01. Python

Instalación de Python

1. Verificar si Python está instalado ejecutando `python --version` o `python3 --version` en la terminal.
2. Si no está instalado, descargar el instalador desde python.org.
3. Seguir las indicaciones de instalación y marcar la opción "Agregar Python a PATH" para facilitar el acceso desde la línea de comandos.
4. Verificar la instalación ejecutando `python --version`.

Extensiones para Visual Code

- Python (https://marketplace.visualstudio.com/items?itemName=ms-python.python)
- Python Debugger (https://marketplace.visualstudio.com/items?itemName=ms-python.debugpy)

Creación de un entorno virtual

```
python -m venv mi_entorno
source mi_entorno/bin/activate  # Mac/Linux
mi_entorno\Scripts\activate     # Windows
```

Variables, tipos de datos y operadores

Una variable es un nombre que hace referencia a un valor almacenado en memoria; es decir, son usadas para almacenar y manipular datos a través del código.

En Python las variables son tipadas dinámicamente, lo que quiere decir que no se necesita declarar el tipo al definirlas.

1️⃣ Tipos de datos en Python

2️⃣ Operadores en Python

Enteros (`int`): Números sin decimales

Flotantes (`float`): Números con decimales

Cadenas (`str`): Secuencias de caracteres

Booleanos (`bool`): Valores `True` o `False`.

Listas (`list`): Colecciones ordenadas de elementos

Tuplas (`tuple`): Listas inmutables

Diccionarios (`dict`): Pares clave-valor

Aritméticos (`+`, `-`, `*`, `/`, `//` — División entera —, `%` — módulo —, `**` — exponenciación —).

Comparación (`==`, `!=`, `>`, `<`, `>=`, `<=`).

Lógicos (`and`, `or`, `not`).

Asignación (`=`, `+=`, `-=`, `*=`, `/=`).

Pertenencia (`in`, `not in`).

Conversión de tipos en Python y manipulación de cadenas en Python

1️⃣ Conversion de tipos en Python

Permite convertir un tipo de dato a otro, ya sea de forma implícita o explícita.

La conversión implícita ocurre automáticamente cuando es necesario, por ejemplo, al sumar un entero con un flotante, Python convierte el entero a flotante.

La conversión explícita se realiza mediante funciones como `int()`, `float()`, `str()`, `bool()`, ... para convertir variables a un tipo específico.

2️⃣ Manipulación de cadenas en Python

Implica varias operaciones como concatenación, formateo y uso de métodos de cadenas para manipular datos de texto.

La concatenación de cadenas permite combinar cadenas utilizando el operador `+` para crear cadenas nuevas.

El formateo de cadenas se realiza mediante **f-strings** (Python 3.6+) o el método `format()` para insertar variables o expresiones en una cadena.

Proporciona numerosos métodos de cadenas para manipular texto de manera eficiente, como `upper()`, `lower()`, `strip()`, `replace()`, `split()`, `join()`, ...

```
# Conversion de tipos
x = 10
y = 20.5
z = "30"
suma = x + y + int(z)  # Convertir z a entero
print(f"La suma es: {suma}") # La suma es: 60.5

# Manipulación de cadenas
nombre = "Juan"
apellido = "Pérez"
nombre_completo = nombre + " " + apellido  # Concatenación
print(f"Nombre completo: {nombre_completo}") # Nombre completo: Juan Pérez

edad = 25
mensaje = f"Hola, me llamo {nombre} {apellido} y tengo {edad} años."  # F-strings
print(mensaje) # Hola, me llamo Juan Pérez y tengo 25 años.

texto = "   Hola, ¿cómo estás?   "
print(texto.upper())  # Convertir a mayúsculas → "   HOLA, ¿CÓMO ESTÁS?   "
print(texto.strip())  # Eliminar espacios en blanco → "Hola, ¿cómo estás?"
print(texto.replace("estás", "te encuentras"))  # Reemplazar texto → "   Hola, ¿cómo te encuentras?   "
```

Listas, tuplas y diccionarios

1️⃣ Listas (`list`, `[]`)

- Secuencias mutables en Python
- Permite almacenar colecciones de elementos
- Ideal en situaciones donde los elementos necesitan ser actualizados o modificados

2️⃣ Tuplas (`tuple`, `()`)

- Secuencias inmutables
- Usadas para colecciones de elementos fijas que no cambiarán
- Generalmente utilizadas como claves en diccionarios debido a su inmutabilidad

3️⃣ Diccionarios (`dict` , `{}`)

- Mapeos mutables
- Almacena parejas clave-valor
- Esencial para crear arreglos (*arrays*) asociativos o mapeos donde claves únicas están asociadas con valores específicos.

