**MODULO I**

**DATA SCIENCE - PROGRAMACIÓN EN PYTHON**

[Introducción](#_dsdbv945at5v)

[Documentación y Herramientas](#_bo842qwcyqm9)

[Librerías de python](#_sasz23ydbzcm)

[PARTE I - Conceptos de programación en PyTHoN](#_ytfq6ty6lb6v)

[Listas](#_b7vm0uj20fs1)

[Acceso por posición:](#_wcmj3wegkxuw)

[Ejemplo:](#_8ob4aawpryml)

[Modificación de elementos por posición:](#_jyz37rx279j)

[Copiar una lista:](#_8fxp41sk6tep)

[Ejemplo de copia con alias:](#_2kugl0wa9f1s)

[Copia independiente (copia profunda):](#_4p42fgumgaxa)

[Resumen:](#_9mez3jjznxhu)

[Tuplas](#_aa46xe2qbvf5)

[Características principales:](#_kwtx8bandj8l)

[Ejemplo de una tupla:](#_btr0ns1unrmc)

[Trabajo con elementos mutables dentro de una tupla:](#_vo8tnvmsw863)

[Copias y alias:](#_kofayqf2fu4s)

[Modificación de tuplas mediante conversión a lista:](#_89el3peeyzvq)

[Resumen:](#_fm7jqzsfzahc)

[Diccionarios](#_60zg4qklm0v2)

[Sintaxis básica:](#_gif3r2st1l5)

[Acceso a valores:](#_ooa0fhkmkxs6)

[Modificar o añadir elementos:](#_yoqq88ajhrwv)

[Eliminar elementos:](#_9cookl8ris2l)

[Características principales:](#_ensg3madycaf)

[Usos comunes:](#_bpz1eusqn37n)

[Conjuntos](#_ufaygl41iapc)

[Ejemplo:](#_js1rw9ptnrau)

[Características importantes:](#_xpbyvltgu39h)

[Error común: "unhashable type"](#_46dwjlt740a1)

[Tipos de datos hashables (inmutables):](#_fatdf9rvrvfm)

[Tipos de datos no hashables (mutables):](#_jfovh6mzcoun)

[Solución al error:](#_4nwy60fknz1z)

[Conclusión:](#_9eajd15l3lhs)

[Condiciones](#_jbofvg3n9yz4)

[1. if y else:](#_tqgjydczgkwa)

[2. elif:](#_7onrcbtar7ms)

[Ejemplo:](#_odsyqkxqa6vq)

[Incrementos y operadores en bucles:](#_39z4v26vwcep)

[Notas adicionales:](#_vtf3atkod91n)

[Patrones de diseño](#_xlhrqblc5qw4)

[Ejemplo:](#_ofgklew14dtw)

[Explicación:](#_6w72lsqnjxyf)

[Importancia:](#_8lwqb3hra7ne)

[Consejos:](#_wzcp1gsnhmok)

[Argumentos](#_qrgvj3kv0gbj)

[Ejemplos de cómo hacer lo mismo de dos formas distintas:](#_tbc98mbdlf2i)

[Detalles importantes:](#_1zvnrsmqpb6)

[FUNCIONES](#_rsd4zqy3s6dl)

[Funciones I](#_lad7iavs6y9d)

[Funciones II](#_wyju173hff0h)

[Funciones III (generadoras)](#_bmex41pdkxag)

[Características principales:](#_a5bu76o93y2n)

[Ejemplo de una función generadora:](#_8ck1nrbecbhr)

[Ventajas de las funciones generadoras:](#_gbgdccu4wsg1)

[Resumen:](#_dnjoa2t0foqq)

[Manejo de errores](#_b769b820fr5e)

[Ejemplo básico con errores predefinidos:](#_jtdj1to9jshz)

[Crear errores personalizados:](#_um8bcb58zb0f)

[Estructura básica de manejo de errores:](#_wfs32fthdws1)

[Uso de pass:](#_sn1izgqbs33a)

[Importancia del manejo de errores:](#_po4in1tt8ph2)

[Archivos](#_cy0e82c7o7i1)

[Ejemplo de lectura de un archivo CSV:](#_sscn7za1w6u)

[Ejemplo de escritura en un archivo CSV:](#_dancw57tc10a)

[Otras librerías útiles:](#_vcu54xd8gyet)

[Ejemplo de uso de os:](#_bx7ig1qi63km)

[Apertura y cierre de archivos](#_6ysea7i58i2e)

[Gestionar los permisos de los archivos](#_pgep0za2w0ag)

[1. Verificación de permisos](#_noc7yhqk7uww)

[2. Cambiar permisos de archivos](#_ih16h3kcv3qv)

[3. Sistemas de archivos virtuales](#_75whwzqahaf7)

[4. Autenticación y autorización (para aplicaciones web)](#_b0v7tid4aw8j)

[Sintaxis de lambda:](#_ekxdguc2y97k)

[Ejemplo básico de lambda:](#_ed036nbm2b7g)

[Ejemplo de uso:](#_rla5wrte1dqe)

[¿Por qué usar lambda?](#_j0t8n9ild6xs)

[Ejemplos prácticos:](#_hryk9i1v21zd)

[1. Usando lambda con map():](#_eblgtverndl0)

[2. Usando lambda con filter():](#_e7gc1ebypal)

[3. Usando lambda con sorted():](#_g05c3yff55cy)

[¿Cuándo evitar lambda?](#_9kid3543xpvt)

[Resumen:](#_fpvs8hrl9gph)

[map()](#_yh7qkh7evx52)

[Sintaxis:](#_75qzqfr66kqw)

[Ejemplo de map():](#_nvuenjoczabq)

[Ejemplo usando una función lambda:](#_167csyrwle7j)

[filter()](#_kvi189y6i10k)

[Sintaxis:](#_dveopmofavdi)

[Ejemplo de filter():](#_448qh36ab27m)

[Ejemplo usando una función lambda:](#_6r6t21g2d8uc)

[zip()](#_nobp3bd3sqti)

[Ejemplo básico de zip():](#_9te8mjgua26s)

[enumerate()](#_lh04scd8b0we)

[Ejemplo básico de enumerate():](#_3zot8gmayu1)

[Combinando zip() y enumerate():](#_b7izema2zwyo)

[Ejemplo combinando zip() y enumerate():](#_j0fh8o38qxpj)

[Resumen:](#_3pw5r1enaky3)

[Funciones decoradoras](#_vco16nat1nnr)

[¿Cómo funciona un decorador?](#_dic46kta5912)

[Ejemplo básico de un decorador:](#_am7tzi0xt2q)

[Explicación:](#_za0n3e7wapuw)

[¿Cuándo son útiles los decoradores?](#_ujg7p76sdjaf)

[Resumen:(sintaxis standard - WRAPPER)](#_rovy2xswsyiz)

[\*args y \*\*kargs](#_73t9u28c4f94)

[\*args](#_6wfklxuka1j)

[\*\*kwargs](#_813on4lv3t2n)

[Relación con \* y \*\*](#_3lg9qzblmfo6)

[PARTE II - Programación orientada a objetos (POO)](#_92v8b72l8fg2)

[Conceptos clave en POO](#_uj1gklllpviq)

[1. Instrucción (NO se guarda en memoria)](#_k1ib8atzi719)

[2. Clase (se guarda en memoria)](#_64edsv6xdgov)

[3. Instancia de la clase (se guarda en memoria)](#_e3yuwlkf5hvi)

[4. Atributo (se guarda en memoria)](#_1i06epovm5ht)

[5. ¿Qué hace dir()? ¿Es un método?](#_nwr7a0m1ln21)

[6. Método](#_geth3dx2iutb)

[Relación de todos estos conceptos con objetos en la POO](#_ha4bjdpszpe9)

[El papel de self](#_i7h76nnr3qmn)

[Diferencias entre método de clase y método de instancia:](#_tf70j5m074wl)

[Resumen:](#_zgsycje77oad)

[Métodos especiales o mágicos](#_lt0f08kw6cga)

[1. \_\_init\_\_ (Inicializador)](#_2zh39rvhsmyg)

[2. \_\_str\_\_ (Representación en cadena)](#_wzhggnq1h97)

[3. \_\_repr\_\_ (Representación oficial)](#_28x378cjcuzj)

[4. \_\_len\_\_ (Longitud)](#_roxedda9tlqg)

[5. \_\_eq\_\_ (Comparación de igualdad)](#_2d8ets79c4qo)

[6. \_\_add\_\_ (Suma de objetos)](#_heqb4khgyonf)

[7. \_\_call\_\_ (Convertir el objeto en una función)](#_arky5kaxk2y9)

[8. \_\_del\_\_ (Destructor)](#_2v2s2jeeuhf5)

[Resumen:](#_7q69xkz70w81)

[El constructor](#_5y7skst9452v)

[Concepto clave:](#_qjgbduo3bt8q)

[Ejemplo de constructor:](#_c9u6sel20os1)

[Función del constructor:](#_429s6pjozn49)

[¿Por qué es importante?](#_9au866saypvq)

[El destructor](#_b494lrulp65u)

[Concepto clave:](#_y0knp1sybae4)

[Ejemplo de destructor:](#_jqy6cutmismx)

[Funcionamiento:](#_i62yxhmv3ysp)

[¿Cuándo se llama al destructor?](#_7wlrqwgd0ja9)

[¿Cuándo usar el destructor?](#_guqd4oofrki5)

[Resumen:](#_izs9zszi439b)

[Herencia y polimorfismo](#_3pn9n3n76yos)

[Herencia](#_madrigfashus)

[super() en herencia](#_puosa7q6g5f5)

[raise en herencia](#_f8mu5st2xp95)

[Polimorfismo](#_wbg9ovjwrzc2)

[raise en polimorfismo](#_vtous9j2is52)

[try...except y raise en herencia y polimorfismo](#_c1e1cxrs2p6r)

[Resumen final](#_gxig50vssiqz)

