ventajas:

- Autocompletado avanzado, navegación por clases, refactorizaciones seguras.
- Excelente para proyectos grandes, testing, desarrollo Django y análisis de código.
- Integración con Docker, bases de datos, frameworks web y herramientas DevOps.

Recomendado para: usuarios intermedios y avanzados, desarrollo backend, testing complejo, grandes bases de código.

JupyterLab y Notebooks

Jupyter Notebooks son una herramienta popular para tareas exploratorias, prototipos y ciencia de datos. Permiten mezclar código ejecutable, visualizaciones y anotaciones de texto en un mismo documento.

Ventajas:

- Ideal para visualización de datos, análisis estadístico y aprendizaje automático.
- Integración con bibliotecas como Pandas, Matplotlib, Seaborn, Scikit-learn.
- Ampliamente usado en investigación, universidades y proyectos de IA.

Instalación con pip:

```
bash
```

```
pip install notebook
```

Ejecutar un notebook:

```
bash
```

```
jupyter notebook
```

Esto abrirá una interfaz web donde puedes crear, guardar y ejecutar tus notebooks.

Nota: Aunque son potentes, los notebooks no son recomendables para desarrollo de aplicaciones completas o proyectos en producción. Se usan como herramienta auxiliar.

La Biblia de Python

Recomendación final

Para un entorno de trabajo profesional con Python:

- Si estás comenzando o trabajas en proyectos generales: **VSCode** es ideal.
- Si desarrollas aplicaciones complejas o proyectos web grandes: **PyCharm Professional** te ofrecerá todo lo que necesitas.
- Si haces ciencia de datos o análisis exploratorio: **JupyterLab** es tu aliado.

Asegúrate también de versionar tu código desde el inicio con **Git** y de mantener un entorno virtual por cada proyecto, lo que veremos en detalle más adelante.

Tipos de Datos y Operadores en Python

Tipos de datos primitivos

Python es un lenguaje de tipado dinámico, lo que significa que no es necesario declarar el tipo de una variable al momento de definirla. El intérprete determina automáticamente el tipo según el valor asignado.

Números

Existen tres tipos principales:

- **int**: enteros (5, -10, 0)
- **float**: decimales (3.14, -0.001)
- **complex**: números complejos (1+2j)

```
python
```

```
entero = 42
```

```
decimal = 3.1416
```

```
complejo = 2 + 3j
```

Puedes usar funciones como `type()` para comprobar el tipo:

```
python
```

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

La Biblia de Python

Booleanos

Solo existen dos valores booleanos en Python:

```
python  
  
verdadero = True  
  
falso = False
```

Se usan para tomar decisiones en condiciones (`if`, `while`, etc.). Internamente, `True` equivale a 1 y `False` a 0.

Cadenas (str)

Son secuencias de caracteres:

```
python  
  
texto = "Hola, Python"  
  
nombre = 'Roberto'  
  
multilinea = """Esto  
es un texto  
de varias líneas"""
```

Puedes concatenar, repetir y usar formato:

```
python  
  
nombre = "Roberto"  
  
saludo = f"Hola, {nombre}" # f-strings desde Python 3.6
```

Colecciones: estructuras de datos fundamentales

Listas

Estructura mutable y ordenada. Permite elementos repetidos y de distintos tipos.

```
python  
  
frutas = ["manzana", "pera", "plátano"]
```


La Biblia de Python

```
frutas.append("naranja")
```

Soporta slicing:

```
python
```

```
print(frutas[1:3]) # ['pera', 'plátano']
```

Tuplas

Estructura inmutable. Útil para representar datos fijos.

```
python
```

```
coordenadas = (10.5, 20.3)
```

Se puede desempaquetar fácilmente:

```
python
```

```
x, y = coordenadas
```

Sets

Colección no ordenada y **sin duplicados**.

```
python
```

```
numeros = {1, 2, 3, 2, 1}
```

```
print(numeros) # {1, 2, 3}
```

Soporta operaciones de conjunto:

```
python
```

```
a = {1, 2, 3}
```

```
b = {3, 4, 5} print(a & b) # {3} print(a | b) # {1, 2, 3, 4, 5}
```

Diccionarios

Colección de pares clave-valor. Muy usada para representar objetos o estructuras tipo JSON.

```
python
```

La Biblia de Python

```
persona = {  
    "nombre": "Roberto",  
    "edad": 35,  
    "profesion": "Ingeniero"  
}  
print(persona["nombre"]) # Roberto
```

Operadores en Python

Operadores aritméticos

python

a + b # Suma

a - b # Resta

a * b # Multiplicación

a / b # División flotante

a // b # División entera

a % b # Módulo

a ** b # Potencia

Operadores de comparación

python

== # Igual

!= # Distinto

< # Menor que

> # Mayor que

<= # Menor o igual

>= # Mayor o igual

Operadores lógicos

La Biblia de Python

python

and # y lógico

or # o lógico

not # negación

Ejemplo:

python

edad = 25

es_adulto = edad >= 18 and edad <= 65

Operadores de identidad y pertenencia

python

Identidad

a is b

a is not b

Pertenencia 3 in [1, 2, 3]

True "Py" in "Python" # True

Conversión de tipos (type casting)

Python permite convertir explícitamente entre tipos cuando es necesario:

python

int("10") # 10

float("3.14") # 3.14

str(100) # "100"

list("abc") # ['a', 'b', 'c']

Conviene realizar estas conversiones de forma controlada para evitar errores de tipo.