Trabajando con listas

Hay varias maneras de recorrerlas: Usando un bucle `for` (sólo valores), `for` en conjunción con `enumerate` (índices y valores) y `while`

```
lista = [1, 2, 3]
# Adición de elementos
lista.append(4)  # [1, 2, 3, 4]
lista.extend([5, 6])  # [1, 2, 3, 4, 5, 6]
lista.insert(1, 99)  # [1, 99, 2, 3, 4, 5, 6]

# Eliminación de elementos
lista.remove(99)  # [1, 2, 3, 4, 5, 6]
eliminado = lista.pop(2)  # eliminado = 3, lista = [1, 2, 4, 5, 6]
lista.clear()  # []

numeros = [3, 1, 4, 2]
# Ordenación y búsqueda
numeros.sort()  # [1, 2, 3, 4]
nueva_lista = sorted(numeros, reverse=True)  # [4, 3, 2, 1]
posicion = numeros.index(3)  # 2
repeticiones = numeros.count(3)  # 1

# Otras operaciones
numeros.reverse()  # [4, 3, 2, 1]
copia = numeros.copy()

##### RECORRIENDO LISTAS #####
```

```
numeros = [10, 20, 30, 40]

# for
for num in numeros:
    print(num)

# for ... enumerate
for i, num in enumerate(numeros):
    print(f"Índice {i}: {num}")

# while (hay que calcular la longitud para tener una condición de parada)
while i < len(numeros):
    print(numero[i])
    i += 1
```

Trabajando con tuplas

Las tuplas son inmutables, por lo que tienen menos métodos que las listas.

```
tupla = (1, 2, 3, 2, 2)
# Operaciones básicas
tupla.count(2)  # 3
tupla.index(3)  # 2

# Conversión a lista, añadir un elemento y reconvertir a tupla
lista = list(tupla)
lista.append(4)
tupla = tuple(lista)

##### SE RECORREN IGUAL QUE LAS LISTAS #####
```

Trabajando con diccionarios

Los diccionarios almacenan pares clave-valor y permiten acceso rápido a los datos.

```
diccionario = {"nombre": "Ana", "edad": 25}
# Acceso y actualización
```

```
diccionario.get("nombre")  # "Ana"
diccionario.get("ciudad", "No disponible")  # "No disponible"
diccionario.update({"ciudad": "Madrid"})

# Añadir y eliminar elementos
edad = diccionario.pop("edad")  # edad = 25
diccionario.popitem()  # Elimina ("ciudad", "Madrid") (el último para clave-v
alor agregado)
diccionario.clear()  # {}

diccionario = {"nombre": "Ana", "edad": 25}
# Obtener claves, valores y elementos
claves = diccionario.keys()  # dict_keys(['nombre', 'edad'])
valores = diccionario.values()  # dict_values(['Ana', 25])
elementos = diccionario.items()  # dict_items([('nombre', 'Ana'), ('edad', 2
5)])

# Copiar diccionarios
nuevo_diccionario = diccionario.copy()

##### RECORRIENDO DICCIONARIOS #####
datos = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

# Recorriendo claves
for clave in datos:
    print(clave)

# Recorriendo valores
for valor in datos.values():
    print(valor)

# Recorrer claves y valores
for clave, valor in datos.items():
    print(f"{clave}: {valor}")
```

Resumen de listas, tuplas y diccionarios

Estructura	Método de Recorrido
Lista	`for` y `while`
Tupla	`for` y `enumerate()`
Diccionario	`.keys()`, `.values()`, `.items()`
Lista de Diccionarios	`for` con acceso por clave

Tipo	Método	Descripción
Lista	`append(x)`	Agrega un elemento al final
	`extend(iterable)`	Agrega múltiples elementos
	`insert(i, x)`	Inserta un elemento en una posición
	`remove(x)`	Elimina la primera aparición de un elemento
	`pop(i)`	Elimina un elemento en una posición y lo devuelve
	`sort()`	Ordena la lista en orden ascendente
	`reverse()`	Invierte el orden de la lista
Tupla	`count(x)`	Cuenta las veces que aparece un elemento
	`index(x)`	Retorna la primera aparición de un elemento
Diccionario	`get(key, default)`	Obtiene el valor de una clave
	`update(dict)`	Agrega o actualiza claves y valores
	`pop(key)`	Elimina un elemento por clave y lo devuelve
	`keys()`	Devuelve todas las claves
	`values()`	Devuelve todos los valores
	`items()`	Devuelve lista de tuplas (clave, valor)

Compresiones de listas y funciones lambda

1️⃣ Comprensión de listas (List Comprehensions)

Proporcionan una forma concisa de crear listas aplicando una función a cada elemento de una secuencia o iterable.

