Estructuras de control

python

Indice

1. Estructuras de control

Las estructuras de control permiten decidir qué partes del programa se ejecutan y repetir acciones varias veces.
En Python, las estructuras de control más importantes son:

- Condicionales → if, elif, else
- Bucles → for, while
- Control del flujo → break, continue, pass

Python se basa en la indentación (sangría) para agrupar bloques de código.
Cada bloque debe tener la misma cantidad de espacios (4 por convención) o usar tabuladores (ya que el propio VSCode transforma esos tabuladores a espacios).

1.1 Sentencia condicional

La sentencia condicional más simple es "if", sirve para entrar en un trozo de código solo si se cumple una condición, ejemplo:

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

Aquí podemos ver que después del "if" hay que poner ":" para indicar que a partir de aquí va el contenido del "if" y este contenido debe tener una tabulación, ojo que, si ponemos varias líneas, solo estará dentro del "if" las que tengan tabulación.

```
edad=19
if edad >= 18:
    print("Eres mayor de edad")
    print("Esto está dentro de if")
print("Esto estaría fuera del if")
```

Si queremos que se ejecute un código que no cumpla la condición el "if", usar "else":

```
edad=19
if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Igual que con el "if", poner ":" al final de la sentencia else y su contenido debe tener tabulación delante.

Si quiero tener más condiciones a cumplir, se puede usar "elif":

```
nota = 7
if nota >= 9:
   print("Sobresaliente")
elif nota >= 7:
```

```
    print("Notable")
elif nota >= 5:
    print("Aprobado")
else:
    print("Suspenso")
```

En este caso, solo se ejecuta una rama del if (la primera que cumpla la condición).

Condiciones combinadas usando operadores lógicos

Los operadores lógicos son AND y OR, AND solo da como resultado true si ambos operandos son true: true AND true == true y OR da como resultado false si ambos son false, en caso contrario es true. Ejemplo:

```
edad = 25
tiene_carnet = True
if edad >= 18 and tiene_carnet:
    print("Puedes conducir")
else:
    print("No puedes conducir")
```

Una forma fácil de acordarse si el resultado es true o false en un operador lógico, es pensar en AND como multiplicación y OR como una suma, así podemos indicar (sustituimos true como 1 y false como 0):

0 AND 0 = 0 0 OR 0 = 0
0 AND 1 = 0 0 OR 1 = 1
1 AND 0 = 0 1 OR 0 = 1
1 AND 1 = 1 1 OR 1 =1

Negación se haría con not, por ejemplo, not True dará como resultado False.

No siempre es necesario que la condición de una sentencia condicional o bucle tenga una comparación, sino que podemos poner directamente un valor booleano (o variable que almacena un valor booleano). En el ejemplo, el flujo entra en el bloque IF si “var_booleano” tiene almacenado el valor True.

```
var_booleano = True
if var_booleano:
    print("La variable es verdadera")
else:
    print("La variable es falsa")
```

Además de usar valores booleanos True y False directamente como condiciones, también se pueden evaluar otros valores literales como enteros, cadenas o el valor None. El resultado de la evaluación está predefinido en función de las siguientes reglas:

- Son evaluados a False:
    - El 0 y el 0.0
    - Una estructura vacía (por ejemplo, una lista).

- Una cadena vacía, abre y cierra comillas sin texto.
- El valor None.

● Son evaluados a True:
- Una estructura con contenido.
- Una cadena con al menos un carácter.
- Un numérico distinto de 0.

Operadores de comparación

<, >, <=, >=, ==, !=

Las comparaciones de cadenas pueden hacerse de la forma: manzana < pera, pero lo que hace es comparar la primera letra de manzana con la primera letra de pera, y llega a la conclusión de que m < p, porque la "m" viene antes que la "p" en el abecedario (aunque realmente es porque viene antes en su código ASCII). Si tengo manzana < mora, como la primera letra es igual en ambas cadenas, compara con la segunda letra, en este caso como a<o, deduce que manzana es menor que mora.

Ejercicio 1

Pide la edad del usuario y muestra un mensaje según su rango de edad:
- Menor de 12 → "Eres un niño."
- Entre 12 y 17 → "Eres un adolescente."
- Entre 18 y 64 → "Eres un adulto."
- 65 o más → "Eres un jubilado."

Ternarias

Es una forma corta de hacer un if … else, lo primero que se pone es el código que se ejecuta si cumple la condición, luego el if (condición) else y el código que no cumple.

Ejemplo:

```
edad = 18
mensaje = "Es mayor de edad" if edad>=18 else "Es menor de edad"
print(mensaje)
```

```
numero = 10
paridad = "par" if numero % 2 == 0 else "impar"
print(f"El número {numero} es {paridad}.")
```

Podemos utilizar el operador "in" en nuestras condiciones, este operador se utiliza tanto con cadenas de texto como con listas, tuplas, conjuntos o diccionarios.

En Python, los operadores "in" y "not in" se utilizan para comprobar si un valor se encuentra (o no) dentro de una secuencia o colección.

Estos operadores devuelven un valor booleano (True o False) según el resultado de la comprobación.

Sintaxis general:

```
elemento in secuencia
elemento not in secuencia
```

- in → devuelve True si el elemento está contenido en la secuencia.
- not in → devuelve True si el elemento no está contenido.

Uso con cadenas de texto(str)

En una cadena, “in” verifica si una letra o palabra está dentro de otra cadena.