La Biblia de Python

Tipado dinámico vs tipado estático

Python permite escribir código sin declarar tipos:

```
python
x = 10
x = "ahora soy una cadena"
```

Sin embargo, desde Python 3.5 en adelante se introdujo el **tipado opcional** con `type hints`:

```
python
def saludar(nombre: str) -> str:
    return f"Hola, {nombre}"
```

Este tipado no es obligatorio, pero herramientas como **mypy** o editores como PyCharm y VSCode pueden ayudarte a detectar errores antes de ejecutar el código.

Control de Flujo en Python

Condicionales: if, elif, else

Las estructuras condicionales permiten tomar decisiones basadas en condiciones lógicas. En Python, se usan las palabras clave `if`, `elif` (else if) y `else`.

Sintaxis básica

```
python

edad = 25

if edad >= 18:
    print("Es mayor de edad")

elif edad >= 13:
    print("Es adolescente")
```

La Biblia de Python

```
else:  
    print("Es niño")
```

Importante:

- La indentación es obligatoria (por convención, 4 espacios).
- No se usan llaves ni paréntesis como en otros lenguajes (C, Java).

Operadores comunes en condiciones:

```
python  
  
== # Igualdad  
  
!= # Distinto  
  
> # Mayor que  
  
< # Menor que  
  
>= # Mayor o igual  
  
<= # Menor o igual
```

Condiciones compuestas

```
python  
  
usuario = "admin"  
  
clave = "1234"  
  
if usuario == "admin" and clave == "1234":  
    print("Acceso permitido")
```

Buenas prácticas

- Usa condiciones claras y explícitas.
- Evita comparaciones innecesarias como `if activo == True:` y usa simplemente `if activo:`.

Bucles: for y while

Python tiene dos estructuras principales de repetición: `for` y `while`.

Bucle for

La Biblia de Python

El bucle `for` itera sobre elementos de una secuencia (lista, cadena, tupla, diccionario, rango, etc.).

```
python

nombres = ["Ana", "Luis", "Carlos"]

for nombre in nombres:

    print(f"Hola, {nombre}")
```

Usando `range()`:

```
python

for i in range(5):

    print(i)  # Imprime del 0 al 4
```

```
python

for i in range(2, 10, 2):  # inicio, fin, paso

    print(i)  # 2, 4, 6, 8
```

Bucle while

Repite un bloque mientras se cumpla una condición lógica.

```
python

contador = 0

while contador < 5:

    print(f"Contador: {contador}")

    contador += 1
```

Cuidado con bucles infinitos:

```
python

# while True:

#     print("Esto nunca se detendrá")
```

La Biblia de Python

Instrucciones útiles

- `break`: termina el bucle.
 - `continue`: salta a la siguiente iteración.
 - `else`: (poco común) se ejecuta si el bucle no se interrumpe con `break`.
-

python

```
for i in range(5):  
    if i == 3:  
        Break  
    else:  
        print("Bucle finalizado sin interrupciones")
```

Comprensión de listas (List Comprehensions)

Una de las características más potentes de Python es la capacidad de crear listas de forma concisa y expresiva mediante *list comprehensions*.

Ejemplo básico

python

```
cuadrados = [x**2 for x in range(10)]  
print(cuadrados)  # [0, 1, 4, 9, 16, ..., 81]
```

Con condiciones

python

```
pares = [x for x in range(20) if x % 2 == 0]
```

Equivalente a:

python

```
pares = []  
for x in range(20):
```

La Biblia de Python

```
if x % 2 == 0:
    pares.append(x)
```

Anidamiento

python

```
matriz = [[1, 2], [3, 4], [5, 6]]
```

```
plana = [num for fila in matriz for num in fila]
```

Con transformaciones

python

```
palabras = ["Python", "fluyente", "biblioteca"]
```

```
mayusculas = [p.upper() for p in palabras]
```

Cuándo usar cada estructura

Estructura	Usar cuando...
if / elif / else	Hay decisiones que tomar según condiciones
for	Se necesita iterar sobre elementos conocidos
while	La condición de parada es desconocida o depende del tiempo
list comprehension	Se necesita construir una nueva lista de forma limpia y concisa

La Biblia de Python

Ejercicio propuesto:

Escribe un programa que pida una lista de números al usuario (separados por comas), y muestre una lista nueva con los valores multiplicados por 3, solo si son mayores que 5.

```
python

entrada = input("Introduce números separados por coma: ")
numeros = [int(n) for n in entrada.split(",")]
resultado = [n * 3 for n in numeros if n > 5]

print(resultado)
```

Manejo de errores y excepciones en Python

¿Qué es una excepción?

Una **excepción** es una señal de que algo inesperado ocurrió durante la ejecución del programa. Puede tratarse de un archivo que no se encuentra, una división por cero, un acceso inválido a una lista, o un error del usuario.

Si no se maneja adecuadamente, una excepción detendrá el programa con un mensaje de error.