# Introducción

Consultas relacionado con dudas de Lanbide:

erandio@lanbide.eus

ChatGPT tiene un delay de 18 meses. 6 para entrenar , 6 verificar y 6 para liberar

Stackoverflow es la plataforma más actualizada por ello para la resolución de muchos problemas

# Documentación y Herramientas

Documentación del curso: [bit.ly/laner\_python](http://bit.ly/laner_python)

Pizarra: [bit.ly/laner\_meet](http://bit.ly/laner_meet)

Herramientas si se usa windows

Instalar Anaconda: <https://www.anaconda.com/download/success> (instala librerías por defecto y las más habituales pero existen más y es bueno cotillearlas)

* Spider (específico ciencia de datos): Entorno de desarrollo específico para ciencia de datos también vale VSC
* Jupiter notebook
* [Google collab](https://colab.research.google.com/): Entorno virtual de Google con servidores más potentes
* VSC
* Python Tutor (compilador): <https://pythontutor.com/python-compiler.html#mode=edit>
* Documentación de strings methods: <https://www.w3schools.com/python/python_ref_string.asp>
* Herramientas de IA:
  + ChatGPT\_ 18 meses de delay aproximadamente, es posible que de código antiguo ya que son 6 meses de entrenamiento, 6 meses de revisión revisión y 6 meses de despliegue
  + Gemini: es de google y esta guay para que te saque scripts de google
  + Youtube: Xavier Mitjana
  + [harpa.ai](http://harpa.ai): Resumen de video y subtítulos, y puedes chatear con el contenido del video.
  + NotebookLM: Producto de google
  + [Whisper](https://openai.com/index/whisper/): OpenAI, puede instalarse como librería ya que es un producto mantenido por OpenAI pero es de software libre
  + Copilot de GitHUB es el mejor para código
* PYQT + CoPilot

# Librerías de python

* Oficial Python: <https://docs.python.org/3/library/index.html>
* Librería Matplotlib: <https://matplotlib.org/>
* Pandas: <https://pandas.pydata.org/>
* Laravel: <https://laravel.com/>
* PyQT

Para linux con instalar python que ya viene por defecto y VSC es suficiente, aunque es importante que hay que añadir python .exe al path, de esta manera puedes encontrar el ejecutable al path , si es en windows habría que “saltarse los 250 caracteres”

En terminal al poner python en terminal te lleva al path y el prompt es >>> al final, para llegar aquí es mejor tener un poco más de nivel, es mejor ir al entorno de desarrollo de Anaconda que te facilita la vida.

Para empezar hay que crear el proyecto en Spyder

Ruta windows : C:\Users\rportatil112\PYTHON\_BÁSICO

sincronizado con el drive local / MEGA

Para anular una comilla dentro de un string se pone \

Herramientas IA:

* [perplexity](https://www.perplexity.ai/) - te saca fuentes y enlaces
* [gemini](https://gemini.google.com/app?hl=es-ES) - como perplexity te saca las fuentes

# PARTE I - Conceptos de programación en PyTHoN

# Listas

*Las* ***listas*** *en Python son una estructura de datos* ***ordenada y mutable****. Esto significa que los elementos dentro de una lista tienen un orden definido (pueden ser accesibles a través de su índice) y se pueden modificar después de ser creadas.*

#### ***Acceso por posición:***

* ***Índices positivos****: Se utilizan para acceder o modificar los elementos de la lista desde el principio (de izquierda a derecha). El primer elemento tiene índice 0.*
* ***Índices negativos****: Se utilizan para acceder o modificar los elementos desde el final (de derecha a izquierda). El último elemento tiene índice -1.*

#### ***Ejemplo:***

*python*

*lista = [1, 'Bilbao', 'Ana', 'Orestes', 2.5]*

*En esta lista:*

* *El índice 0 corresponde a 1.*
* *El índice 1 corresponde a 'Bilbao'.*
* *El índice -1 corresponde a 2.5 (último elemento de la lista).*

### ***Modificación de elementos por posición:***

*python*

*# Modificar por la izquierda (índice positivo)*

*lista[2] = 'Sevilla' # Cambia 'Ana' por 'Sevilla'*

*# Modificar por la derecha (índice negativo)*

*lista[-2] = 'Perro' # Cambia 'Orestes' por 'Perro'*

*Después de estas modificaciones, la lista será:*

*python*

*[1, 'Bilbao', 'Sevilla', 'Perro', 2.5]*

### ***Copiar una lista:***

*Cuando copias una lista en Python, si simplemente asignas una nueva variable, ambas variables seguirán apuntando a la* ***misma lista en memoria****. Esto significa que los cambios en una lista se reflejarán en la otra. Para crear una* ***copia independiente*** *de una lista, puedes utilizar el módulo copy.*

#### ***Ejemplo de copia con alias:***

*python*

*import copy as cp*

*# Crear una lista y asignarla a otra variable*

*lista = [1, 'Bilbao', 'Ana', 'Orestes', 2.5]*

*copia\_lista = lista # Esto es un alias, no una copia independiente*

*En este caso, tanto lista como copia\_lista apuntan al* ***mismo objeto en memoria****. Si modificas una, la otra también se verá afectada.*

#### ***Copia independiente (copia profunda):***

*python*

*copia\_lista = cp.deepcopy(lista) # Crea una copia independiente*

*Con deepcopy(), los cambios realizados en copia\_lista no afectarán a la lista original lista.*

### ***Resumen:***

* *Las* ***listas*** *son ordenadas y mutables. Se pueden modificar directamente accediendo a sus elementos por su índice.*
* *Los* ***índices positivos*** *se utilizan para acceder a los elementos desde el principio, mientras que los* ***índices negativos*** *permiten acceder a los elementos desde el final.*
* *Para hacer una* ***copia independiente*** *de una lista y evitar cambios reflejados en ambas, usa copy.deepcopy().*

Visualización: [Modificar valores en lista](https://pythontutor.com/render.html#code=import%20copy%20as%20cp%0A%0Alista%20%3D%20%5B%201,%20'Bilbao',%20'Ana',%20'Orestes',%202.5%5D%0Acopia_lista%20%3D%20lista%0A%0Alista%5B2%5D%20%3D%20'Sevilla'%0Alista%20%5B-2%5D%20%3D%20'Perro'&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

# Tuplas

Una **tupla** es una estructura de datos en Python similar a una lista, pero con la diferencia clave de que es **inmutable**. Esto significa que, una vez creada, no se puede modificar su tamaño ni su contenido de manera directa. Se distingue por usar **paréntesis** en lugar de corchetes como las listas.

#### **Características principales:**

1. **Inmutabilidad**: No puedes cambiar ni agregar elementos a una tupla después de su creación. Sin embargo, si una tupla contiene un **objeto mutable** (como una lista), ese objeto en particular **sí** puede modificarse.
2. **Uso de paréntesis**: Las tuplas se definen con paréntesis () en lugar de corchetes [].
3. **Mejor rendimiento**: Al ser inmutables, las tuplas tienen un rendimiento ligeramente mejor que las listas, especialmente en operaciones de acceso y almacenamiento.

#### **Ejemplo de una tupla:**

python

tupla = (1, 2, 'Python', True, 3.14)

En este ejemplo, la tupla contiene varios tipos de datos (números, cadenas, booleanos y flotantes). No puedes cambiar los elementos directamente:

python

tupla[0] = 10 # Esto generará un error porque las tuplas son inmutables

#### **Trabajo con elementos mutables dentro de una tupla:**

Si una tupla contiene elementos mutables, como una **lista**, esos elementos sí pueden ser modificados, aunque no puedes cambiar el propio contenedor (la tupla):

python

lista = ['elemento1', 'elemento2']

tupla = (1, 2, 'Python', lista)

# Modificar la lista dentro de la tupla

tupla[3].append('elemento3')

print(tupla) # Salida: (1, 2, 'Python', ['elemento1', 'elemento2', 'elemento3'])

En este caso, hemos modificado la lista dentro de la tupla. **La tupla en sí sigue siendo inmutable**, pero los elementos dentro de la lista, que es mutable, sí pueden cambiar.

#### **Copias y alias:**

Cuando asignas una **lista** o un objeto mutable a otra variable, estás creando un **alias**, es decir, ambas variables apuntan al mismo objeto en la memoria. Cualquier cambio en la lista afectará a todas las referencias a esa lista:

python

alias = lista

alias.append('elemento4')

print(lista) # Salida: ['elemento1', 'elemento2', 'elemento3', 'elemento4']

Esto demuestra que las listas son mutables, y los cambios realizados a través de alias se reflejan en la lista original.

#### **Modificación de tuplas mediante conversión a lista:**

Aunque no puedes modificar directamente una tupla, puedes convertirla temporalmente en una lista para modificar sus valores y luego volver a convertirla en tupla:

python

tupla = (1, 2, 'Python')

lista = list(tupla) # Convertir tupla a lista

lista.append('Nuevo elemento')

tupla = tuple(lista) # Convertir de nuevo a tupla

print(tupla) # Salida: (1, 2, 'Python', 'Nuevo elemento')

Este método es útil si necesitas cambiar los elementos de una tupla de manera indirecta.

### **Resumen:**

* Las **tuplas** son inmutables, lo que significa que no puedes modificar sus elementos directamente.
* Los elementos mutables dentro de una tupla, como listas, **sí** se pueden modificar.
* Puedes usar la conversión entre tuplas y listas si necesitas realizar cambios temporales en los elementos de una tupla.

Visualización: [Tupla y anidación con lista](https://pythontutor.com/render.html#code=lista%20%3D%20%5B'elemento%201','elemento2'%5D%0Aalias%20%3D%20lista%0A%0Atupla%20%3D%20%281,%202,%20'Python',%20True,%203.14,lista%29%0A%0Alista%20%3D'Reasignada!'%0A%0Atupla%5B5%5D.append%28'elemento%203'%29%0Aalias.append%28'elemento%204'%29&cumulative=false&curInstr=5&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

# Diccionarios

Un **diccionario** es una estructura de datos en Python que almacena **pares de clave-valor**. A diferencia de las listas o tuplas, los diccionarios son **no ordenados** (en versiones anteriores a Python 3.7), lo que significa que no tienen un orden específico para los elementos. Cada elemento en un diccionario está formado por dos partes: una **clave** y su correspondiente **valor**.