Ejemplo: Crea una lista de los cuadrados del 0 al 9 usando comprensión de listas: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

La comprensión de listas anidadas pueden ser usadas para aplanar una matriz en una lista simple, mejorando la capacidad de manipulación de datos.

```
[<expresion> for <elemento> in <iterable>]
# aplicar condiciones
[<expresion> for <elemento> in <iterable> if <condicion>]
# aplanar una matriz
[<expresion> for <elemento> in <iterable> for <elemento> in <iterable>]
```

2️⃣ Funciones lambda

También conocidas como *funciones anónimas*, permiten la creación de pequeñas funciones únicas de forma concisa.

Ejemplo: Crea una función lambda que añada 10 a un número dado: `add_ten = lambda x: x + 10`.

El uso de funciones lambda con funciones como `map()`, `filter()` y `reduce()` permite aplicar una función a cada elemento de la lista eficientemente.

En general:

1. **List Comprehensions** son más eficientes y legibles cuando se trata de transformar y filtrar secuencias, ya que están optimizadas en Python y suelen ser más rápidas que `map()` y `filter()` con funciones lambda.
2. **Funciones Lambda con `map()` y `filter()`** pueden ser útiles en algunos casos, pero suelen ser menos legibles y, en la práctica, las list comprehensions suelen ser preferibles.

```
nums = [1, 2, 3, 4, 5]

# Elevar al cuadrado cada número en la lista
# Comparando cómo sería con comprensión y cómo con funciones lambda
cuadrados = [x**2 for x in nums] # Comprensión de listas (más rápido y cla
ro)
cuadrados = list(map(lambda x: x**2, nums)) # Funciones lambda (mismo r
esultado, menos legible)

# Filtrado de elemenos (por ejemplo, obtener los números pares)
```

```
pares = [x for x in nums if x % 2 == 0] # Comprensión de listas
pares = list(filter(lambda x: x % 2 == 0, nums)) # Funciones lambda
```

✅ **Usa List Comprehensions** cuando estés transformando o filtrando listas, ya que son más rápidas y más legibles.

✅ **Usa Lambdas** si necesitas una función anónima en `sorted()`, `reduce()`, o como argumento en funciones personalizadas.

❌ **Evita** `map()` **y** `filter()` **con lambdas** en la mayoría de los casos, porque las list comprehensions son más Pythonic y eficientes.

Si buscas eficiencia y claridad, la **list comprehension** es casi siempre la mejor opción. 🚀

Funciones

1️⃣ Definición de las funciones

Las funciones en Python permiten agrupar el código en bloques reusables.

Definir una función implica usar la palabra reservada `def` seguida del nombre de la función y los parámetros.

El bloque de código dentro de una función está sangrado por claridad y organización

2️⃣ Llamada a funciones (ejecución)

Las funciones pueden ser llamadas usando sus nombres seguidas de paréntesis.

Los **parámetros** son marcadores de posición para los valores pasados a las funciones, mientras que los **argumentos** son los valores reales proporcionados. Las funciones pueden tener cero o más parámetros.

Las funciones pueden retornar valores usando la palabra clave `return`. Proporciona una manera de pasar datos de vuelta a quien la llamó.

Comprender los parámetros de la función, argumentos y valores de retorno es esencial para crear programas modulares y mantenibles en Python.

Ámbito y duración de las variables

Scope → Ámbito

- Se refiere a la región del código donde una variable es accesible.
- Ejemplo: una variable definida dentro de una función tiene **ámbito local**, existen únicamente en este ámbito.

Lifetime → Duración

- Se refiere al tiempo durante el cual una variable existe en la memoria antes de ser eliminada.
- Ejemplo: una variable global en Python vive hasta que el programa termina, mientras que una variable dentro de una función se destruye cuando la función finaliza.

El conocimiento de ambas cosas es crucial para manejar variables de forma eficaz, garantizando un código eficiente y libre de fallos.