Ejemplo de error sin control:

```
python

numero = int(input("Introduce un número: "))

print(10 / numero)
```

Si el usuario introduce 0, el programa lanza una excepción:

```
vbnet

ZeroDivisionError: division by zero
```

try / except: atrapando errores

La forma correcta de manejar estos casos es usando la instrucción `try`.

```
python

try:
```

La Biblia de Python

```
numero = int(input("Introduce un número: "))

resultado = 10 / numero

print(f"Resultado: {resultado}")

except ZeroDivisionError:

    print("No puedes dividir entre cero.")

except ValueError:

    print("Entrada inválida: se esperaba un número.")
```

Con esto, el programa continúa su ejecución sin cerrarse de forma abrupta.

Uso de else y finally

- `else` se ejecuta si **no se lanza ninguna excepción**.
- `finally` se ejecuta **siempre**, haya o no haya error. Se usa para liberar recursos (como cerrar archivos o conexiones).

```
python

try:

    archivo = open("datos.txt", "r")

    contenido = archivo.read()

except FileNotFoundError:

    print("El archivo no existe.")

else:

    print("Lectura exitosa.")

finally:

    print("Cerrando el archivo.")

    archivo.close()
```

Tipos comunes de excepciones

La Biblia de Python

Excepción	Descripción
ZeroDivisionError	División por cero
ValueError	Conversión de tipo inválida
TypeError	Operación con tipos incompatibles
IndexError	Acceso fuera del rango de una lista
KeyError	Acceso a una clave inexistente en un diccionario
FileNotFoundError	Archivo no encontrado
ImportError	Error al importar un módulo

Definir tus propias excepciones

Puedes crear excepciones personalizadas que hereden de `Exception` para casos específicos de tu aplicación.

```
python
```

```
class SaldoInsuficiente(Exception):  
    pass  
  
def retirar(saldo, cantidad):  
    if cantidad > saldo:  
        raise SaldoInsuficiente("Fondos insuficientes")  
    return saldo - cantidad
```

Uso:

```
python
```

```
try:  
    nuevo_saldo = retirar(100, 150)  
except SaldoInsuficiente as e:  
    print(f"Error: {e}")
```

La Biblia de Python

Buenas prácticas en el manejo de errores

- Captura **solo las excepciones que puedas manejar** correctamente.
- No uses `except:` sin especificar el tipo, salvo que sea estrictamente necesario.
- Utiliza mensajes claros para el usuario y logs detallados para desarrolladores.
- Evita usar excepciones como control de flujo general del programa.

Funciones y Módulos

¿Qué es una función?

Una **función** es un bloque de código reutilizable que se ejecuta cuando se le llama por su nombre. Permite dividir un programa en partes pequeñas, facilitando su lectura, mantenimiento y pruebas.

Definición básica

```
python
```

```
def saludar():  
    print("Hola desde una función")
```

Llamada:

```
python
```

```
saludar()
```

Argumentos y parámetros

Funciones con parámetros

```
python
```

```
def saludar(nombre):  
    print(f"Hola, {nombre}")
```

```
python
```

```
saludar("Roberto")
```

La Biblia de Python

Parámetros con valores por defecto

python

```
def saludar(nombre="desconocido"):
    print(f"Hola, {nombre}")
```

Parámetros posicionales vs nombrados

python

```
def perfil(nombre, edad):
    print(f"{nombre} tiene {edad} años")
```

```
perfil("Ana", 30)                # Posicional
```

```
perfil(edad=30, nombre="Ana")    # Nombrado
```

Retorno de valores

Las funciones pueden devolver resultados con `return`:

python

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

```
resultado = suma(3, 5)
```

Una función sin `return` explícito devuelve `None`.

Ámbito de las variables (scope)

Python tiene reglas claras para la visibilidad de las variables. Las variables definidas dentro de una función son **locales**, y las externas son **globales**.

python

La Biblia de Python

```
x = 10 # Variable global

def mostrar():
    x = 5 # Variable local
    print(x)

mostrar() # 5

print(x) # 10
```

La palabra clave `global`

Permite modificar una variable global desde dentro de una función (se debe usar con cuidado):

```
python

contador = 0

def incrementar():
    global contador
    contador += 1
```

Funciones anónimas: `lambda`

Una **lambda** es una función pequeña, anónima, y generalmente de una sola línea.

```
python

doble = lambda x: x * 2

print(doble(4)) # 8
```

Son útiles cuando necesitas una función rápida y sin nombre, por ejemplo, en filtros o ordenamientos.

La Biblia de Python

Funciones integradas de orden superior: `map`, `filter`, `reduce`

`map()`

Aplica una función a cada elemento de una lista.

```
python
numeros = [1, 2, 3]
cuadrados = list(map(lambda x: x**2, numeros))
```

`filter()`

Filtra elementos que cumplan una condición.

```
python
pares = list(filter(lambda x: x % 2 == 0, range(10)))
```

`reduce()`

Aplica una función acumulativa a los elementos. Requiere importar desde `functools`.

```
python
from functools import reduce
numeros = [1, 2, 3, 4]
producto = reduce(lambda x, y: x * y, numeros) # 24
```

Documentación y anotaciones de tipo

Docstrings

Una buena práctica es documentar todas tus funciones.

```
python
def saludar(nombre: str) -> str:
```

La Biblia de Python

```
"""
Devuelve un saludo personalizado.

:param nombre: Nombre de la persona
:return: Cadena con el saludo
"""

return f"Hola, {nombre}"
```

Anotaciones de tipo

Python permite declarar tipos de entrada y salida, lo cual mejora la legibilidad y permite que editores y linters detecten errores antes de tiempo.

```
python

def elevar(x: int, y: int) -> int:

    return x ** y
```

Estas anotaciones no son obligatorias pero se consideran una buena práctica profesional, especialmente en proyectos grandes.