#### **Sintaxis básica:**

python

mi\_diccionario = {

"nombre": "Ana",

"edad": 30,

"ciudad": "Bilbao"

}

* **Claves**: Son únicas y suelen ser de tipos inmutables como cadenas (str), números (int), o tuplas.
* **Valores**: Pueden ser de cualquier tipo, incluyendo otros diccionarios, listas, números, etc.

#### **Acceso a valores:**

Puedes acceder a un valor utilizando su **clave**:

python

print(mi\_diccionario["nombre"]) # Salida: Ana

#### **Modificar o añadir elementos:**

python

mi\_diccionario["edad"] = 31 # Modifica el valor de la clave 'edad'

mi\_diccionario["profesión"] = "Ingeniera" # Añade una nueva clave-valor

#### **Eliminar elementos:**

python

del mi\_diccionario["ciudad"] # Elimina la clave 'ciudad' y su valor

### **Características principales:**

* **Claves únicas**: Las claves en un diccionario deben ser únicas. Si intentas asignar un valor a una clave que ya existe, simplemente sobrescribirás el valor anterior.
* **No ordenados** (antes de Python 3.7): El orden de los elementos no está garantizado. A partir de Python 3.7, los diccionarios mantienen el orden de inserción, aunque esto no siempre debe ser asumido para versiones más antiguas.

### **Usos comunes:**

Los diccionarios son muy útiles para almacenar datos relacionados, como por ejemplo:

* Información de contacto: nombre, teléfono, dirección.
* Resultados de encuestas con preguntas como claves y respuestas como valores.
* Configuraciones de aplicaciones, donde cada opción tiene una clave y su valor correspondiente.

# Conjuntos

Un **conjunto** en Python (set) es una colección **no ordenada** de elementos **únicos**, lo que significa que **no permite duplicados**. Es similar a un diccionario, pero no tiene la estructura de clave-valor. Por esta razón, los conjuntos son útiles cuando se quiere eliminar elementos duplicados de una colección de datos.

#### **Ejemplo:**

python

# Crear un conjunto

mi\_conjunto = {1, 2, 3, 4, 4, 5}

print(mi\_conjunto) # Salida: {1, 2, 3, 4, 5} (los duplicados se eliminan automáticamente)

### **Características importantes:**

1. **No duplicados**: Cualquier elemento duplicado que agregues a un conjunto se eliminará automáticamente, lo que lo hace útil para **limpiar listas**.
2. **Desordenados**: A diferencia de las listas o tuplas, los elementos de un conjunto **no tienen un orden definido**, por lo que no se puede acceder a ellos mediante índices.
3. **No tipos mutables**: Los elementos de un conjunto deben ser **inmutables** (es decir, no pueden cambiar). Por eso, puedes añadir números, cadenas, o tuplas, pero **no listas o diccionarios**, ya que son mutables.

#### **Error común: "unhashable type"**

El error **"unhashable type"** ocurre cuando intentas agregar un tipo de dato mutable (como una lista o un diccionario) a un conjunto. Esto se debe a que los conjuntos en Python están implementados usando una **tabla hash**. Solo los objetos que son inmutables (y por lo tanto, hashables) pueden ser elementos de un conjunto.

Por ejemplo:

python

mi\_conjunto = {1, 2, [3, 4]} # Esto generará un error porque una lista es mutable y no puede ser hashable

### **Tipos de datos hashables (inmutables):**

* **Cadenas** (str)
* **Números** (int, float)
* **Tuplas** (solo si todos sus elementos son inmutables)

### **Tipos de datos no hashables (mutables):**

* **Listas** (list)
* **Diccionarios** (dict)

#### **Solución al error:**

Si necesitas agregar una lista a un conjunto, puedes convertirla en una **tupla** primero, ya que las tuplas son inmutables y, por lo tanto, hashables:

python

mi\_conjunto = {1, 2, tuple([3, 4])} # Convertimos la lista en una tupla

print(mi\_conjunto) # Salida: {(3, 4), 1, 2}

### **Conclusión:**

Los conjuntos son una estructura útil cuando se necesita manejar colecciones de datos únicos y eliminar duplicados automáticamente. Ten en cuenta el error "unhashable type", que te recordará no incluir tipos mutables como listas o diccionarios en un conjunto.

# Condiciones

El uso de estructuras de control como if, elif y else te permite tomar decisiones en tu código según ciertas condiciones.

#### **1. if y else:**

El bloque **if** evalúa una condición. Si es verdadera, ejecuta el código dentro de ese bloque. **No es necesario** que siempre esté acompañado de un else, ya que else solo se usa cuando quieres que algo ocurra si la condición del if no se cumple.

#### **2. elif:**

La sentencia **elif** (abreviatura de "else if") es un punto intermedio entre if y else. Se usa para evaluar **condiciones adicionales** cuando la primera condición del if no se cumple. Si ninguna condición if o elif es verdadera, entonces el bloque **else** (si está presente) se ejecuta como salida final.

#### **Ejemplo:**

python

numero\_secreto = 7

adivinanza = None

while adivinanza != numero\_secreto:

adivinanza = int(input("¿Qué número es?: "))

if adivinanza < numero\_secreto:

print("Muy bajo")

elif adivinanza > numero\_secreto:

print("Muy alto")

print("¡Zorionak! Has acertado.")

* En este ejemplo, el programa pide una **adivinanza** en un bucle while. Si el número es menor que el número secreto, imprime "Muy bajo". Si es mayor, imprime "Muy alto". Cuando aciertas, el bucle se detiene y te felicita.

### **Incrementos y operadores en bucles:**

En algunos lenguajes, verás incrementos como **i++** o **i+=1**. En Python:

* **i+=1** incrementa el valor de i en 1 (Python no usa i++ como en otros lenguajes como C o Java).
* Usar correctamente los incrementos es crucial en bucles for o while para evitar que el programa quede en un bucle infinito o se detenga abruptamente.

### **Notas adicionales:**

* **i**: Representa el **índice** en un bucle (comúnmente usado en for).
* **j**: Puede representar una **segunda iteración** (útil en bucles anidados).
* Evita **detener un bucle** sin razones claras, ya que podría generar errores en la ejecución.

# Patrones de diseño

*El bloque* ***try-except*** *en Python es un mecanismo utilizado para* ***manejar errores*** *de manera controlada. Funciona de manera similar a las condiciones y bucles en cuanto a su estructura, permitiendo que el código reaccione ante errores específicos sin detener la ejecución del programa por completo.*

#### ***Ejemplo:***

*python*

*try:*

*num = int(input("Introduce un número: "))*

*resultado = 10 / num*

*print("Resultado: " + str(resultado))*

*# Excepciones específicas*

*except ZeroDivisionError as e:*

*print("Error: No se puede dividir por cero.")*

*except ValueError as e:*

*print("Error: Debes introducir un número entero.")*

*# Bloque 'finally' se ejecuta siempre*

*finally:*

*print("Este bloque se ejecuta pase lo que pase.")*

#### ***Explicación:***

* ***try****: Aquí se coloca el código que puede causar errores. Si todo va bien, se ejecuta normalmente.*
* ***except****: Se utiliza para capturar errores específicos. En este ejemplo:*
  + ***ZeroDivisionError****: Se lanza si se intenta dividir por 0.*
  + ***ValueError****: Ocurre si el usuario introduce algo que no es un número entero.*
* ***finally****: Este bloque se ejecuta siempre, ocurra o no un error. Es útil para realizar tareas de limpieza como cerrar archivos, liberar recursos, etc.*

#### ***Importancia:***

*El manejo adecuado de excepciones permite que el programa siga funcionando aunque ocurra un error, brindando retroalimentación útil a la persona usuaria en lugar de simplemente detener el programa abruptamente.*

### ***Consejos:***

* *Es buena práctica manejar solo los* ***errores que esperas*** *que puedan ocurrir. Evita capturar excepciones generales como Exception, ya que puede ocultar errores que deberían ser tratados de forma diferente.*
* *Utiliza finally para asegurarte de que ciertas acciones se realicen, independientemente de si ocurre un error o no.*

Visualización: [Ejemplo para ZeroDivisionError](https://pythontutor.com/render.html#code=try%3A%0A%20%20%20%20num%20%3D%20int%28input%28%22Introduce%20un%20n%C3%BAmero%3A%20%22%29%29%0A%20%20%20%20%0A%20%20%20%20resultado%20%3D%2010%20/%20num%0A%20%20%20%20print%28%22Resultado%3A%20%22%20%2B%20str%28resultado%29%29%0A%0Aexcept%20ZeroDivisionError%20as%20e%3A%20%20%23%20Manejar%20la%20excepci%C3%B3n%20de%20divisi%C3%B3n%20por%20cero%0A%20%20%20%20print%28%22Error%3A%20No%20se%20puede%20dividir%20por%20cero.%22%29%0Aexcept%20ValueError%20as%20e%3A%20%20%23%20Manejar%20la%20excepci%C3%B3n%20si%20la%20entrada%20no%20es%20un%20n%C3%BAmero%0A%20%20%20%20print%28%22Error%3A%20Debes%20introducir%20un%20n%C3%BAmero%20entero.%22%29%0A%20%20%20%20%0Afinally%3A%0A%20%20%20%20print%28%22Me%20ejecuto%20s%C3%AD%20o%20s%C3%AD.%22%29%0A&cumulative=false&curInstr=7&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=311&rawInputLstJSON=%5B%220%22%5D&textReferences=false)

# Argumentos

Los **argumentos** son los datos que pasamos dentro del paréntesis de una función cuando la llamamos. Son esenciales para que las funciones puedan trabajar con diferentes valores o datos de entrada.