```
frase = "Python es divertido"


print("Python" in frase)      # True
print("python" in frase)      # False (distingue mayúsculas/minúsculas)
print("diver" in frase)       # True (aparece dentro de la palabra)
print("Java" not in frase)    # True
```

En cadenas, “in” funciona carácter por carácter, y la comparación distingue mayúsculas y minúsculas.

Si necesitas ignorar las mayúsculas, puedes comparar usando .lower() o .upper():

```
"python" in frase.lower()
```

Uso con listas, tuplas y conjuntos

Aunque no las hemos visto en profundidad, indicar que también puedes usar “in” y “not in” para comprobar si un valor se encuentra dentro de colecciones como listas (list), tuplas (tuple) o conjuntos (set).

```
colores = ["rojo", "verde", "azul"]


print("rojo" in colores)     # True
print("negro" in colores)    # False
print("amarillo" not in colores)  # True
```

Si solo esperas una letra, como S o N, se puede usar:

```
if respuesta in "SN":
```

Pero si esperamos varias letras, como SI o NO, mejor usar una tupla:

```
if respuesta in ("SI", "NO"):
```

Ejercicio 2

Pide al usuario si desea continuar (solo podrá responder “S” o “N” (tanto en mayúsculas como en minúsculas), si el usuario introduce otra letra que no sea alguna de esas 2, dará el error “Respuesta no válida, introduce S o N”, si la respuesta es S mostrará “Continuamos…” y si es N mostrará “Fin del programa”.

1.2 Estructuras iterativas FOR

Como ya hemos comentado, los bucles permiten repetir acciones varias veces.
El bucle “for” se usa cuando sabemos cuántas veces queremos repetir una acción.
Python recorre directamente los elementos de una secuencia (lista, cadena, rango…).
Ejemplo para recorrer una lista:

```
frutas = ["manzana", "pera", "naranja"]
for fruta in frutas:
    print("Me gusta la", fruta)
```

Si queremos obtener el índice de cada elemento, usar enumerate():

```
frutas = ["manzana", "pera", "naranja"]
for indice,fruta in enumerate(frutas):
    print(f"El indice {indice} corresponde a la fruta: {fruta}")
```

En la primera variable (en mi caso "indice") guarda el número del índice del array y en la segunda variable (en mi caso "fruta") guarda el contenido del array del índice actual.

Es habitual tener que hacer un "for" de un rango de números, para ello utilizaremos range().
range(inicio, fin, paso) genera una secuencia de números. Por defecto, empieza en 0 y el fin no se incluye.

```
for i in range(5):
    print(i)
# 0, 1, 2, 3, 4
```

Si solo indicamos un parámetro, empieza en 0 y en este caso no incluye 5, por lo que muestra los valores entre 0 y 4.
Si queremos mostrar un rango entre 2 números y no empezar en cero, utilizaremos los dos primeros parámetros: inicio y fin

```
for i in range(7, 10):
    print(i)
# 7,8,9
```

Si queremos mostrar un rango pero que los números no vayan de 1 en 1, podemos usar el parámetro "paso":

```
for i in range(1, 10, 2):
    print(i)
# 1, 3, 5, 7, 9
```

Si queremos mostrar los números de mayor a menor, en "paso" pondremos -1 si queremos que cuente de 1 en 1 hacia atrás, -2 si el salto es de 2 en 2 hacia atrás… Ten en cuenta que como vamos hacia atrás, inicio debe ser el número mayor.

```
for i in range(10, 1, -2):
    print(i)
# 10, 8, 6, 4, 2
```

Hay situaciones en las que no necesitamos usar la variable que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces. Para estos casos se suele recomendar usar el guion bajo "_" como nombre de variable, que da a entender que no estamos usando esta variable de forma explícita:

```
for _ in range(5):
    print("Se repite 5 veces")
```

En Python, una cadena es un elemento iterable, por lo que podemos usar un bucle for con ella:

```
cadena = "hola"
for caracter in cadena:
```

```
    print(caracter)
```

Si tienes un range y quieres obtener una lista de esos elementos del range, porque recordemos que range es un tipo de datos, NO una lista, usar:

```
nums = range(10)
lista_de_numeros = list(nums)
print(lista_de_numeros)
```

Bucles anidados

Como ya vimos en las sentencias condicionales, el anidamiento es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Veamos un ejemplo de 2 bucles anidados en el que generamos las tablas de multiplicar del 1 al 9:

```
for tabla in range(1,10):
    print(f"\nTabla del {tabla}:")
    for numero in range(1,11):
        resultado = tabla * numero
        print(f"{tabla} x {numero} = {resultado}")
```

Ejercicio 3

Pide al usuario un número y muestra su tabla de multiplicar (del 1 al 10).

Comprensión de listas (list comprehension)

Si queremos convertir todos los elementos de una lista en mayúsculas, indicar entre corchetes [lo_que_le_queremos_hacer_a_cada_elemento for X in lista]. Ejemplo:

```
animales = ["perro", "gato", "pez", "loro", "canario", "hamster", "tortuga"]
animales_mayusculas = [animal.upper() for animal in animales]
print(animales_mayusculas)
```

Hay que tener en cuenta que si uso “for animal in …”, animal es cada elemento de la lista, por lo que tenemos que usar el mismo nombre al poner animal.upper() (si cambio el nombre de uno, debo cambiar el nombre del otro).

Comprensión de listas con condicionales if

Si queremos obtener todos los números pares de una lista:

```
pares = [num for num in [1,2,3,4,5,6] if num % 2 == 0]
print(pares)
```

Esto es para enseñaros el for con condicionales, si queremos sacar los pares, mejor usar range():

```
for num in range(0,7,2):
    print(num)
```

1.3 Estructuras iterativas WHILE

Se usa cuando no sabemos cuántas veces se repetirá el bloque, pero depende de una condición que puede cambiar.
Ejemplo básico:

```
contador = 1
while contador <= 5:
    print("Contador:", contador)
    contador += 1
```

Ejercicio 4: Haz que el programa elija un número secreto entre 1 y 10. El usuario deberá adivinarlo y el programa le dirá si ha acertado o si el número secreto es más bajo o alto del indicado. Para importar el módulo de números aleatorios, usaremos import random e indicaremos mediante randint que queremos un número aleatorio entre 1 y 10.

```
import random


secreto = random.randint(1, 10)
```

choice() es una función del módulo random que elige un elemento al azar de una secuencia no vacía (por ejemplo, una lista, tupla o cadena).
Sintaxis:

```
import random
random.choice(secuencia)
```

• secuencia → es una lista, tupla, cadena, o cualquier tipo que se pueda indexar (como un rango).
• Devuelve un único elemento aleatorio de esa secuencia.
Ejemplo:

```
import random


colores = ["rojo", "verde", "azul", "amarillo"]
color_aleatorio = random.choice(colores)


print(color_aleatorio)
```

Cada vez que lo ejecutes, devolverá uno de los colores al azar.
Funciona tanto con cadena como con tuplas o rangos:

```
import random


letra = random.choice("Python")
print(letra)
```

```
import random
numeros = (10, 20, 30, 40)
print(random.choice(numeros))  # Elige uno al azar


print(random.choice(range(1, 11)))  # Número entre 1 y 10
```

Si queremos crear un bucle infinito podemos hacerlo con while true y para salir de dicho bucle usaremos break:

```
while True:
    numero = int(input("Dime un número "))
    if numero == 5:
        print("Salgo del bucle")
        break
    else:
        print("Continuo en el bucle")
print("Ya he salido del bucle")
```

Como podemos ver “break” modifica el comportamiento normal de los bucles, los interrumpe.

```
for letra in "Python":
    if letra == "h":
        break
    print(letra)
# Muestra: P, y, t
```

En este ejemplo, no recorre todas las letras de la palabra “Python”, el bucle se interrumpe al encontrar la letra “h”.

La instrucción “continue” salta una iteración:

```
for num in range(1, 6):
    if num == 3:
        continue
    print(num)
# Muestra: 1, 2, 4, 5
```

Aquí podemos ver que print(num) no se ejecuta cuando “num” vale 3, ya que salta la iteración actual y no sigue ejecutando lo que haya después de continue en el bucle for.

La instrucción “pass” se usa como marcador de posición cuando aún no sabemos qué poner en el bloque.

```
for i in range(5):
    pass  # Aquí escribiré el código más adelante
```

Ejercicio 5
Pide una palabra al usuario y muestra cuántas letras tiene, pero sin contar las vocales.

Else en el while
En un bucle while también podemos usar else de la siguiente forma:

```
contador = 0
while contador < 5:
    print(f"Contador: {contador}")
    contador += 1
```

```
else:
    print("Bucle terminado.")
```

Pero pensaras, para que vale el else, porque si lo quito sale el mismo resultado:

```
contador = 0
while contador < 5:
    print(f"Contador: {contador}")
    contador += 1
print("Bucle terminado.")
```

Tienes razón, sale el mismo resultado, el único caso en el que no se ejecuta el else es cuando usamos un break dentro del while, ya que el "else" se ejecuta solo cuando deje de cumplirse la condición del while y nosotros forzamos la salida con el break, pero la condición del while sigue cumpliéndose, solo sale porque se lo decimos con el break.

```
contador = 0
while contador < 5:
    print(f"Contador: {contador}")
    contador += 1
    break
else:
    print("Bucle terminado.")
```

En este caso no se ejecuta el contenido del "else".

Ejercicios FOR

Ejercicio 1

Imprimir números pares. Imprime todos los números pares del 2 al 20 (inclusive) usando un bucle for.

Ejercicio 2

Calcular la media de una lista. Dada la siguiente lista de números:
numeros = [10, 20, 30, 40, 50]. Calcula la media de los números usando un bucle for.

Ejercicio 3

Buscar el máximo de una lista. Dada la siguiente lista de números:
numeros = [15, 5, 25, 10, 20]
Encuentra el número máximo en la lista usando un bucle for.

Ejercicio 4

Filtrar cadenas por longitud. Dada la siguiente lista de palabras:
palabras = ["casa", "arbol", "sol", "elefante", "luna"]
Crea una nueva lista que contenga solo las palabras con más de 5 letras usando un bucle for y list comprehension.

Ejercicio 5

Contar palabras que empiezan con una letra. Dada la siguiente lista de palabras:
palabras = ["casa", "arbol", "sol", "elefante", "luna", "coche"]
Pide al usuario que introduzca una letra. Cuenta cuántas palabras en la lista empiezan con esa letra (sin diferenciar mayúsculas/minúsculas).

Ejercicios RANGE

Ejercicio 1

Imprimir números del 1 al 10. Imprime los números del 1 al 10 (inclusive) usando un bucle for y range().

Ejercicio 2

Imprimir números impares del 1 al 20. Imprime todos los números impares entre 1 y 20 (inclusive) usando un bucle for y range().

Ejercicio 3

Imprimir múltiplos de 5. Imprime los múltiplos de 5 desde 5 hasta 50 (inclusive) usando un bucle for y range().

Ejercicio 4

Imprimir números en orden inverso. Imprime los números del 10 al 1 (inclusive) en orden inverso usando un bucle for y range().

Ejercicio 5

Suma de números en un rango. Calcula la suma de los números del 1 al 100 (inclusive) usando un bucle for y range().

1.4 Sentencia condicional MATCH-CASE

Esta sentencia (llamada patter matching) se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

En su versión más simple, el "pattern matching" permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias "if" encadenadas.

Veamos esta aproximación mediante un ejemplo:

```
dia = 'lunes'
match dia:
    case 'lunes':
        print('Tengo que hacer la compra')
    case 'martes':
        print('Revisar el trastero')
    case 'miércoles':
        print('Descanso')
```

Si se quiere controlar cualquier otro valor que no se haya definido en las sentencias "case" se usa el guion bajo " _ ":

```
dia = 'lunes'
match dia:
    case 'lunes':
        print('Tengo que hacer la compra')
    case 'martes':
        print('Revisar el trastero')
    case 'miércoles':
        print('Descanso')
    case _:
        print('Dia sin planificar')
```

Match con patrones compuestos

El pattern matching permite **comparar estructuras completas de listas**, incluso si están anidadas o tienen diferentes longitudes.

```
match numeros:
    case [1, 2, 3]:
```

```
        print("Coincide con la lista [1, 2, 3]")
    case _:
        print("No coincide")
```

Salida: Coincide con la lista [1, 2, 3]

Se pueden **extraer elementos** de una lista directamente dentro del patrón:

```
datos = [10, 20, 30]


match datos:
    case [a, b, c]:
        print(f"Valores: {a}, {b}, {c}")
```

Salida: Valores: 10, 20, 30

Patrones con longitud variable

El operador * se puede usar para capturar el resto de los elementos de una lista.

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


match valores:
    case [x, y, *resto]:
        print(f"Primeros: {x}, {y}")
        print(f"Resto: {resto}")
```

Salida: Primeros: 1, 2
Resto: [3, 4, 5]

Otro ejemplo:

```
lista = ["Juan", "Antonio", "Ana"]
match lista:
    case [a]:
        print(f"Hola {a}")
    case [a, b]:
        print(f"Hola {a} y {b}")
    case [a, *b]:
        print(f"Hola {a} y los demás: {', '.join(b)}")


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

Salida: Hola Juan y los demás: Antonio, Ana

Ignorar elementos con _

Si no te interesa algún valor, puedes ignorarlo:

```
punto = [5, 10, 15]


match punto:
    case [x, _, z]:
```

```
        print(f"Coordenadas: x={x}, z={z}")
```

Salida: Coordenadas: x=5, z=15

Listas de listas

También puedes usar match con listas anidadas, como matrices o colecciones bidimensionales.

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


match matriz:
    case [[1, 2], [3, 4]]:
        print("Coincide con la matriz [[1, 2], [3, 4]]")
    case [[a, b], [c, d]]:
        print(f"Elementos: {a}, {b}, {c}, {d}")
```

Salida: Coincide con la matriz [[1, 2], [3, 4]]

Aquí podemos ver que aunque se cumple la condición en ambos “case”, coge el primero.

Lista de tuplas

También es posible usar match sobre estructuras de listas que contienen tuplas, como pares (clave, valor).

```
pares = [(1, "uno"), (2, "dos"), (3, "tres")]


match pares:
    case [(1, "uno"), *resto]:
        print("Empieza con (1, 'uno')")
        print(f"Resto: {resto}")
    case _:
        print("No coincide")
```

Salida: Empieza con (1, 'uno')
Resto: [(2, 'dos'), (3, 'tres')]

Se puede indicar una coincidencia parcial de tuplas dentro de una lista.

```
datos = [("A", 10), ("B", 20)]


match datos:
    case [(_, 10), (_, 20)]:
        print("Lista con dos tuplas con valores 10 y 20")
```

Salida: Lista con dos tuplas con valores 10 y 20.

Match con condiciones

Podemos añadir condiciones adicionales a un patrón usando if después de case.

```
numeros = [5, 10, 15]


match numeros:
```

```
    case [x, y, z] if sum(numeros) > 20:
        print(f"Suma mayor que 20. Elementos: {x}, {y}, {z}")
    case _:
        print("No cumple la condición")
```

Salida: Suma mayor que 20. Elementos: 5, 10, 15

Ejercicios match

Ejercicio 1

Dada una lista, muestra un mensaje diferente si está vacía, si tiene un solo elemento, si tiene 2 elementos o si tiene más de dos.