Buenas prácticas

- Limitar el uso de variables globales.
- Usar nombres descriptivos para las variables. Emplea nombres descriptivos para clarificar su ámbito y su propósito.
- Evitar modificaciones innecesarias a variables globales.
- Evitar el ocultamiento de variables (*variable shadowing*) previniendo comportamientos no deseados en el código.

```
x = 10  # Variable global

def my_function():
    x = 5  # Variable local que oculta a la global
    print(x)  # Imprime 5, no 10

my_function()
print(x)  # Imprime 10, la variable global no se modificó
```

Ámbitos de variables en Python

1️⃣ Ámbito Local

- Las variables están definidas dentro de una función
- Son sólo accesibles dentro de esa función

- Se crean cuando se llama a la función

```
def my_function():
    x = 10  # Variable local
    print(x)

my_function()
print(x)  # ❌ Error: x no está definida fuera de la función
```

2️⃣ Ámbito enclosing (Ámbito no local)

- En funciones anidadas, las variables de la función externa son accesibles desde la función interna
- Puede usarse la palabra clave `nonlocal` para modificarlas

```
def outer():
    y = 20  # Variable en el ámbito "Enclosing"

    def inner():
        print(y)  # Puede acceder a y porque está en el ámbito superior

    inner()

outer() # Imprime 20
```

```
def outer():
    y = 20

    def inner():
        nonlocal y
        y = 30  # Modifica la variable de outer()
        print(y)

    inner()
    print(y)  # Ahora y es 30

outer()
```

3️⃣ Ámbito Global

- Variables definidas fuera de una función
- Son accesibles desde cualquier parte del programa
- Existe durante toda la ejecución del programa
- Puede usarse la palabra clave `global` para modificarlas

```
x = 100  # Variable global

def my_function():
    print(x)  # Puede acceder a x sin problemas

my_function() # Imprime 100
```

```
x = 100

def modify_global():
    global x
    x = 200  # Ahora sí modificamos la variable global

modify_global()
print(x)  # 200
```

4️⃣ Ámbito Built-in (Nombres predefinidos)

- Incluye funciones y palabras clave predefinidas de Python, como `print()`, `len()`, `sum()`, etc.
- Accesibles en cualquier parte del programa

```
print(len([1, 2, 3]))  # Usa la función built-in len()
```

Manejo de errores y excepciones

Importancia de manejo de errores y excepciones

Tipos de excepciones comunes

- ZeroDivisionError: División por cero.

Garantiza la estabilidad del programa.

Previene que el programa se detenga abrupta e inesperadamente.

Mejora la experiencia del usuario.

Facilita la depuración y el mantenimiento del código.

Uso de bloques try-except

Envuelve el código en un bloque try para capturar excepciones.

Usa el bloque except para manejar excepciones específicas.

Usa un bloque else opcional para ejecutar código si no hay excepciones.

Usa un bloque finally opcional para ejecutar código de limpieza.

Creando excepciones personalizadas

Define clases de excepciones personalizadas.

Hereda de la clase Exception de Python.

Usa excepciones personalizadas para condiciones de error específicas.

Mejora la legibilidad y mantenibilidad del código.

Proporciona información detallada sobre el error.

- TypeError: Operaciones inválidas entre diferentes tipos de datos.
- ValueError: Valor inválido para una función u operación.
- FileNotFoundError: Archivo no encontrado.
- IndexError: Índice fuera de rango.

Mejores prácticas para el manejo de excepciones

Evita capturar excepciones genéricas.

Proporciona mensajes de error claros y significativos.

Registra o informa las excepciones para su análisis y depuración.

Maneja las excepciones en el nivel adecuado de la aplicación.

Usa tipos de excepciones específicos en lugar de genéricos.

Módulos, Paquetes y Librerías

1️⃣ Comprendiendo los módulos

Los módulos en Python son archivos individuales que contienen funciones, variables y clases relacionadas.

Crear un módulo implica guardar un archivo con código Python y la extensión `.py`.

La **importación** de módulos a scripts se realiza mediante la instrucción `import` seguida del nombre del módulo.

Funciones o variables específicas de un módulo se pueden importar utilizando la palabra reservada `from`. Ejemplo: `from pandas import DataFrame`.

El **cambio de nombre** de módulos o funciones se puede realizar utilizando la palabra reservada `as`. Ejemplo: `import numpy as np`.

Los módulos más populares en Python incluyen `math`, `random`, `datetime`, `os`, `sys`, `re`, `json`, `csv`, entre otros.

2️⃣ Comprendiendo los paquetes

Los paquetes en Python son colecciones de módulos relacionados organizados en directorios con un archivo `__init__.py`.