#### **Ejemplos de cómo hacer lo mismo de dos formas distintas:**

**Forma 1: Usando un bucle for y una función separada**

python

lista\_numeros = [1, 2, 3, 4, 5]

lista\_paridad = []

def es\_par(numero):

return numero % 2 == 0

for numero in lista\_numeros:

lista\_paridad.append(es\_par(numero))

print(lista\_paridad)

Aquí, la función es\_par() devuelve True si un número es par y False si no lo es. Luego usamos un bucle for para recorrer lista\_numeros y añadimos el resultado (si es par o impar) a lista\_paridad.

**Forma 2: Todo dentro de una función**

python

def es\_par\_lista(lista\_num):

lista\_paridad = []

for numero in lista\_num:

lista\_paridad.append(numero % 2 == 0)

return lista\_paridad

lista\_numeros = [1, 2, 3, 4, 5]

lista\_paridad = es\_par\_lista(lista\_numeros)

print(lista\_paridad)

En este ejemplo, la función es\_par\_lista() recibe una lista, verifica si cada número es par, y devuelve una lista con True o False según la paridad de cada número.

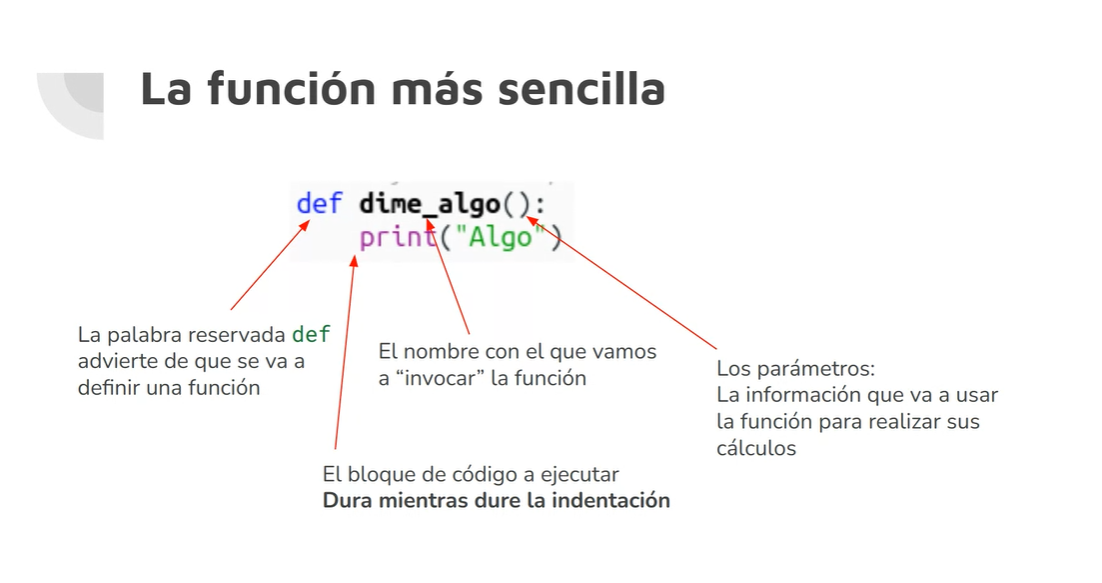
### **Detalles importantes:**

1. **El print() no devuelve resultados**:
   * La función print() solo muestra información en la consola, pero **no devuelve ningún valor**. Por eso, si necesitas usar el resultado de una operación o función más adelante, deberías usar return dentro de la función, no print().
2. **Comprobaciones para evitar problemas**:
   * Al ejecutar código, es importante evitar dejar procesos sin control (por ejemplo, en un bucle infinito), ya que esto puede colapsar el sistema o el servidor. **Siempre es recomendable hacer validaciones** y comprobaciones en el código para evitar que se quede bloqueado.
3. **Cuidado con los booleanos**:
   * Los valores True y False en Python se comportan también como **enteros** (True se interpreta como 1 y False como 0). Por lo tanto, ten cuidado al usar booleanos en operaciones matemáticas, ya que podrían comportarse como números sin que te des cuenta.

# FUNCIONES

## Funciones I

Siempre detrás del nombre de la función siempre va a haber unos paréntesis que pueden estar vacíos o no, el contenido serán los argumentos que pueden ser indefinidos el contenido de la función se considera o se trata como una TUPLA.



La **indentación** es una tabulación de 4 espacios, a diferencia de otros lenguajes, en python el bloque de código a ejecutar no va entre corchetes, por eso es importante la indentación en python porque si no no entiende el código.

## Funciones II

En Python, los argumentos de las funciones se pueden pasar de dos maneras: por **posición** o por **clave-valor** (llamados también **argumentos nombrados** o **keyword arguments**).

1. **Argumentos por posición**:
   * Cuando llamas a una función, los argumentos se pasan en el **orden en que están definidos** en la declaración de la función. Python espera recibir los valores en ese orden.

Ejemplo de código:  
  
def saludo(nombre, edad):

print(f"Hola, mi nombre es {nombre} y tengo {edad} años.")

saludo("Ana", 25) # Los argumentos se pasan por posición

1. **Argumentos por clave-valor (keyword arguments)**:
   * Puedes pasar los valores a una función usando **nombres específicos de los parámetros** (clave), en lugar de hacerlo por la posición en que aparecen en la definición.
   * Esto es útil cuando no recuerdas el orden exacto de los parámetros o quieres que tu código sea más legible.
   * Ejemplo de código:  
       
     saludo(edad=25, nombre="Ana") # Los argumentos se pasan por clave-valor

## Funciones III (generadoras)

Las **funciones generadoras** son aquellas que **generan** valores de manera **dinámica y perezosa** (uno a uno), lo que permite ahorrar memoria y gestionar grandes secuencias de datos de manera eficiente. A diferencia de las funciones tradicionales que utilizan return, las funciones generadoras utilizan la palabra clave **yield**.

#### **Características principales:**

1. **yield en lugar de return**:
   * Mientras que return detiene por completo la ejecución de una función, **yield pausa** la ejecución de la función, **guardando su estado**, lo que permite que la función se reanude más tarde desde donde se dejó.
   * Cada vez que la función generadora es llamada, **produce** (o "genera") el siguiente valor en la secuencia.
2. **Uso de next()**:
   * Las funciones generadoras no ejecutan todo el código de una vez, sino que **se llaman secuencialmente** usando la función **next()**, lo que permite que los valores se generen uno a la vez según se soliciten.
   * Si llegamos al final de los valores generados, se lanza la excepción StopIteration.

#### **Ejemplo de una función generadora:**

python

def contador(max\_num):

num = 0

while num < max\_num:

yield num # Devuelve el valor actual y pausa la ejecución

num += 1

# Crear un generador

gen = contador(3)

# Obtener los valores con next()

print(next(gen)) # Salida: 0

print(next(gen)) # Salida: 1

print(next(gen)) # Salida: 2

En este ejemplo, la función contador() genera números de 0 a 2, y cada vez que llamamos a next(), se reanuda la función donde la dejó el último yield.

#### **Ventajas de las funciones generadoras:**

* **Eficiencia en memoria**: Al generar los elementos bajo demanda, las funciones generadoras son útiles para trabajar con grandes cantidades de datos o secuencias infinitas, ya que no almacenan todos los valores en la memoria.
* **Lógica compleja**: Permiten crear secuencias complejas de valores sin necesidad de almacenar todos los resultados de una vez, lo que las hace ideales para algoritmos iterativos o procesos largos.

### **Resumen:**

* Las funciones generadoras usan yield para pausar y reanudar la ejecución, generando valores de uno en uno.
* Para obtener los valores de un generador, se utiliza la función built-in **next()**.
* Son eficientes en términos de memoria y procesamiento cuando trabajamos con grandes volúmenes de datos.

# Manejo de errores

En Python, el manejo de errores se puede realizar de dos maneras:

1. **Errores predefinidos** (los que Python ya reconoce, como ValueError, IndexError, etc.).
2. **Errores personalizados**, donde puedes crear tus propios mensajes de error para dar información más detallada o localizada (por ejemplo, mensajes de error en español para usuarios no angloparlantes).

El manejo de errores es importante para que la aplicación pueda gestionar situaciones imprevistas sin detenerse bruscamente y permitir una mejor experiencia de usuario, mostrando mensajes claros cuando algo va mal.

#### **Ejemplo básico con errores predefinidos:**

python

try:

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

except ValueError:

print("Error: No es un número válido.")

En este ejemplo, si el usuario introduce algo que no es un número, se lanzará un ValueError, y se mostrará un mensaje personalizado.

#### **Crear errores personalizados:**

Puedes lanzar tus propios errores en situaciones específicas usando la palabra clave raise, y personalizar el mensaje que le llegará al usuario.

python

x = -10

if x < 0:

raise ValueError("El número no puede ser negativo.")

#### **Estructura básica de manejo de errores:**

python

try:

# Bloque donde puede ocurrir un error

except TipoDeError as error:

# Qué hacer si ocurre ese error

else:

# Si no ocurre ningún error

finally:

# Código que se ejecuta siempre, ocurra o no el error

* **try**: Aquí se coloca el código que podría causar un error.
* **except**: Se captura el error específico que puede ocurrir, y puedes personalizar el mensaje.
* **else**: Opcional. Se ejecuta si **no** ocurre ningún error.
* **finally**: También opcional, se ejecuta siempre, ocurra o no un error (útil para limpiar recursos, cerrar archivos, etc.).

#### **Uso de pass:**

La instrucción **pass** es útil cuando quieres dejar una parte de tu código sin ejecutar, o si estás construyendo una función que no has completado aún. También se puede usar en la captura de errores si no deseas realizar ninguna acción en el bloque except.

python

try:

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

except ValueError:

pass # No hace nada si ocurre el error, solo ignora el error

### **Importancia del manejo de errores:**

El manejo de errores es esencial en cualquier aplicación para evitar que el programa se detenga de forma inesperada. Si manejas bien los errores, puedes guiar a la persona usuaria sobre qué hacer cuando algo no funciona como se espera, mejorando la experiencia de uso.