Ejercicio 2

Dada una lista de listas con dos coordenadas, determina si están en el mismo eje (x o y).
puntos = [[2, 5], [2, 9]]

Ejercicio 6

Crea un programa que pida al usuario un número entre 1 y 10 y que:

1. Compruebe que el número está en ese rango (si no, pide otro).
2. Muestre su tabla de multiplicar.
3. Pregunte si desea repetir con otro número (s/n).

1.5 Operador morsa

El operador morsa (:=), también llamado operador de asignación en expresión, permite asignar un valor a una variable al mismo tiempo que se evalúa dentro de una expresión (por ejemplo, dentro de un if, while o una comprensión de lista).
Su nombre proviene del símbolo := que se parece a la cara de una morsa.

Antes de Python 3.8, si queríamos usar un valor dentro de una condición y conservarlo en una variable, teníamos que escribirlo dos veces.
Por ejemplo:

```
linea = input("Escribe algo: ")
while linea != "salir":
    print("Has escrito:", linea)
    linea = input("Escribe algo: ")
```

Aquí input() se llama dos veces: una para comparar, otra para guardar el valor.

Con el operador morsa, se puede hacer en una sola línea:

```
while (linea := input("Escribe algo: ")) != "salir":
    print("Has escrito:", linea)
```

Cómo funciona

variable := expresión

- Evalúa primero la expresión (a la derecha).
- Asigna su resultado a la variable (a la izquierda).
- Devuelve el valor asignado, por lo que puede usarse dentro de otra expresión.

Ejemplo de uso en un while:

```
secreto = random.randint(1, 10)
while (intento := int(input("Adivina el número (1–10): "))) != secreto:
    if intento < secreto:
        print("Demasiado bajo.")
    else:
        print("Demasiado alto.")
print("¡Correcto!")
```

Aquí el operador morsa permite leer el número y compararlo en la misma línea del while.

Ejemplo de uso en un if:

```
if (nombre := input("Introduce tu nombre: ")):
    print(f"Hola, {nombre}!")
else:
    print("No has escrito nada.")
```

Si el usuario escribe algo, se guarda en la variable "nombre" y se evalúa como True.
Si está vacío, el if se evalúa como False.

Ejercicio 7

Crea un programa que pida números al usuario y calcule su suma. El programa termina cuando el usuario introduce un número vacío. Usa el operador morsa.

1.6 Try…except

Cuando un programa se ejecuta, pueden ocurrir errores que detienen su funcionamiento. En Python, estos errores se llaman excepciones (exceptions).
Ejemplo:

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

Si el usuario escribe una letra en lugar de un número, el programa muestra:

```
ValueError: invalid literal for int() with base 10: 'a'
```

Si no controlamos ese error, el programa se interrumpe abruptamente.
Para evitarlo, Python ofrece el bloque **try...except**.

```
try:
    # Código que podría causar un error
except:
    # Código que se ejecuta si ocurre un error
```

Ejemplo para controlar un error simple:

```
try:
    numero = int(input("Introduce un número: "))
    print(f"El doble es: {numero * 2}")
except:
    print("Debes introducir un número válido.")
```

Puedes capturar errores concretos para dar mensajes más útiles.
Algunos tipos de excepción comunes son:

Error	Descripción	Ejemplo

ValueError	Conversión de tipo inválida	int("abc")
ZeroDivisionError	División por cero	10 / 0
TypeError	Tipo de dato incompatible	"a" + 3
NameError	Variable no definida	print(x)
KeyboardInterrupt	El usuario interrumpe con Ctrl+C	(interrupción manual)

Ejemplo capturando tipos de error

```
try:
    a = int(input("Introduce el primer número: "))
    b = int(input("Introduce el segundo número: "))
    print(a / b)
except ValueError:
    print("Debes introducir solo números.")
except ZeroDivisionError:
    print("No puedes dividir entre cero.")
```

En este ejemplo, se manejan dos errores distintos con mensajes personalizados.

Bloques else y finally

El bloque try puede complementarse con else y finally:

```
try:
    # Código que puede fallar
except:
    # Se ejecuta si ocurre un error
else:
    # Se ejecuta si NO ocurre ningún error
finally:
    # Se ejecuta SIEMPRE (haya o no error)
```

Ejemplo:

```
try:
    n = int(input("Introduce un número: "))
except ValueError:
    print("Error: no es un número.")
else:
    print(f"El cuadrado es {n ** 2}.")
finally:
    print("Programa finalizado.")
```

A veces conviene ver el error exacto para depurar:

```
try:
    x = int("a")
except ValueError as e:
    print("Ha ocurrido un error:", e)
```

Ejercicio 8

Haz un programa que pida dos números y muestre su división.
Debe controlar que el usuario escriba números válidos y no divida entre cero.

2. Listas

¿Qué es una lista?

Una lista es una estructura de datos muy flexible que permite almacenar múltiples elementos, ya sean del mismo tipo o de tipos diferentes (números, cadenas, booleanos, otras listas, etc.).
Las listas en Python son dinámicas, es decir, su tamaño puede aumentar o disminuir durante la ejecución del programa.