Módulos: reutilización y organización del código

Un **módulo** es un archivo `.py` que contiene funciones, clases o variables y puede ser importado en otros archivos.

Crear un módulo

Supongamos que tienes un archivo `utilidades.py` con:

```
python

def saludar(nombre):

    return f"Hola, {nombre}"
```


La Biblia de Python

Puedes usarlo desde otro archivo:

```
python

import utilidades

print(utilidades.saludar("Python"))
```

Importar partes específicas

```
python

from utilidades import saludar
```

También se pueden usar alias:

```
python

import utilidades as ut
```

Paquetes

Un **paquete** es un conjunto de módulos organizados en carpetas con un archivo `__init__.py`. Es la base para crear bibliotecas o aplicaciones distribuidas profesionalmente.

```
bash

mi_paquete/
├── __init__.py
├── modulo1.py
└── modulo2.py
```

Esto permite importar como:

```
python

from mi_paquete.modulo1 import funcion
```

La Biblia de Python

Ejercicios recomendados

1. Tipos de datos y operaciones básicas

Ejercicio 1.1 – Conversión de temperaturas

Crea un programa que solicite una temperatura en grados Celsius y la convierta a Fahrenheit. Muestra el resultado con dos decimales.

Ejercicio 1.2 – Análisis de cadenas

Solicita al usuario una cadena y muestra:

- Número de caracteres
- Primera y última letra
- Cuántas veces aparece la letra “a” (o “A”)

Ejercicio 1.3 – Operaciones aritméticas

Dado un número entero positivo, calcula y muestra su cuadrado, raíz cuadrada, módulo 10 y si es par o impar.

2. Estructuras de datos

Ejercicio 2.1 – Lista de tareas

Solicita al usuario una lista de tareas separadas por comas. Almacena los datos en una lista y muestra:

- Total de tareas
- Primera y última tarea
- Lista ordenada alfabéticamente

Ejercicio 2.2 – Diccionario de productos

Crea un diccionario con nombres de productos como claves y precios como valores. Permite consultar el precio de un producto introducido por el usuario.

Ejercicio 2.3 – Conjunto de emails únicos

Solicita al usuario 5 emails (pueden repetirse) y muestra solo los valores únicos utilizando un set.

3. Control de flujo

Ejercicio 3.1 – Clasificador de edad

Pide la edad al usuario y clasificala en: niño (0–12), adolescente (13–17), adulto (18–64) o senior (65+).

Ejercicio 3.2 – Contador de pares e impares

Solicita 10 números y cuenta cuántos son pares y cuántos impares. Usa un bucle `for`.

La Biblia de Python

Ejercicio 3.3 – Contraseña segura

Pide una contraseña al usuario. Repite la solicitud hasta que cumpla:

- Mínimo 8 caracteres
 - Al menos una mayúscula
 - Al menos un número
-

4. Funciones y reutilización

Ejercicio 4.1 – Función calculadora

Crea una función que reciba dos números y una operación (+, -, *, /) y devuelva el resultado. Controla errores (división por cero, operación inválida).

Ejercicio 4.2 – Función de validación de email

Crea una función que reciba un string y devuelva `True` si contiene un @ y un punto . en los lugares adecuados.

Ejercicio 4.3 – Map, filter y lambda

Dada una lista de números:

- Usa `map` para elevarlos al cuadrado
 - Usa `filter` para quedarte solo con los pares
-

5. Proyecto integrador de la Parte 1

Ejercicio 5.1 – Gestor de estudiantes (mini-proyecto)

Crea un programa que permita:

- Registrar estudiantes con nombre, edad y nota final
- Listarlos todos
- Mostrar el promedio de notas
- Mostrar el estudiante con mejor nota
- Utiliza funciones para organizar el código

La Biblia de Python

Capítulo 2: Programación Orientada a Objetos

Introducción a la Programación Orientada a Objetos (POO)

¿Qué es la programación orientada a objetos?

La programación orientada a objetos (POO) es un **paradigma de programación** que organiza el código en torno a **objetos**, en lugar de funciones sueltas o listas de instrucciones. Un objeto es una entidad que **representa algo del mundo real**: una persona, un coche, una cuenta bancaria, un botón en una aplicación.

Cada objeto tiene:

- **Atributos**: características o propiedades (por ejemplo, un coche tiene color, marca, velocidad).
- **Métodos**: acciones que puede realizar (por ejemplo, un coche puede arrancar, frenar, girar).

La POO se basa en **modelar el mundo real** dentro del código, permitiendo construir aplicaciones más claras, mantenibles, escalables y reutilizables.

¿Por qué usar la POO?