# Archivos

Para trabajar con archivos **.CSV** en Python, se utiliza la librería csv, la cual facilita la lectura y escritura de este tipo de archivos.

#### **Ejemplo de lectura de un archivo CSV:**

python

import csv

with open("nombre\_archivo.csv", mode="r", encoding="utf-8") as archivo:

lector\_csv = csv.reader(archivo)

for fila in lector\_csv:

print(fila) # Procesa cada fila del archivo CSV

#### **Ejemplo de escritura en un archivo CSV:**

python

import csv

with open("nombre\_archivo.csv", mode="w", encoding="utf-8", newline="") as archivo:

escritor\_csv = csv.writer(archivo)

escritor\_csv.writerow(["columna1", "columna2", "columna3"]) # Escribe una fila

**Nota**:

* **Modo**: Algunos modos comunes son "r" para leer, "w" para escribir (sobrescribe el archivo), "a" para añadir datos al final del archivo, y "b" para archivos binarios.
* **Permisos**: Dependiendo del modo elegido, también puedes gestionar permisos de acceso al archivo.

#### **Otras librerías útiles:**

* **os**: Para interactuar con el sistema operativo, como obtener el directorio de trabajo actual con os.getcwd().
* **sys**: Permite interactuar con el intérprete de Python y gestionar aspectos como argumentos de la línea de comandos o rutas de módulos.

#### **Ejemplo de uso de os:**

python

import os

print(os.getcwd()) # Obtiene el directorio de trabajo actual

### **Apertura y cierre de archivos**

La función **open()** abre un archivo, pero no lo cierra automáticamente. Si no cierras un archivo manualmente con **close()**, corres el riesgo de corromper el archivo o generar problemas de acceso. Para evitar esto, es **recomendable** utilizar la estructura with, que garantiza que el archivo se cierre automáticamente después de que se complete el bloque de código.

python

with open("nombre\_archivo.txt", "r") as archivo:

contenido = archivo.read()

# Aquí el archivo se cierra automáticamente

Con esta estructura, no es necesario llamar a close() explícitamente.

## Gestionar los permisos de los archivos

Python no tiene un sistema de permisos propio, como en sistemas operativos Linux o Windows, pero proporciona herramientas para interactuar con los permisos del sistema de archivos subyacente. A través del módulo os, puedes verificar y modificar los permisos de archivos y directorios.

#### **1. Verificación de permisos**

El módulo os te permite comprobar si tienes permisos de lectura, escritura o ejecución sobre un archivo. Para esto, puedes usar os.access() con constantes como os.R\_OK para lectura, os.W\_OK para escritura y os.X\_OK para ejecución.

python

import os

if os.access('ruta/al/archivo', os.R\_OK):

print("El archivo tiene permisos de lectura")

#### **2. Cambiar permisos de archivos**

Para cambiar los permisos de un archivo o directorio, puedes usar la función os.chmod(). Esta función te permite establecer los permisos utilizando valores octales, como en los sistemas Unix.

python

import os

os.chmod('ruta/al/archivo', 0o644) # Establece permisos de lectura y escritura para el propietario, y solo lectura para otros

En este ejemplo, 0o644 establece permisos de lectura y escritura para el usuario, y solo lectura para el grupo y otros.

#### **3. Sistemas de archivos virtuales**

Si estás desarrollando una aplicación que requiere un sistema de archivos virtual, puedes usar la biblioteca **pyfilesystem2**. Esta herramienta te permite crear sistemas de archivos en memoria o en otros entornos controlados, ofreciendo flexibilidad para gestionar permisos y accesos de manera programática.

python

import fs

# Crear un sistema de archivos en memoria

mem\_fs = fs.open\_fs('mem://')

# Crear un archivo en este sistema

mem\_fs.touch('archivo.txt') # Crear un archivo llamado archivo.txt

pyfilesystem2 es útil para simular sistemas de archivos o trabajar con archivos de manera temporal, sin necesidad de afectar el sistema de archivos real.

#### **4. Autenticación y autorización (para aplicaciones web)**

Si necesitas controlar el acceso a recursos en una aplicación, Python ofrece bibliotecas como **oauthlib** o **flask-login** para gestionar la autenticación y autorización de usuarios. Estas bibliotecas te permiten verificar la identidad de los usuarios y gestionar los permisos de acceso a los recursos según roles o privilegios asignados.

lambda()

La función **lambda** en Python se utiliza para crear **funciones anónimas** o **funciones pequeñas** de una sola línea. Se llaman "anónimas" porque no necesitan un nombre como las funciones normales que se definen con def.

### **Sintaxis de lambda:**

python

lambda argumentos: expresión

* **lambda**: Palabra clave para definir la función.
* **argumentos**: Los parámetros que tomará la función.
* **expresión**: El valor o expresión que será devuelto. Solo puede contener una única expresión, no puede incluir varias líneas de código o sentencias complejas.

### **Ejemplo básico de lambda:**

Una función normal para sumar dos números se vería así:

python

def sumar(a, b):

return a + b

Usando lambda, podrías hacer lo mismo en una sola línea:

python

sumar = lambda a, b: a + b

Ambas definiciones hacen lo mismo: toman dos argumentos a y b, y devuelven su suma.

#### **Ejemplo de uso:**

python

resultado = sumar(3, 4)

print(resultado) # Salida: 7

### **¿Por qué usar lambda?**

La función lambda es útil cuando necesitas una función rápida y sencilla que solo se usará una vez o en un contexto específico, por ejemplo, como argumento a otra función como map(), filter() o sorted().

### **Ejemplos prácticos:**

#### **1. Usando lambda con map():**

python

numeros = [1, 2, 3, 4, 5]

# Usando lambda para duplicar los números en la lista

duplicados = map(lambda x: x \* 2, numeros)

print(list(duplicados)) # Salida: [2, 4, 6, 8, 10]

Aquí, lambda toma cada elemento x de la lista numeros, lo multiplica por 2, y lo devuelve.

#### **2. Usando lambda con filter():**

python

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Usando lambda para filtrar los números impares

impares = filter(lambda x: x % 2 != 0, numeros)

print(list(impares)) # Salida: [1, 3, 5, 7, 9]

En este caso, lambda verifica si cada número x es impar (x % 2 != 0), y solo devuelve aquellos que cumplen con esta condición.

#### **3. Usando lambda con sorted():**

python

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

# Ordenar los nombres por la longitud de cada nombre

nombres\_ordenados = sorted(nombres, key=lambda nombre: len(nombre))

print(nombres\_ordenados) # Salida: ['Ana', 'Luis', 'Carlos', 'Beatriz']

Aquí, lambda se utiliza para pasar una función a sorted() que ordena los nombres por su longitud (len(nombre)).

### **¿Cuándo evitar lambda?**

Aunque lambda puede ser útil, también puede hacer que el código sea menos legible si se abusa de él o si las expresiones se vuelven complicadas. En esos casos, es mejor definir una función con def para mayor claridad.

### **Resumen:**

* **lambda** permite crear funciones simples y anónimas en una sola línea.
* Se usa típicamente para funciones rápidas que se pasan como argumentos a otras funciones como map(), filter(), o sorted().
* Aunque es poderosa, debe usarse con moderación para mantener el código legible.

### map()

La función **map()** se usa para **aplicar una función a cada elemento** de un iterable (como una lista o tupla) y devolver un nuevo iterable (normalmente una lista).

#### **Sintaxis:**

python

map(función, iterable)

* **función**: Es la función que quieres aplicar a cada elemento.
* **iterable**: Es la lista (o cualquier iterable) cuyos elementos serán procesados por la función.

#### **Ejemplo de map():**

Imagina que tienes una lista de números y quieres duplicar cada número:

python

numeros = [1, 2, 3, 4, 5]

# Función para duplicar un número

def duplicar(n):

return n \* 2

# Usamos map para aplicar la función a cada número

resultado = map(duplicar, numeros)

# Convertimos a lista para ver el resultado

print(list(resultado))

**Salida**:

python

[2, 4, 6, 8, 10]

Aquí, **map()** toma cada número de la lista numeros, lo pasa a la función duplicar(), y devuelve una nueva lista con los resultados.

#### **Ejemplo usando una función lambda:**

Puedes usar una **función lambda** (una función anónima o corta) en lugar de definir una función por separado:

python

numeros = [1, 2, 3, 4, 5]

# Usamos map con una función lambda que duplica el número

resultado = map(lambda n: n \* 2, numeros)

print(list(resultado))

**Salida**:

python

[2, 4, 6, 8, 10]

### filter()

La función **filter()** se usa para **filtrar elementos** de un iterable según una condición que especifiques. Solo los elementos que cumplan con la condición se mantendrán en el resultado.

#### **Sintaxis:**

python

filter(función, iterable)

* **función**: Es la función que devuelve True o False para cada elemento.
* **iterable**: Es la lista (o cualquier iterable) cuyos elementos serán filtrados.

#### **Ejemplo de filter():**

Imagina que quieres filtrar los números impares de una lista:

python

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Función para comprobar si un número es impar

def es\_impar(n):

return n % 2 != 0

# Usamos filter para filtrar solo los números impares

resultado = filter(es\_impar, numeros)

# Convertimos a lista para ver el resultado

print(list(resultado))

**Salida**:

python

[1, 3, 5, 7, 9]

Aquí, **filter()** aplica la función es\_impar() a cada número y devuelve una nueva lista solo con los números que son impares.

#### **Ejemplo usando una función lambda:**

También puedes usar una **función lambda** en lugar de una función tradicional:

python

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Usamos filter con una función lambda para obtener solo los impares

resultado = filter(lambda n: n % 2 != 0, numeros)

print(list(resultado))

**Salida**:

python

[1, 3, 5, 7, 9]

### zip()

La función **zip()** toma dos o más listas (u otros iterables) y las combina **elemento a elemento**, creando tuplas con los elementos correspondientes de cada lista.