Los paquetes son directorios coteniendo múltiples módulos y deben tener un archivo `__init__.py` para ser reconocidos como paquetes.

La importación de módulos de un paquete se hace usando la notación de punto. Ejemplo: `import package_name.module_name`.

Los paquetes más populares en Python incluyen `numpy`, `pandas`, `matplotlib`, `scikit-learn`, `tensorflow`, `keras`, `requests`, `flask`, `django`, entre otros.

Concepto	Descripción	Ejemplo
Módulo	Un solo archivo `.py` con código reutilizable.	`math.py`, `random.py`
Paquete	Un conjunto de módulos organizados en una carpeta con un `__init__.py`.	`requests` tiene varios módulos internos.
Librería	Conjunto de módulos y paquetes que pueden ser estándar o externas.	`numpy`, `pandas`, `tensorflow`

```
# Librería estándar
import math

print(math.sqrt(16))  # 4.0
print(math.factorial(5))  # 120
print(math.pi)  # 3.141592653589793
```

```
# Librería externa (hay que importarla antes con pip install requests)
import requests

respuesta = requests.get("https://api.github.com")
print(respuesta.status_code)  # 200
print(respuesta.json())  # Devuelve la respuesta en formato JSON
```

Trabajando con ficheros

Operaciones de ficheros: Modos y métodos

Comprendiendo los modos de ficheros

Modo lectura ('**r**'): Abre un fichero para lectura, lanzando un FileNotFoundError si el fichero no existe.

Modo escritura ('**w**'): Abre un fichero para escritura, creando un nuevo fichero si no existe o truncando el fichero si ya existe.

Modo añadir ('**a**'): Abre un fichero para añadir contenido al final del fichero, creando un nuevo fichero si no existe.

Modo lectura y escritura ('**r+**'): Abre un fichero para lectura y escritura.

Abriendo y cerrando ficheros

Para trabajar con un fichero, usa la función `open()` especificando el modo.

Ciérralo con el método `close()` para liberar recursos.

Leer de un fichero

Métodos como `read()`, `readline()`, `readlines()` permiten leer contenido de un fichero como una cadena o línea por línea.

Escribir en un fichero

Usa el método `write()` o `writelines()` para escribir contenido en un fichero, ya sea una cadena o una lista de cadenas.

Mejor práctica: usar la declaración `with`

Se recomienda la declaración `with` para manejar un fichero.

Automáticamente maneja la apertura y cierre del fichero, incluso si ocurre una excepción, asegurándose que los recursos se liberen correctamente.

Beneficios

- Manejo automático de recursos
- Código limpio y eficiente
- Manejo simplificado de ficheros
- Mejora la legibilidad del código

Trabajando con diferentes tipos de ficheros en Python

Archivos CSV

CSV (*Comma Separated Values*) es un formato de archivo simple que se utiliza para almacenar datos tabulares.

Cada línea en un archivo CSV es una línea de la tabla con los campos separados por comas.

CSV guarda la información como texto plano, tratando todos los valores como cadenas.

Archivos JSON

JSON (*JavaScript Object Notation*) se usa para datos complejos y jerárquicos con una estructura de datos anidada.

JSON es un formato de texto que es fácil de leer y escribir para los humanos, pero puede volverse

Leyendo datos CSV

Usa el módulo `csv` para leer y escribir archivos CSV.

Los datos se pueden leer línea por línea o cargar en una lista de diccionarios.

Escribiendo datos JSON

Usa el módulo `json` para leer y escribir archivos JSON.

Convierte los objetos de Python en cadenas JSON y viceversa para el intercamio de datos.

Diferencias entre CSV y JSON

JSON es adecuado para datos complejos y jerárquicos, mientras que CSV es mejor para datos tabulares simples.

JSON es usado comúnmente en aplicaciones web y APIs para el

complejo con datos profundamente anidados.