1. **Organiza mejor el código**: agrupa datos y comportamiento relacionados.
2. **Facilita la reutilización**: puedes crear nuevas funcionalidades a partir de clases existentes.
3. **Permite simular entidades reales**: muy útil en videojuegos, aplicaciones web, software financiero, etc.
4. **Hace el mantenimiento más fácil**: es más sencillo entender y modificar el código cuando está bien estructurado en objetos.

Una analogía sencilla: el plano de una casa

Imagina que quieres construir 10 casas iguales. En lugar de repetir todo, creas un **plano (clase)** que define cómo debe ser cada casa: cuántas habitaciones, cuántas ventanas, de qué color será, etc.

Luego, puedes construir tantas casas (**objetos**) como quieras, todas basadas en ese plano. Cada una puede tener **valores distintos** aunque compartan la misma estructura.

La Biblia de Python

Cómo se traduce esto en Python

python

```
class Perro:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

    def ladrar(self):

        print(f"{self.nombre} dice: ¡Guau!")

# Crear un objeto (instancia)

mi_perro = Perro("Firulais", 3)

mi_perro.ladrar()
```

En este ejemplo:

- `Perro` es una **clase** (el plano).
- `mi_perro` es un **objeto** (una casa construida a partir del plano).
- `nombre` y `edad` son **atributos**.
- `ladrar()` es un **método**.

Características clave de la POO

En esta parte del libro aprenderás los **principios fundamentales** de la POO, que incluyen:

- **Clases y objetos:** cómo se definen y cómo se relacionan.
- **Atributos y métodos:** qué hacen y cómo se accede a ellos.
- **Encapsulamiento:** proteger los datos internos del objeto.
- **Herencia:** crear nuevas clases a partir de otras ya existentes.
- **Polimorfismo:** permitir que distintas clases respondan de forma distinta al mismo método.
- **Abstracción:** ocultar detalles internos y mostrar solo lo necesario.

Además, aprenderás a usar **decoradores útiles como** `@classmethod` y `@staticmethod`, y a aplicar **patrones de diseño como Singleton, Factory y Strategy**, fundamentales para escribir código limpio, profesional y extensible.

La Biblia de Python

¿Es difícil aprender POO?

No. Es un cambio de mentalidad. Si estás acostumbrado a escribir funciones independientes, al principio puede parecer extraño. Pero una vez comprendes los conceptos y los aplicas con ejemplos reales, se vuelve natural. Este capítulo está diseñado para acompañarte paso a paso.

Clases, objetos, atributos y métodos

¿Qué es una clase?

Una **clase** es un plano, molde o plantilla que define cómo serán los objetos creados a partir de ella. Describe:

- qué datos (atributos) tendrán los objetos,
- qué acciones (métodos) podrán realizar.

En Python, se define con la palabra clave `class`.

Ejemplo básico:

```
python
```

```
class Persona:
```

```
    pass
```

Esto crea una clase vacía llamada `Persona`. No hace nada todavía, pero sirve como punto de partida para construir objetos.

¿Qué es un objeto?

Un **objeto** es una instancia concreta de una clase. Cuando creas un objeto, estás generando una copia funcional de la clase con datos reales.

```
python
```

```
personal = Persona()
```

```
persona2 = Persona()
```

La Biblia de Python

Ambos son objetos del tipo `Persona`.

Atributos de instancia

Los **atributos** representan el estado del objeto. Se definen usualmente dentro del método especial `__init__()` (constructor), que se ejecuta automáticamente al crear el objeto.

python

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad
```

- `__init__()` es el **constructor**.
- `self` representa al objeto actual.
- `self.nombre` y `self.edad` son atributos asignados a cada objeto.

Crear un objeto con valores

python

```
juan = Persona("Juan", 30)

print(juan.nombre)  # Juan

print(juan.edad)    # 30
```

Métodos: comportamiento de los objetos

Un **método** es una función definida dentro de una clase que actúa sobre los atributos del objeto o realiza alguna operación relacionada con él.

python

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad
```

La Biblia de Python

```
def saludar(self):  
    print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años.")
```

python

```
maria = Persona("María", 28)
```

```
maria.saludar()# Output: Hola, me llamo María y tengo 28 años.
```

Atributos de clase vs de instancia

Los **atributos de clase** son compartidos por todas las instancias. Se definen fuera del `__init__`.

python

```
class Persona:
```

```
    especie = "humano" # Atributo de clase
```

```
    def __init__(self, nombre):
```

```
        self.nombre = nombre # Atributo de instancia
```

python

```
p1 = Persona("Carlos")
```

```
p2 = Persona("Lucía")
```

```
print(p1.especie) # humano
```

```
print(p2.especie) # humano
```

Modificar atributos

Puedes modificar atributos directamente:

python

La Biblia de Python

```
p1.edad = 35
```

```
p1.nombre = "Carlos Jr."
```

También es buena práctica encapsular esta lógica usando métodos (lo veremos con más detalle en encapsulamiento).