Crear listas

Las listas se definen entre corchetes [ ], separando sus elementos con comas.

```
lista1 = [1, 2, 3, 4, 5]
lista2 = ["manzanas", "peras"]
lista3 = [1, "manzanas", 2, "peras"]
```

En Python es posible indicar el tipo esperado mediante anotaciones de tipo (type hints):

```
lista3: list[int] = [1, "manzanas", 2, "peras"]
```

En esta caso VSCode nos indica un error, ya que la lista contiene cadenas además de enteros.
También podemos tener listas dentro de listas (listas anidadas):

```
lista4 = [[1, 2], [3, 4]]
```

Acceso a los elementos

Los elementos de una lista se acceden por su índice, que empieza en 0.

```
print(lista2[0])  # Muestra: manzanas
```

En Python podemos acceder a los índices de forma negativa comenzando por el final.

```
print(lista2[-1])  # Muestra: peras
print(lista2[-2])  # Muestra: manzanas
```

En listas anidadas:

```
print(lista4[1][0])  # Muestra: 3
```

Primero accedemos a la posición 1 que tendría [3,4] y luego de esta lista indicamos que queremos acceder a la primera posición, por lo que devolvería “3”

Rebanado de listas (slicing)

El rebanado permite extraer partes de una lista creando una nueva.
La sintaxis es:
lista[inicio:fin:paso]
Ejemplo:

```
lista = [1, 2, 3, 4, 5]
print(lista[1:4])  # [2, 3, 4]
print(lista[:3])   # [1, 2, 3] — los tres primeros
print(lista[3:])   # [4, 5] — desde el índice 3 hasta el final
```

```
print(lista[:])    # copia completa
```

Hay que tener en cuenta que si queremos coger la lista desde el índice 1 hasta el índice 4 éste último es no inclusive.

También se puede hacer una copia de una lista usando “copy”:

```
copia_2 = original.copy()
```

Podemos indicar saltos. Por defecto, rebana la lista de uno en uno, pero podemos indicar que vaya haciendo saltos:

```
lista = [1,2,3,4,5,6,7,8,9,10]
print(lista[::2]) #[1,3,5,7] Devuelve índices impares
```

Para invertir una lista:

```
print(lista[::-1])
```

Empieza desde el final, por lo que devuelve [10,9,8,7,6,5,4,3,2,1]

Modificar una lista

La mayoría de las listas se crean para ser dinámicas, es decir, que sus elementos crecerán o decrecerán en el tiempo. Si queremos modificar un elemento de una lista, hay que acceder al elemento que queremos modificar y asignarle el nuevo valor. Ejemplo:

```
lista = ["manzana", "pera", "plátano"]
lista[1] = "naranja"
print(lista)  # ['manzana', 'naranja', 'plátano']
```

Añadir elementos

La forma más fácil de añadir elementos a una lista es añadiéndolo al final de la misma usando la función “append”.

```
lista = ["manzana", "pera"]
lista.append("kiwi")
print(lista)  # ['manzana', 'pera', 'kiwi']
```

También podemos añadir elementos a una lista de esta forma:

```
lista = lista + [3,4,5] #Concatena a lista 3,4,5
#O también:
lista += [3,4,5]
```

Se pueden agregar varios elementos (concatena otra lista) utilizando extend.

```
lista.extend(["melón", "uva"])
print(lista)  # ['manzana', 'pera', 'kiwi', 'melón', 'uva']
```

o también

```
lista.extend([3,4,5,6])
```

Insertar elementos dentro de una lista

Si queremos añadir un elemento dentro de una lista en una posición específica, usar el método “insert”, que tiene 2 parámetros, el primero es el índice donde añadir el elemento y es segundo es el contenido a añadir en el índice indicado. Ejemplo:

```
lista = ["manzana", "pera", "plátano"]
lista.insert(1, "kiwi")
print(lista)  # ['manzana', 'kiwi', 'pera', 'plátano']
```

Hay que tener en cuenta que no modifica el elemento del índice 1, sino que lo inserta en el índice 1 y el contenido que antes estaba en el índice 1 pasa al índice 2. Ejemplo:

Eliminar elementos de una lista

Usando la función “remove”, se elimina el primer elemento que coincida con el valor.

```
frutas = ["pera", "manzana", "pera"]
frutas.remove("pera")
print(frutas)  # ['manzana', 'pera']
```

Usando la función “pop”, se elimina un elemento por índice (por defecto, el último) y lo devuelve.

```
numeros = [10, 20, 30, 40]
ultimo = numeros.pop()
print(ultimo)    # 40
print(numeros)   # [10, 20, 30]
```

También podemos eliminar un índice concreto:

```
numeros.pop(1)  # elimina el elemento con índice 1
```

También existe una forma (un poco más fea) de eliminar un elemento o rango de elementos.

```
del numeros[-1]     # elimina el último
del numeros[1:3]    # elimina del índice 1 al 2
```

La ventaja de usar “del” es que permite eliminar un rango de elementos de la lista:

Si queremos eliminar todos los elementos de una lista.

```
numeros.clear()
print(numeros)  # []
```

Ordenar listas

Para ordenar listas veremos 3 formas:

- Usando la función sort, que ordena la lista modificándola.

```
numeros = [3, 10, 4, 76, 103, 56]
numeros.sort()
print(numeros)  # [3, 4, 10, 56, 76, 103]
```

- Si usamos la función sorted, devuelve una nueva lista ordenada sin modificar la original.

```
numeros = [3, 10, 4, 76, 103, 56]
numeros_ordenados = sorted(numeros)
print(numeros_ordenados)
```

Si queremos ordenar una lista de palabras, a veces tenemos un problema cuando hay mayúsculas y minúsculas, ya que, por defecto, las mayúsculas se ordenan antes que las minúsculas. Para evitarlo:

```
frutas = ["manzana", "Pera", "kiwi", "limón", "Melocotón"]
frutas.sort(key=str.lower)
print(frutas)
```

frutas.sort(key=str.lower) #Indicamos la comparación con la cadena y en este caso la transforma en minúsculas.