Soporta varios tipos de datos, incluyendo cadenas, números, objetos, arreglos, valores booleanos y nulos.

intercambio de datos, mientras que CSV es más común en hojas de cálculo y bases de datos.

```
import csv

with open("datos.csv", newline='', encoding='utf-8') as archivo_csv:
    lector = csv.reader(archivo_csv)  # Crea un objeto lector
    for fila in lector:
        print(fila)  # Cada fila es una lista de valores
```

```
# CSV con encabezados
import csv

with open("datos.csv", newline='', encoding='utf-8') as archivo_csv:
    lector = csv.DictReader(archivo_csv)  # Lee el CSV como diccionario
    for fila in lector:
        print(fila)  # Cada fila es un diccionario con claves basadas en los enc
abezados
```

```
import json

with open("datos.json", encoding='utf-8') as archivo_json:
    datos = json.load(archivo_json)  # Carga el JSON en una variable tipo dic
cionario
    print(datos)
```

Tipo	Método	Descripción
CSV	`csv.reader(archivo)`	Lee CSV como lista de listas
	`csv.DictReader(archivo)`	Lee CSV como lista de diccionarios
JSON	`json.load(archivo_json)`	Carga JSON desde un archivo
	`json.loads(cadena_json)`	Carga JSON desde una cadena

Manejo de excepciones en operaciones de ficheros

- **Manejando FileNotFoundError:** Asegúrate de manejar excepciones potenciales como FileNotFoundError cuado trabajes con ficheros en Python. Algunos excepciones que pueden surgir:
  - `FileNotFoundError` → El archivo no existe.
  - `PermissionError` → No tienes permisos para acceder al archivo.
  - `IsADirectoryError` → Se intentó abrir un directorio como archivo.
  - `IOError` → Error general de entrada/salida (por ejemplo, disco lleno).
  - `EOFError` → Se alcanzó el final del archivo inesperadamente.
- **Usa bloques `try-except`:** Usa bloques `try` y `except` para gestionar los errores que puedan surgir durante las operaciones con archivos.
- **Importancia de manejo de excepciones**: El manejo de excepciones es crucial para un manejo sólido de archivos que garantice una ejecución fluida incluso en presencia de errores.
- **Mejora la fiabilidad y estabilidad:** Al implementar un manejo adecuado de excepciones, puede mejorar la confiabilidad y la estabilidad de sus operaciones con archivos.

```
# uso seguro with open()
try:
    with open("datos.txt", "r") as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print("Error: No se encontró el archivo.")
except Exception as e:
    print(f"Error inesperado: {e}")
```

```
# manejo de lectura y escritura con excepciones
try:
    with open("output.txt", "w") as archivo:
        archivo.write("Hola, mundo!")
        print("Archivo escrito correctamente.")
```

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

Programación Orientada a Objetos (POO)

La **Programación Orientada a Objetos (POO)** en Python es un paradigma que organiza el código en **clases** y **objetos**. Permite modelar entidades del mundo real con atributos y comportamientos.
Una
clase es un molde para crear **objetos**.

```
class Persona:
    def __init__(self, nombre, edad): # Constructor (Método mágico)
        self.nombre = nombre  # Atributo de instancia
        self.edad = edad

    def saludar(self):  # Método
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

# Creación de un objeto
persona1 = Persona("Carlos", 30)
persona1.saludar()
```

📌 **Explicación:**

- `__init__` es el **constructor** que inicializa los atributos.
- `self` representa la instancia actual de la clase.
- Se accede a los atributos y métodos con `objeto.atributo` o `objeto.metodo()`.

Principios de la POO

- **Encapsulamiento (Modificadores de acceso):** El encapsulamiento controla el acceso a los atributos y métodos.
- **Herencia:** Permite que una clase hija herede de una clase padre.
- **Polimorfismo:** Permite usar el mismo método en diferentes clases.

```
# Encapsulamiento
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo  # Atributo privado

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def __mostrar_saldo(self):  # Método privado
        print(f"Saldo: {self.__saldo}")

# Crear objeto
cuenta = CuentaBancaria("Ana", 1000)
cuenta.depositar(500)
# cuenta.__mostrar_saldo()  # Error: método privado
```

📌 **Niveles de acceso:**

- **Público (`self.atributo`)** → Accesible desde fuera.
- **Privado (`self.__atributo`)** → No accesible directamente.

```
# Herencia
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass  # Método abstracto

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau!")

class Gato(Animal):
    def hacer_sonido(self):
        print("Miau!")
```

```
# Uso de herencia
perro = Perro("Rex")
gato = Gato("Whiskers")

perro.hacer_sonido()
gato.hacer_sonido()
```

📌 **Notas:**

- `class Hija(Padre)` → La clase hija hereda de la clase padre.
- Se puede sobrescribir (`override`) métodos.

```
# Polimorfismo
def hacer_sonido(animal):
    animal.hacer_sonido()

hacer_sonido(Perro("Max"))
hacer_sonido(Gato("Mimi"))
```

Aquí, `hacer_sonido(animal)` funciona con diferentes tipos de objetos.