#### **Ejemplo básico de zip():**

python

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

edades = [30, 25, 40]

# Usando zip para combinar las dos listas

combinado = zip(nombres, edades)

# Convertimos a lista para ver el resultado

print(list(combinado))

**Salida**:

python

[('Ana', 30), ('Luis', 25), ('Carlos', 40)]

Aquí, **zip()** toma los primeros elementos de ambas listas, luego los segundos, y así sucesivamente. El resultado es una lista de tuplas.

## enumerate()

La función **enumerate()** se utiliza para obtener tanto el **índice** como el **valor** de un iterable (como una lista). Es útil cuando necesitas saber en qué posición está cada elemento mientras lo recorres.

#### **Ejemplo básico de enumerate():**

python

frutas = ["manzana", "naranja", "plátano"]

# Usando enumerate para obtener índice y valor

for indice, fruta in enumerate(frutas):

print(f"{indice}: {fruta}")

**Salida**:

python

0: manzana

1: naranja

2: plátano

En este caso, **enumerate()** asigna un número (el índice) a cada elemento de la lista, comenzando desde 0. Luego, imprime el índice junto con el valor correspondiente.

### **Combinando zip() y enumerate():**

Puedes combinar ambas funciones para hacer cosas interesantes, como enumerar varias listas al mismo tiempo.

#### **Ejemplo combinando zip() y enumerate():**

python

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

edades = [30, 25, 40]

# Combinamos con zip y luego enumeramos

for indice, (nombre, edad) in enumerate(zip(nombres, edades)):

print(f"{indice}: {nombre} tiene {edad} años")

**Salida**:

python

0: Ana tiene 30 años

1: Luis tiene 25 años

2: Carlos tiene 40 años

Aquí estamos combinando nombres y edades con zip() y luego utilizando enumerate() para obtener el índice de cada tupla, lo que da como resultado una lista numerada con nombre y edad.

### **Resumen:**

* **zip()**: Combina varios iterables en tuplas, uniendo los elementos correspondientes.
* **enumerate()**: Añade un índice a los elementos de un iterable, útil para saber en qué posición se encuentra cada elemento.

Ambas funciones son muy útiles para hacer el código más limpio y comprensible

# Funciones decoradoras

**\* Es importante leer la documentación ya que normalmente no es necesario que nos creemos una función decoradora pero aquí explico como funciona una de estas funciones y saber como funciona.**

**\*\* Son muy útiles como funciones genéricas para verificar datos. Por ejemplo para verificar que el tamaño de una foto y tipo de una imagen sea siempre la mismas o cumpla el standard**

Las **funciones decoradoras** (o simplemente **decoradores**) son funciones que **envuelven** otra función con el propósito de **modificar o extender su comportamiento** sin cambiar su código original. Son útiles cuando necesitas reutilizar código o agregar funcionalidades adicionales de manera elegante y sin duplicar lógica.

#### **¿Cómo funciona un decorador?**

Un decorador toma una función como argumento, ejecuta algún código antes o después de la función original, y devuelve una nueva función que reemplaza a la original.

### **Ejemplo básico de un decorador:**

python

def decorador(func):

def envoltura():

print("Antes de ejecutar la función")

func() # Llamada a la función original

print("Después de ejecutar la función")

return envoltura

@decorador

def saludar():

print("Hola, ¡bienvenidos!")

saludar()

**Salida**:

Antes de ejecutar la función

Hola, ¡bienvenidos!

Después de ejecutar la función

#### **Explicación:**

* **decorador(func)**: Es el decorador que envuelve la función saludar(). Modifica el comportamiento añadiendo código antes y después de la ejecución de la función original.
* **@decorador**: Este símbolo **aplica** el decorador a la función saludar(), lo que significa que cada vez que se llama a saludar(), primero se ejecuta el código en envoltura().

### **¿Cuándo son útiles los decoradores?**

Los decoradores son útiles en situaciones como:

* **Validación de permisos**: Verificar si un usuario tiene permisos antes de ejecutar una función.
* **Logging**: Registrar información de depuración antes y después de la ejecución de una función.
* **Control de ejecución**: Modificar el comportamiento de la función dependiendo de ciertos estados o condiciones.

### **Resumen:(sintaxis standard - WRAPPER)**

* Las funciones decoradoras son poderosas herramientas para modificar o extender el comportamiento de una función.
* Se utilizan frecuentemente para **reutilizar código** y **añadir funcionalidades** sin modificar la función original.
* Ejemplo de sintaxis standard de una función decoradora

**def mi\_decorador(func): # La función decoradora acepta una función como argumento**

**def envoltura(\*args, \*\*kwargs): # Función interna que envuelve la original**

**# Aquí puedes añadir código que se ejecute antes o después**

**print("Antes de ejecutar la función decorada")**

**resultado = func(\*args, \*\*kwargs) # Llamamos a la función original**

**print("Después de ejecutar la función decorada")**

**return resultado # Devolvemos el resultado de la función decorada**

**return envoltura # Devolvemos la función interna, NO la ejecutamos**

# \*args y \*\*kargs

En Python, \*args y \*\*kwargs son formas de permitir que una función acepte un número variable de argumentos.

### **\*args**

Cuando usas \*args en una función, estás diciendo que esa función puede recibir cualquier cantidad de argumentos **posicionales**. Es decir, puedes pasarle muchos valores separados por comas, y Python los recogerá en una **tupla**.  
Ejemplo de código:  
  
def sumar(\*args):

return sum(args)

print(sumar(1, 2, 3)) # Salida: 6

print(sumar(4, 5)) # Salida: 9

En este caso, \*args recoge los números que le pasas a la función y los convierte en una tupla (1, 2, 3) o (4, 5).

### **\*\*kwargs**

\*\*kwargs se utiliza para aceptar cualquier cantidad de argumentos **con nombre** o **clave-valor**. Python convierte esos argumentos en un **diccionario**.  
Ejemplo de código:  
  
def mostrar\_info(\*\*kwargs):

for clave, valor in kwargs.items():

print(f"{clave}: {valor}")

mostrar\_info(nombre="Ana", edad=30)

# Salida:

# nombre: Ana

# edad: 30

En este caso, \*\*kwargs recoge los argumentos con nombre (nombre="Ana", edad=30) y los convierte en un diccionario {nombre: "Ana", edad: 30}.

### **Relación con \* y \*\***

* El \* se usa para desempaquetar listas o tuplas en argumentos posicionales.
* El \*\* se usa para desempaquetar diccionarios en argumentos con nombre.

Ejemplo combinando todo:

python

def mostrar\_todo(\*args, \*\*kwargs):

print("args:", args) # Esto imprime los argumentos posicionales como una tupla.

print("kwargs:", kwargs) # Esto imprime los argumentos con nombre como un diccionario.

mostrar\_todo(1, 2, 3, nombre="Ana", edad=30)

# Salida:

# args: (1, 2, 3)

# kwargs: {'nombre': 'Ana', 'edad': 30}

**En resumen:**

* \*args te permite pasar varios argumentos sin especificar el número exacto.
* \*\*kwargs te permite pasar varios argumentos con nombre.
* Los \* y \*\* también sirven para desempaquetar listas, tuplas y diccionarios cuando llamas a una función.

# PARTE II - Programación orientada a objetos (POO)

La **POO** organiza el código en torno a **objetos**. Un objeto puede ser cualquier cosa con características (atributos) y comportamientos (métodos). Por ejemplo, en nuestro caso:

* Un **Músico** es un objeto.
* Un **Instrumento** es otro objeto.

Cada objeto tiene **atributos** que lo describen (como el nombre del músico y el instrumento que toca) y **métodos** que definen lo que puede hacer (como tocar el instrumento).

En **Python**, las clases y los tipos son lo mismo. Una **clase** define el tipo de objeto y su comportamiento, como un "molde" que luego usamos para crear instancias concretas.

### **Conceptos clave en POO**

#### **1. Instrucción (NO se guarda en memoria)**

Una **instrucción** es una orden específica que le das al programa para que haga algo en un momento determinado. No se guarda en memoria, ya que es algo temporal. Por ejemplo, una instrucción podría ser imprimir algo en la pantalla:

python

Copiar código

print("El músico está tocando la guitarra")

Una vez que esta instrucción se ejecuta, desaparece.

#### **2. Clase (se guarda en memoria)**

Una **clase** es como un "molde" que define cómo será un objeto. En nuestro ejemplo de músicos, la clase **Músico** describe qué atributos (nombre e instrumento) y métodos (acciones) tienen los músicos. Las clases **se guardan en memoria** porque el programa las necesita para crear y gestionar objetos.

Ejemplo de una clase **Músico**:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

Aquí la clase **Músico** tiene dos atributos: **nombre** y **instrumento**.

#### **3. Instancia de la clase (se guarda en memoria)**

Una **instancia** es un **objeto concreto** creado a partir de una clase. Si la clase **Músico** es el molde, una instancia sería un músico específico, como "Ana, que toca la guitarra". Las instancias **se guardan en memoria** porque representan los objetos reales con los que el programa trabaja.

Ejemplo de instancia:

python

Copiar código

ana = Musico("Ana", "guitarra")

Aquí, **ana** es una instancia de la clase **Músico**, con **nombre = "Ana"** e **instrumento = "guitarra"**.

#### **4. Atributo (se guarda en memoria)**

Un **atributo** es una característica o propiedad de una instancia. En nuestro ejemplo, los atributos de **ana** son **nombre** ("Ana") e **instrumento** ("guitarra"). Los atributos **se guardan en memoria** porque son los valores específicos de cada objeto.

Ejemplo de acceso a los atributos:

python

Copiar código

print(ana.nombre) # Muestra "Ana"

print(ana.instrumento) # Muestra "guitarra"

#### **5. ¿Qué hace dir()? ¿Es un método?**

dir() es una **función** en Python que te muestra una lista de todos los atributos y métodos de un objeto, pero **no es un método** de la clase, es una herramienta general de Python.