Se puede ordenar una lista al revés de esta forma:

```
coches = ['bmw', 'audi', 'toyota', 'subaru']
coches.sort(reverse=True)
```

No confundir con la función reverse, que lo que hace es darle la vuelta una lista:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)
cars.reverse()
print(cars) #['subaru', 'toyota', 'audi', 'bmw']
```

Ejercicios listas

Ejercicio 1

El mensaje secreto. Dada la siguiente lista:
mensaje = ["C", "o", "d", "i", "g", "o", " ", "s", "e", "c", "r", "e", "t", "o"]
Utilizando slicing y concatenación, crea una nueva lista que contenga solo el mensaje "secreto".

Ejercicio 2

Intercambio de posiciones. Dada la siguiente lista:
numeros = [10, 20, 30, 40, 50]
Intercambia la primera y la última posición utilizando solo asignación por índice.

Ejercicio 3

El sándwich de listas. Dadas las siguientes listas:
pan = ["pan arriba"]
ingredientes = ["jamón", "queso", "tomate"]
pan_abajo = ["pan abajo"]
Crea una lista llamada sandwich que contenga el pan de arriba, los ingredientes y el pan de abajo, en ese orden.

Ejercicio 4

Duplicando la lista. Dada una lista:

lista = [1, 2, 3]
Crea una nueva lista que contenga los elementos de la lista original duplicados.
Ejemplo: [1, 2, 3] -> [1, 2, 3, 1, 2, 3]

Ejercicio 5
Extrayendo el centro. Dada una lista con un número impar de elementos, extrae el elemento que se encuentra en el centro de la lista utilizando slicing.
Ejemplo: lista = [10, 20, 30, 40, 50] -> El centro es 30

Ejercicio 6
Reversa parcial. Dada una lista, invierte solo la primera mitad de la lista (utilizando slicing y concatenación).
Ejemplo: lista = [1, 2, 3, 4, 5, 6] -> Resultado: [3, 2, 1, 4, 5, 6]

Otras operaciones útiles
Podemos contar cuántas veces aparece un elemento en una lista:

```
frutas = ["limón", "pera", "limón", "kiwi"]
print(frutas.count("limón"))  # 2
```

También podemos obtener el mayor o menor elemento de una lista usando “min” y “max”:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
print(f"El numero mínimo es: {min(numeros)}")
print(f"El numero máximo es: {max(numeros)}")
```

Si queremos comprobar si hay cierto elemento en la lista.

```
print("pera" in frutas)      # True
print("naranja" not in frutas)  # True
```

Para sumar todos los elementos de una lista, puede hacerse usando “sum”.

```
lista = [1, 2, 3]
print(sum(lista))  # Salida: 6
```

round(número, decimales). Sirve para redondear un valor float a una cantidad fija de decimales.

```
numero = 14.54756
redondeo = round(numero, 2)
print(f"Número redondeado: {redondeo}")
#14.55
```

abs(número). Obtiene el valor absoluto de un número float o int. Esto es, su valor positivo si es negativo.

```
numero = -5
print(abs(numero))
```

startswith(). Comprueba cómo empieza una cadena. Sintaxis:
cadena.startswith(prefijo[, inicio[, fin]])

- prefijo: texto que quieres comprobar si está al principio.
- inicio (opcional): posición desde la que comienza la búsqueda.
- fin (opcional): posición hasta donde se busca.

Devuelve True si la cadena empieza con el texto indicado, o False si no.

```
frase = "Python es un lenguaje de programación."


print(frase.startswith("Python"))   # True
print(frase.startswith("python"))   # False (distingue mayúsculas)
print(frase.startswith("Py"))       # True
print(frase.startswith("Java"))     # False
```

También puedes indicar posiciones específicas:

```
print(frase.startswith("lenguaje", 12))  # True (empieza en índice 10)
```

Las comparaciones son sensibles a mayúsculas y minúsculas. Si quieres que no lo sean, convierte la cadena a minúsculas:

```
frase.lower().startswith("python")
```

endswith(). Comprueba cómo termina una cadena. Sintaxis:
cadena.endswith(sufijo[, inicio[, fin]])

- sufijo → texto que quieres comprobar si está al final.
- inicio y fin → opcionales, igual que en startswith().

Devuelve True si la cadena termina con el texto indicado, o False si no.

```
archivo = "informe_final.pdf"


print(archivo.endswith(".pdf"))     # True
print(archivo.endswith(".txt"))     # False
print(archivo.endswith("final"))    # False (no es el final exacto)
```

También puedes usar tuplas de posibles sufijos:

```
print(archivo.endswith((".pdf", ".docx", ".txt")))  # True
```

El método .join() se usa para unir elementos de una lista (u otro iterable) en una única cadena de texto (string), separándolos por el texto que tú elijas.
La estructura general es: SEPARADOR.join(lista_de_cadenas)

```
pistas = ["La", "clave", "está", "en", "el", "mensaje"]
resultado = ' '.join(pistas)
print(resultado)
```

Une todos los elementos de la lista pistas, poniendo un espacio ' ' entre cada uno.