Representación de objetos

Python permite definir cómo se muestra un objeto al imprimirlo, mediante el método especial `__str__()`:

```
python
```

```
class Persona:
```

```
    def __init__(self, nombre, edad):
```

```
        self.nombre = nombre
```

```
        self.edad = edad
```

```
    def __str__(self):
```

```
        return f"{self.nombre}, {self.edad} años"
```

```
python
```

```
CopiarEditar
```

```
p = Persona("Roberto", 36)
```

```
print(p) # Output: Roberto, 36 años
```

Ejemplo práctico completo

```
python
```

```
class CuentaBancaria:
```

```
    def __init__(self, titular, saldo=0):
```

```
        self.titular = titular
```

```
        self.saldo = saldo
```

La Biblia de Python

```
def depositar(self, cantidad):  
    self.saldo += cantidad  
  
def retirar(self, cantidad):  
    if cantidad <= self.saldo:  
        self.saldo -= cantidad  
    else:  
        print("Fondos insuficientes")  
  
def mostrar_saldo(self):  
    return f"Saldo actual: {self.saldo} €"
```

Uso:

python

```
cuenta = CuentaBancaria("Ana")  
cuenta.depositar(1000)  
cuenta.retirar(300)  
print(cuenta.mostrar_saldo()) # Saldo actual: 700€
```

Este ejemplo ya representa una lógica mínima **real y testable**, que puede crecer con validaciones, logs, excepciones o pruebas unitarias.

Herencia, Polimorfismo y Encapsulamiento

Herencia: reutilizar código de otras clases

La **herencia** permite crear nuevas clases basadas en una clase existente. La clase hija (o derivada) hereda atributos y métodos de la clase padre (o base), y puede extenderlos o modificarlos.

La Biblia de Python

Sintaxis básica

python

```
class Animal:

    def __init__(self, nombre):

        self.nombre = nombre

    def hablar(self):

        print("Este animal hace un sonido.")

class Perro(Animal):

    def hablar(self):

        print(f"{self.nombre} dice: ¡Guau!")
```

```
class Gato(Animal):

    def hablar(self):

        print(f"{self.nombre} dice: Miau")
```

python

```
p = Perro("Toby")
```

```
g = Gato("Michi")
```

```
p.hablar() # Toby dice: ¡Guau!
```

```
g.hablar() # Michi dice: Miau
```

En este ejemplo:

- Perro y Gato heredan de Animal.

La Biblia de Python

- Cada subclase puede redefinir (`override`) métodos del padre.

Herencia múltiple

Python permite que una clase herede de varias clases, aunque se recomienda con precaución:

```
python
```

```
class Volador:
    def volar(self):
        print("Volando...")
```

```
class Nadador:
    def nadar(self):
        print("Nadando...")
```

```
class Pato(Volador, Nadador):
    pass
```

Polimorfismo: una misma interfaz, múltiples comportamientos

El **polimorfismo** permite usar una misma interfaz para distintos tipos de objetos. En otras palabras, puedes llamar al mismo método sobre objetos de diferentes clases, y cada uno responderá a su manera.

Ya vimos este comportamiento en `hablar()` de `Perro` y `Gato`.

Ejemplo práctico con una función genérica:

```
python
```

```
def hacer_hablar(animal):
    animal.hablar()
```

```
hacer_hablar(Perro("Max"))    # Max dice: ¡Guau!
```

```
hacer_hablar(Gato("Luna"))    # Luna dice: Miau
```

La Biblia de Python

Esto permite **escribir código genérico** que funcione con cualquier clase que implemente una determinada interfaz (método).

Encapsulamiento: proteger el estado interno

El **encapsulamiento** busca proteger los datos internos del objeto, permitiendo acceder o modificarlos solo a través de métodos específicos. Es una práctica que ayuda a evitar errores y a mantener la integridad del estado de un objeto.

Niveles de acceso en Python

Python no tiene modificadores de acceso estrictos como `private`, `protected` o `public`, pero usa convenciones:

Notación	Nivel de acceso	Uso común
atributo	público	acceso libre
<code>_atributo</code>	protegido	uso interno (por convención)
<code>__atributo</code>	privado	nombre "mangleado"

Ejemplo:

python

```
class Cuenta:
```

```
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo # atributo privado
```

```
    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad
```

```
    def mostrar_saldo(self):
```

La Biblia de Python

```
return self.__saldo
```

```
python
```

```
c = Cuenta("Roberto", 1000)
c.depositar(500)
print(c.mostrar_saldo()) # 1500
# print(c.__saldo) # Error: atributo privado
```

Propiedades con `@property`

Una forma profesional de acceder a atributos privados es usando `@property`, que permite definir **getters** y **setters** de manera elegante:

```
python
```

```
class Usuario:
    def __init__(self, nombre):
        self._nombre = nombre

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, nuevo_nombre):
        if len(nuevo_nombre) > 0:
            self._nombre = nuevo_nombre
```

```
python
```

```
u = Usuario("Luis") print(u.nombre)           # Luis
u.nombre = "Andrés" print(u.nombre)           # Andrés
```

La Biblia de Python

Cuándo aplicar cada uno

- Usa **herencia** cuando compartas lógica entre clases similares.
- Aplica **polimorfismo** cuando diferentes clases deban responder a una misma acción.
- Usa **encapsulamiento** para proteger y controlar el acceso a los datos internos.

Decoradores `@classmethod` y `@staticmethod`

¿Qué es un decorador en Python?