Ejemplo de uso de dir():

python

Copiar código

print(dir(ana)) # Muestra todos los atributos y métodos de 'ana'

#### **6. Método**

Un **método** es una función que pertenece a una clase y define qué puede hacer un objeto. Por ejemplo, un método en la clase **Músico** podría ser **tocar\_instrumento()**, que indica lo que hace el músico.

Ejemplo de un método:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

def tocar\_instrumento(self):

print(f"{self.nombre} está tocando la {self.instrumento}")

Cuando llamamos al método:

python

Copiar código

ana.tocar\_instrumento() # Muestra: "Ana está tocando la guitarra"

### **Relación de todos estos conceptos con objetos en la POO**

En la POO, todo gira en torno a los **objetos**. En nuestro ejemplo:

* **Objeto**: Músico (por ejemplo, **Ana**).
* **Clase**: Define cómo es el objeto (por ejemplo, **Músico**).
* **Atributos**: Características del objeto (por ejemplo, **nombre** e **instrumento**).
* **Métodos**: Acciones que puede realizar el objeto (por ejemplo, **tocar\_instrumento()**).

### **El papel de self**

En Python, **self** es una palabra clave que se utiliza dentro de las clases para referirse a la **instancia actual** de un objeto. Es necesario para que los métodos puedan acceder a los atributos y métodos del objeto con el que estás trabajando.

Ejemplo de uso de self:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre # 'self.nombre' hace referencia al atributo de la instancia

self.instrumento = instrumento

def tocar\_instrumento(self):

print(f"{self.nombre} está tocando la {self.instrumento}")

Cuando llamamos al método:

python

Copiar código

ana.tocar\_instrumento() # "Ana está tocando la guitarra"

En este caso, **self.nombre** se refiere a **"Ana"** y **self.instrumento** a **"guitarra"**. **self** asegura que se está trabajando con la instancia correcta.

### **Diferencias entre método de clase y método de instancia:**

**Método de clase**: Está asociado con la clase en general y se puede llamar sin crear una instancia. Usa **cls** en lugar de **self** y se define con el decorador **@classmethod**.  
Ejemplo:  
python  
Copiar código  
@classmethod

def tipo\_de\_musico(cls):

print(f"Todos los músicos son {cls.tipo}")

**Método de instancia**: Se asocia con una instancia específica de la clase y necesita que crees un objeto antes de poder llamarlo. Usa **self** para referirse a la instancia.  
Ejemplo:  
python  
Copiar código  
def tocar\_instrumento(self):

print(f"{self.nombre} está tocando la {self.instrumento}")

### **Resumen:**

* **Clase**: El molde para crear objetos (por ejemplo, **Músico**).
* **Instancia**: Un objeto específico creado a partir de una clase (por ejemplo, **Ana**).
* **Atributos**: Características de un objeto (por ejemplo, **nombre** e **instrumento**).
* **Métodos**: Acciones que el objeto puede realizar (por ejemplo, **tocar\_instrumento()**).
* **self**: Hace referencia a la instancia actual del objeto.
* **Método de clase**: Se usa con la clase en general, sin instancias, y usa **cls**.
* **Método de instancia**: Se usa con instancias específicas y usa **self**.

# Métodos especiales o mágicos

Los **métodos especiales o mágicos** en Python son métodos que tienen nombres específicos rodeados por dobles guiones bajos, como **\_\_init\_\_**, **\_\_str\_\_**, o **\_\_len\_\_**. Estos métodos permiten que las instancias de las clases puedan tener comportamientos especiales, como convertir un objeto en una cadena, sumar objetos, o compararlos. Python invoca estos métodos de manera automática cuando usas ciertas funciones o operadores.

Vamos a ver los más comunes:

### **1. \_\_init\_\_ (Inicializador)**

El método **\_\_init\_\_** es el más conocido. Es el **constructor** de la clase y se ejecuta automáticamente cuando se crea una nueva instancia. Se usa para inicializar los atributos del objeto.

Ejemplo:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

ana = Musico("Ana", "guitarra") # '\_\_init\_\_' se ejecuta automáticamente

### **2. \_\_str\_\_ (Representación en cadena)**

El método **\_\_str\_\_** define lo que se debe mostrar cuando conviertes un objeto en una cadena de texto (por ejemplo, con print). Si no lo defines, Python mostrará una representación predeterminada que no suele ser muy legible.

Ejemplo:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

def \_\_str\_\_(self):

return f"{self.nombre} toca la {self.instrumento}"

ana = Musico("Ana", "guitarra")

print(ana) # "Ana toca la guitarra"

### **3. \_\_repr\_\_ (Representación oficial)**

El método **\_\_repr\_\_** también devuelve una representación del objeto, pero está pensado para los desarrolladores. Se usa en el **modo interactivo de Python** o cuando quieres una representación oficial que sirva para reconstruir el objeto.

Ejemplo:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

def \_\_repr\_\_(self):

return f"Musico('{self.nombre}', '{self.instrumento}')"

ana = Musico("Ana", "guitarra")

print(repr(ana)) # "Musico('Ana', 'guitarra')"

### **4. \_\_len\_\_ (Longitud)**

El método **\_\_len\_\_** se usa para devolver la longitud de un objeto, por ejemplo, si quieres definir cuántos elementos tiene un objeto personalizado.

Ejemplo:

python

Copiar código

class Banda:

def \_\_init\_\_(self, musicos):

self.musicos = musicos

def \_\_len\_\_(self):

return len(self.musicos)

banda = Banda(["Ana", "Carlos", "Luis"])

print(len(banda)) # 3

### **5. \_\_eq\_\_ (Comparación de igualdad)**

El método **\_\_eq\_\_** permite comparar dos objetos usando el operador de igualdad (==). Sin este método, Python compara las referencias en memoria, pero si lo defines, puedes comparar atributos específicos.

Ejemplo:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

self.nombre = nombre

self.instrumento = instrumento

def \_\_eq\_\_(self, otro):

return self.nombre == otro.nombre and self.instrumento == otro.instrumento

ana = Musico("Ana", "guitarra")

otro\_musico = Musico("Ana", "guitarra")

print(ana == otro\_musico) # True

### **6. \_\_add\_\_ (Suma de objetos)**

Este método se utiliza para definir el comportamiento del operador + cuando se usa entre dos objetos de una clase. Por ejemplo, si quisieras sumar el número de músicos en dos bandas.

Ejemplo:

python

Copiar código

class Banda:

def \_\_init\_\_(self, musicos):

self.musicos = musicos

def \_\_add\_\_(self, otra\_banda):

return Banda(self.musicos + otra\_banda.musicos)

banda1 = Banda(["Ana", "Carlos"])

banda2 = Banda(["Luis", "Pedro"])

banda\_grande = banda1 + banda2 # Suma los músicos de ambas bandas

print(banda\_grande.musicos) # ['Ana', 'Carlos', 'Luis', 'Pedro']

### **7. \_\_call\_\_ (Convertir el objeto en una función)**

El método **\_\_call\_\_** permite que un objeto se comporte como una función. Cuando defines este método, puedes "llamar" a la instancia como si fuera una función.

Ejemplo:

python

Copiar código

class Saludo:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def \_\_call\_\_(self):

print(f"Hola, {self.nombre}")

saludo = Saludo("Ana")

saludo() # "Hola, Ana"

### **8. \_\_del\_\_ (Destructor)**

El método **\_\_del\_\_** se llama cuando un objeto es destruido o eliminado. No es muy común usarlo en la práctica, pero puede ser útil para liberar recursos o realizar alguna acción cuando un objeto deja de existir.

Ejemplo:

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def \_\_del\_\_(self):

print(f"{self.nombre} ha sido eliminado")

ana = Musico("Ana")

del ana # "Ana ha sido eliminado" se imprimirá al eliminar el objeto

### **Resumen:**

* **Métodos especiales o mágicos** son aquellos que tienen dobles guiones bajos (\_\_) al principio y al final de su nombre.
* **\_\_init\_\_**, **\_\_str\_\_**, **\_\_repr\_\_**, **\_\_len\_\_**, **\_\_eq\_\_**, **\_\_add\_\_**, **\_\_call\_\_**, y **\_\_del\_\_** son algunos de los métodos mágicos más comunes.
* Estos métodos permiten definir cómo los objetos de tu clase se comportan frente a operaciones específicas, como ser impresos, sumados, comparados o incluso llamados como funciones.

# El constructor

El **constructor** es un método especial en las clases de Python (y otros lenguajes orientados a objetos) que se encarga de **inicializar los atributos** de una instancia cuando se crea un objeto de esa clase. En Python, el constructor se define mediante el método **\_\_init\_\_()**.

Cada vez que creas una nueva instancia de una clase, el constructor **\_\_init\_\_()** se ejecuta automáticamente. El propósito principal del constructor es asignar valores iniciales a los atributos del objeto.

### **Concepto clave:**

* **Constructor**: Método especial que se ejecuta automáticamente cuando se crea un objeto. Inicializa los atributos de la instancia.

### **Ejemplo de constructor:**

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre, instrumento):

# Este es el constructor

self.nombre = nombre # Inicializamos el atributo 'nombre'

self.instrumento = instrumento # Inicializamos el atributo 'instrumento'

# Crear una instancia de la clase 'Musico'

ana = Musico("Ana", "guitarra")

# Acceder a los atributos de la instancia

print(ana.nombre) # "Ana"

print(ana.instrumento) # "guitarra"

En este ejemplo:

* **\_\_init\_\_()** es el constructor.
* Cuando creamos el objeto **ana = Musico("Ana", "guitarra")**, se llama automáticamente al constructor, asignando los valores **"Ana"** y **"guitarra"** a los atributos **nombre** e **instrumento** del objeto.

### **Función del constructor:**

* **Automatiza la inicialización** de los atributos cuando se crea una instancia de la clase.
* Te permite pasar parámetros para personalizar el objeto cuando lo creas.