Un decorador es una función especial que **modifica el comportamiento de otra función o método**, sin cambiar su código interno. En el contexto de clases, hay dos decoradores muy útiles:

- `@classmethod`
- `@staticmethod`

Ambos permiten **definir métodos que no necesitan una instancia para ejecutarse**, pero tienen usos diferentes.

`@classmethod`: métodos a nivel de clase

Un **método de clase** recibe como primer argumento `cls` (en lugar de `self`). Esto le da acceso a la **clase en sí**, no al objeto particular.

¿Para qué se usa?

- Crear **constructores alternativos**
- Manipular o acceder a **atributos de clase**
- Implementar **patrones de diseño Factory**

Ejemplo práctico: constructor alternativo

python

```
class Libro:

    def __init__(self, titulo, paginas):

        self.titulo = titulo

        self.paginas = paginas
```

La Biblia de Python

```
@classmethod
def desde_string(cls, cadena):
    partes = cadena.split("-")
    titulo = partes[0].strip()
    paginas = int(partes[1])
    return cls(titulo, paginas)
```

Uso:

```
python

entrada = "Python Avanzado - 320"

libro = Libro.desde_string(entrada)

print(libro.titulo)  # Python Avanzado
```

Acceso a atributos de clase

```
python

class Usuario:
    contador = 0

    def __init__(self, nombre):
        self.nombre = nombre
        Usuario.contador += 1
```

```
@classmethod
def total_usuarios(cls):
```


La Biblia de Python

```
return cls.contador
```

```
python
```

```
u1 = Usuario("Ana")
```

```
u2 = Usuario("Pedro")print(Usuario.total_usuarios()) # 2
```

`@staticmethod`: **métodos utilitarios sin acceso a instancia ni clase**

Un **método estático** es una función que **no necesita ni `self` ni `cls`**. Se comporta como una función normal, pero **se agrupa dentro de la clase por organización lógica**.

¿Cuándo usarlo?

- Cuando el método no depende de atributos del objeto ni de la clase.
- Para **funciones auxiliares** relacionadas conceptualmente con la clase.

Ejemplo:

```
python
```

```
class Calculadora:    @staticmethod
```

```
    def sumar(a, b):
```

```
        return a + b
```

```
    @staticmethod
```

```
    def es_par(n):
```

```
        return n % 2 == 0
```

Uso:

```
python
```

```
print(Calculadora.sumar(10, 20)) # 30
```

```
print(Calculadora.es_par(7))      # False
```

Esto evita tener funciones “sueltas” y mantiene el código mejor estructurado.

La Biblia de Python

Comparación rápida

Tipo de método	Primer parámetro	Accede a self	Accede a cls	Cuándo usar
método normal	self	Sí	No	Lógica dependiente del objeto
@classmethod	cls	No	Sí	Constructores alternativos, factories
@staticmethod	Ninguno	No	No	Utilidades que no acceden a nada

Patrones de Diseño en Python

Los **patrones de diseño** son soluciones probadas y reutilizables a problemas comunes en el diseño de software. No son recetas exactas, sino **modelos mentales y estructurales** que ayudan a organizar el código de manera más robusta, mantenible y flexible.

Patrón Singleton

Un solo objeto compartido globalmente

Objetivo: asegurar que una clase tenga **una única instancia**, y proporcionar un punto de acceso global a ella.

Este patrón se usa, por ejemplo, para manejar una conexión a base de datos, configuración de aplicación, logs, etc.

Implementación básica en Python:

```
python
```

```
class Singleton:
```

```
    _instancia = None
```

```
    def __new__(cls):
```

```
        if cls._instancia is None:
```

La Biblia de Python

```
        cls._instancia = super().__new__(cls)

    return cls._instancia
```

python

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

Mejora profesional con inicialización controlada:

python

```
class Configuracion:

    _instancia = None

    def __new__(cls, entorno="producción"):

        if cls._instancia is None:

            cls._instancia = super().__new__(cls)

            cls._instancia.entorno = entorno

        return cls._instancia
```

Patrón Factory

Crear objetos sin exponer la lógica de creación

Objetivo: encapsular la lógica de creación de objetos en una clase o método, especialmente cuando las clases concretas pueden variar según el contexto.

Ejemplo: sistema de notificaciones

python

```
class Notificacion:

    def enviar(self, mensaje):

        raise NotImplementedError
```

La Biblia de Python

```
class Email(Notificacion):  
    def enviar(self, mensaje):  
        print(f"Enviando email: {mensaje}")  
  
class SMS(Notificacion):  
    def enviar(self, mensaje):  
        print(f"Enviando SMS: {mensaje}")  
  
class NotificacionFactory:  
    @staticmethod  
    def crear(tipo):  
        if tipo == "email":  
            return Email()  
        elif tipo == "sms":  
            return SMS()  
        else:  
            raise ValueError("Tipo no soportado")
```

Uso:

python

```
notificador = NotificacionFactory.crear("email")  
notificador.enviar("Hola mundo") # Enviando email: Hola mundo
```

Este patrón **desacopla** el código cliente de las clases concretas que se están instanciando.

La Biblia de Python

Patrón Strategy

Cambiar el comportamiento en tiempo de ejecución

Objetivo: definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. El patrón Strategy permite cambiar el algoritmo usado por un objeto sin modificar su código.