### **¿Por qué es importante?**

El constructor es esencial en la Programación Orientada a Objetos porque permite que cada instancia tenga su propia versión de los atributos, personalizándola en el momento en que se crea el objeto. Sin un constructor, todas las instancias tendrían que ser inicializadas manualmente después de su creación, lo que complicaría el código.

# El destructor

El **destructor** es un método especial en Python que se ejecuta automáticamente cuando un objeto **deja de existir** o se elimina. Este método es llamado **\_\_del\_\_()**, y su función principal es **liberar recursos** o realizar alguna tarea de limpieza cuando el objeto ya no es necesario, como cerrar archivos, liberar memoria o desconectar bases de datos.

Aunque Python gestiona automáticamente la memoria con un sistema de **recolección de basura** (garbage collection), a veces es útil definir un destructor si necesitas asegurarte de que se lleven a cabo acciones específicas cuando un objeto es destruido.

### **Concepto clave:**

* **Destructor**: Método especial llamado **\_\_del\_\_()** que se ejecuta cuando un objeto se elimina o deja de estar en uso. Sirve para liberar recursos o hacer una limpieza final.

### **Ejemplo de destructor:**

python

Copiar código

class Musico:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

print(f"{self.nombre} ha sido creado.")

def \_\_del\_\_(self):

print(f"{self.nombre} ha sido eliminado.")

# Crear una instancia de la clase 'Musico'

ana = Musico("Ana")

# Eliminar el objeto explicitamente

del ana # Se llamará al destructor '\_\_del\_\_()'

**Salida del programa:**

Copiar código

Ana ha sido creado.

Ana ha sido eliminado.

### **Funcionamiento:**

1. **Cuando creas el objeto ana**, el constructor \_\_init\_\_() se ejecuta, inicializando el nombre del músico y mostrando el mensaje "Ana ha sido creado".
2. **Cuando eliminas el objeto con del ana**, el método destructor \_\_del\_\_() se ejecuta, y muestra "Ana ha sido eliminado".

### **¿Cuándo se llama al destructor?**

* Cuando llamas explícitamente a **del** sobre una instancia, como en el ejemplo anterior.
* Cuando el programa ya no necesita el objeto y Python lo elimina automáticamente con su sistema de recolección de basura (por ejemplo, al final de la ejecución del programa).

### **¿Cuándo usar el destructor?**

* El **destructor** es útil cuando quieres asegurarte de que un objeto realice tareas específicas al ser destruido, como:
  + **Cerrar archivos** que el objeto estaba usando.
  + **Liberar recursos** de red o memoria.
  + **Guardar datos** o hacer algún registro antes de eliminar el objeto.

Sin embargo, en la mayoría de los casos en Python, no es necesario usar destructores ya que el sistema de recolección de basura gestiona automáticamente la liberación de recursos. Los destructores suelen ser útiles en casos específicos donde debes manejar manualmente ciertos recursos.

### **Resumen:**

* **\_\_del\_\_()** es el método destructor.
* Se ejecuta cuando un objeto es eliminado o ya no está en uso.
* Es útil para limpiar recursos o realizar tareas específicas antes de que el objeto desaparezca.

# Herencia y polimorfismo

### **Herencia**

La **herencia** permite que una clase (subclase) tome atributos y métodos de otra clase (superclase), evitando duplicar código. Por ejemplo, si tienes una clase Animal con el atributo nombre, una subclase Perro hereda este atributo y lo usa directamente.

python

Copiar código

class Animal:

def \_\_init\_\_(self, nombre):

self.nombre = nombre # Atributo que será heredado

class Perro(Animal): # Perro hereda de Animal

pass

# Ejemplo de uso

mi\_perro = Perro("Firulais")

print(mi\_perro.nombre) # Salida: Firulais

### **super() en herencia**

super() permite que una subclase acceda a métodos o atributos de la superclase sin duplicarlos. Por ejemplo, Perro puede usar super() para inicializar nombre y otros atributos definidos en Animal.

python

Copiar código

class Animal:

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

class Perro(Animal):

def \_\_init\_\_(self, nombre, edad, raza):

super().\_\_init\_\_(nombre, edad) # Llama al constructor de Animal

self.raza = raza # Atributo específico de Perro

# Ejemplo de uso

mi\_perro = Perro("Firulais", 5, "Bulldog")

print(mi\_perro.nombre, mi\_perro.edad, mi\_perro.raza) # Salida: Firulais 5 Bulldog

### **raise en herencia**

Se usa para forzar ciertas reglas en la clase base, como asegurar que los métodos clave se implementen en las subclases. Por ejemplo, lanzar NotImplementedError en un método en Animal asegura que Perro o cualquier subclase defina su propio comportamiento.

python

Copiar código

class Animal:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def hacer\_sonido(self):

raise NotImplementedError("Este método debe ser implementado en la subclase")

class Perro(Animal):

def hacer\_sonido(self):

return "Guau"

# Ejemplo de uso

mi\_perro = Perro("Firulais")

print(mi\_perro.hacer\_sonido()) # Salida: Guau

mi\_animal = Animal("Genérico")

print(mi\_animal.hacer\_sonido()) # Lanzará NotImplementedError

### **Polimorfismo**

El **polimorfismo** permite que métodos compartidos entre varias clases respondan de manera específica según la clase que los llame. Es decir, hacer\_sonido() puede existir en Animal, Perro, y Gato, pero cada clase tiene su propia versión del método.

python

Copiar código

class Gato(Animal):

def hacer\_sonido(self):

return "Miau"

# Ejemplo de uso

animales = [Perro("Firulais"), Gato("Michi")]

for animal in animales:

print(animal.hacer\_sonido()) # Salida: Guau Miau

### **raise en polimorfismo**

raise se utiliza en métodos de la clase base para señalar que el comportamiento debe personalizarse en cada subclase, evitando que se use un método incompleto en la clase base.

python

Copiar código

class Animal:

def hacer\_sonido(self):

raise NotImplementedError("Este método debe ser implementado en la subclase")

Aquí, NotImplementedError fuerza a las subclases a implementar hacer\_sonido, permitiendo un comportamiento específico según la clase que lo llame.

### **try...except y raise en herencia y polimorfismo**

* **raise**: Se usa en clases para lanzar errores específicos y asegurar el diseño de las subclases (por ejemplo, NotImplementedError).
* **try...except**: Es el mecanismo que captura y maneja los errores lanzados (ya sea por raise u otros fallos), permitiendo que el programa siga funcionando y responda de manera controlada.

python

Copiar código

try:

mi\_animal = Animal("Genérico")

print(mi\_animal.hacer\_sonido()) # Intento de usar hacer\_sonido en la clase base

except NotImplementedError as e:

print(f"Error: {e}")

### **Resumen**

En resumen, raise es útil para marcar errores en la estructura de clases, mientras que try...except maneja esos errores cuando el programa se ejecuta, asegurando que responda sin detenerse bruscamente. super() facilita el acceso a métodos de la superclase, lo que permite extender el código de la clase base en la subclase sin duplicarlo.

# Encapsulamiento:

En Python, **la encapsulación** es un principio de la programación orientada a objetos que restringe el acceso directo a los atributos y métodos de una clase, ocultándolos del exterior. La idea es controlar cómo se accede y modifica el estado de un objeto, lo que ayuda a **proteger los datos** y **mantener la integridad de la clase**.

### **Cómo Funciona la Encapsulación en Python**

En Python, no existe una encapsulación "fuerte" como en otros lenguajes (por ejemplo, private en Java o C++). Sin embargo, Python usa convenciones de nombres para señalar el nivel de acceso a los atributos y métodos.

1. **Atributos y métodos públicos**:
   * **Sin guión bajo** al inicio (nombre, edad): se consideran **públicos** y están disponibles desde fuera de la clase.
2. **Atributos y métodos protegidos**:
   * **Un guión bajo al inicio** (\_nombre): se considera **protegido** y, aunque accesible desde fuera, señala que el atributo o método no debe usarse fuera de la clase o sus subclases.
3. **Atributos y métodos privados**:
   * **Doble guión bajo al inicio** (\_\_nombre): se considera **privado** y no es directamente accesible desde fuera de la clase. Python usa un sistema llamado **name mangling** para evitar el acceso directo, cambiando el nombre del atributo internamente. Sin embargo, puede accederse indirectamente si es necesario.

### **Ejemplo de Encapsulación en Python**

Aquí tienes un ejemplo de una clase que encapsula algunos de sus atributos y métodos:

python

Copiar código

class Persona:

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre # Atributo público

self.\_edad = edad # Atributo protegido

self.\_\_salario = 0 # Atributo privado

def obtener\_salario(self):

return self.\_\_salario

def establecer\_salario(self, salario):

if salario >= 0:

self.\_\_salario = salario # Modificar salario solo si es positivo

else:

print("El salario no puede ser negativo.")

# Ejemplo de uso

persona = Persona("Ana", 30)

# Acceso público

print(persona.nombre) # Ana

# Acceso protegido (no recomendado, pero posible)

print(persona.\_edad) # 30

# Acceso privado (solo mediante el método)

persona.establecer\_salario(3000)

print(persona.obtener\_salario()) # 3000

# Intento de acceso directo al atributo privado

# print(persona.\_\_salario) # Esto causará un error

### **Ventajas de la Encapsulación**

1. **Control del acceso a los datos**: Permite que ciertos atributos solo se modifiquen mediante métodos controlados.
2. **Protección de datos**: Evita que se modifiquen atributos de manera inesperada desde fuera de la clase.
3. **Facilidad para cambios futuros**: Si en el futuro decides cambiar la implementación interna, puedes hacerlo sin afectar el código que usa esa clase mientras mantengas los métodos públicos consistentes.

### **Resumen**

La encapsulación en Python se basa en convenciones y el uso de \_ y \_\_ al comienzo de los nombres de atributos y métodos para indicar el nivel de acceso. Aunque Python permite cierto acceso, seguir estas convenciones ayuda a organizar mejor el código y a proteger los datos del objeto.