Ejemplo: cálculo de impuestos

python

```
class Impuesto:

    def calcular(self, monto):

        raise NotImplementedError


class IVA(Impuesto):

    def calcular(self, monto):

        return monto * 0.21


class ISV(Impuesto):

    def calcular(self, monto):

        return monto * 0.15


class CalculadoraImpuestos:

    def __init__(self, estrategia: Impuesto):

        self.estrategia = estrategia

    def calcular_total(self, monto):

        return monto + self.estrategia.calcular(monto)
```

Uso:

python

La Biblia de Python

```
iva = IVA()

calculadora =
CalculadoraImpuestos(iva)print(calculadora.calcular_total(100)) # 121.0

# Cambiar la estrategia en tiempo de ejecución

calculadora.estrategia = ISV()print(calculadora.calcular_total(100)) #
115.0
```

Este patrón es muy útil para:

- motores de validación
- procesamiento de pagos
- Serialización
- algoritmos configurables.

¿Cómo testear estos patrones?

- El **Singleton** se prueba verificando que todas las instancias apunten al mismo objeto.
- El **Factory** se testea comprobando que devuelve la clase correcta según el tipo.
- El **Strategy** se verifica probando distintos algoritmos con los mismos datos de entrada.

Ejercicios recomendados

1. Clases, atributos y métodos

Ejercicio 2.1 – Clase Rectángulo

Crea una clase `Rectangulo` que tenga como atributos base y altura. Incluye métodos para:

- calcular el área
- calcular el perímetro
- representar el objeto con `__str__`.

Ejercicio 2.2 – Clase Persona con validación

Crea una clase `Persona` con nombre y edad. Valida en el constructor que la edad sea mayor que 0. Si no, lanza un `ValueError`.

La Biblia de Python

Ejercicio 2.3 – Contador de instancias

Crea una clase `Empleado` que tenga un contador de cuántas instancias se han creado, usando un atributo de clase y un `@classmethod`.

2. Herencia y polimorfismo

Ejercicio 2.4 – Jerarquía de vehículos

Define una clase base `Vehiculo` con atributos `marca` y `modelo`, y un método `descripcion()`. Luego, crea clases `Coche`, `Moto` y `Camion` que hereden de `Vehiculo` y sobrescriban el método `descripcion()`.

Ejercicio 2.5 – Zoo polimórfico

Crea una clase base `Animal` con un método `hacer_sonido()`. Luego crea subclases `Leon`, `Loro`, `Serpiente`, y haz que cada una emita un sonido distinto. Implementa una función que reciba una lista de animales y los haga "hablar".

Ejercicio 2.6 – Facturas con descuentos (herencia)

Crea una clase base `Factura` con un método `total()`. Crea una subclase `FacturaConDescuento` que aplique un 10% si el total supera 100€. Usa herencia correctamente.

3. Encapsulamiento y propiedades

Ejercicio 2.7 – Cuenta bancaria segura

Implementa una clase `CuentaBancaria` con saldo privado (`__saldo`). Usa métodos públicos para depositar y retirar dinero. Agrega validaciones (no se puede retirar más de lo disponible).

Ejercicio 2.8 – Producto con control de precio

Crea una clase `Producto` con atributos `nombre` y `precio`. Usa `@property` para permitir acceder y modificar el precio, pero con validación: nunca puede ser negativo.

Ejercicio 2.9 – Clase Inmutable

Implementa una clase `ConfiguracionSistema` con atributos solo de lectura: `modo`, `version`, `autor`. Una vez instanciada, los atributos no deben poder cambiarse.

La Biblia de Python

4. Métodos estáticos y de clase

Ejercicio 2.10 – Utilidades matemáticas

Crea una clase `Matematica` con métodos estáticos:

- `es_primo(n)`
 - `factorial(n)`
 - `fibonacci(n)`
-

Ejercicio 2.11 – Registro de usuarios

Crea una clase `Usuario` con un atributo de clase `cantidad_usuarios`. Incrementa este contador cada vez que se cree un nuevo usuario, usando un `@classmethod`.

5. Patrones de diseño

Ejercicio 2.12 – Singleton: Logger

Crea una clase `Logger` que siempre retorne la misma instancia. Debe permitir escribir mensajes en consola y guardar un historial de logs.

Ejercicio 2.13 – Factory: Transacciones

Crea una clase `TransaccionFactory` que devuelva objetos de las clases `Transferencia`, `Deposito`, `Retiro` según un string. Cada clase debe implementar un método `ejecutar()`.

Ejercicio 2.14 – Strategy: Procesador de pagos

Crea una clase `ProcesadorPagos` que acepte estrategias como `PagoConTarjeta`, `PagoConPayPal` y `PagoConCripto`, todas con el método `procesar(monto)`. Cambia la estrategia dinámicamente.

6. Mini-proyecto integrador de POO

Ejercicio 2.15 – Sistema de gestión de biblioteca

Construye un sistema básico con:

- Clase `Libro` (título, autor, disponible)
- Clase `Usuario` (nombre, historial)
- Clase `Biblioteca` que permita:
 - añadir libros

La Biblia de Python

- prestar libros
- devolver libros
- mostrar historial de usuario.

Aplica encapsulamiento, validación, uso de listas, y métodos estáticos donde tenga sentido.