Cosas a tener en cuenta:
- Todos los elementos deben ser cadenas de texto (str). Si hay números (int, float…), dará error:

```
pistas = ["Nivel", 3, "superado"]
' '.join(pistas)  # Error: TypeError
```

- Si necesitas unir elementos no-cadena, primero conviértelos:

```
pistas = ["Nivel", 3, "superado"]
' '.join(str(p) for p in pistas)  # "Nivel 3 superado"
```

Si queremos imprimir un carácter N veces usar la sintaxis:

```
print("*" * 5)
```

Imprime 5 asteriscos.

Cuando usas {nombre_variable=} dentro de un f-string, Python imprime el nombre de la variable junto con su valor. Es como si hicieras esto manualmente:

```
pares=5
impares=4
print(f"pares={pares}, impares={impares}")
```

Es mucho más fácil, rápido y menos propenso a errores usar esto:

```
pares=5
impares=4
print(f"{pares=}, {impares=}")
```

Ejercicios métodos de listas

Ejercicio 1

Añadir y modificar elementos. Crea una lista con los números del 1 al 5.
Añade el número 6 al final usando append().
Inserta el número 10 en la posición 2 usando insert().
Modifica el primer elemento de la lista para que sea 0.

Ejercicio 2

Combinar y limpiar listas. Crea dos listas:
lista_a = [1, 2, 3]
lista_b = [4, 5, 6, 1, 2]
Extiende lista_a con lista_b usando extend().
Elimina la primera aparición del número 1 en lista_a usando remove().
Elimina el elemento en el índice 3 de lista_a usando pop(). Imprime el elemento eliminado.
Limpia completamente lista_b usando clear().

Ejercicio 3

Slicing y eliminación con “del”.
Crea una lista con los números del 1 al 10.
Utiliza slicing y del para eliminar los elementos desde el índice 2 hasta el 5 (sin incluir el 5).
Imprime la lista resultante.

Ejercicio 4

Ordenar y contar. Crea una lista con los siguientes números: [5, 2, 8, 1, 9, 4, 2].
Ordena la lista de forma ascendente usando sort().
Cuenta cuántas veces aparece el número 2 en la lista usando count().
Comprueba si el número 7 está en la lista usando in.

Ejercicio 5

Copia vs. Referencia. Crea una lista llamada original con los números [1, 2, 3].
Crea una copia de la lista original llamada copia_1 usando slicing.
Crea otra copia llamada copia_2 usando copy().
Crea una referencia a la lista original llamada referencia.
Modifica el primer elemento de la lista referencia a 10.
Imprime las cuatro listas (original, copia_1, copia_2, referencia) y observa los cambios.

Ejercicio 6
Ordenar strings sin diferenciar mayúsculas y minúsculas.
Crea una lista con las siguientes cadenas: ["Manzana", "pera", "BANANA", "naranja"].
Ordena la lista sin diferenciar entre mayúsculas y minúsculas.

3. Tuplas

Una tupla es una colección ordenada e inmutable de elementos. Se parece mucho a una lista, pero no se puede modificar una vez creada (no se pueden añadir, eliminar ni cambiar elementos).
Las tuplas se utilizan cuando necesitas almacenar datos agrupados que no deben cambiar, por ejemplo: coordenadas, fechas, datos agrupados por posición, etc.

Cómo crear una tupla

La forma más común de crear una tupla es usando paréntesis y separando los elementos con comas:

```
tupla = (1, 2, 3)
```

También es válido crear una tupla sin paréntesis (Python la reconocerá si hay comas), aunque no es recomendable por claridad:

```
tupla = 1, 2, 3
```

Para crear una tupla de un solo elemento, es obligatorio poner una coma al final, ya que si no Python lo interpreta como un valor normal:

```
tupla_valida = (5,)   # Esto es una tupla
no_es_tupla = (5)     # Esto es un entero, no una tupla
```

Características de las tuplas

Ordenadas: Los elementos tienen un orden fijo.
Inmutables: No se pueden cambiar, añadir ni eliminar elementos.
Pueden contener varios tipos: Cadenas, números, listas, otras tuplas, etc.
Acceso por índice: Igual que en listas, empezando desde 0.
Pueden anidarse: Se pueden tener tuplas dentro de tuplas.

Acceso a los elementos de una tupla

Al igual que con las listas, podemos acceder a los elementos de una tupla usando su posición (índice).
Los índices en Python comienzan en 0, es decir, el primer elemento tiene índice 0, el segundo índice 1, y así sucesivamente.

```
tupla = ("rojo", "verde", "azul")
print(tupla[0])   # rojo
print(tupla[-1])  # azul (último elemento)
```

También podemos usar índices negativos para acceder desde el final:

```
print(colores[-1])  # 'azul' (último)
```

```
print(colores[-2])  # 'verde' (penúltimo)
```

Recorrer una tupla con un bucle

Aunque no se pueden modificar, sí se pueden recorrer las tuplas usando un bucle for. Esto permite leer y procesar sus elementos.

```
dias = ("lunes", "martes", "miércoles")
for dia in dias:
    print(dia)
```

Desempaquetar tuplas (asignación múltiple)

Una característica útil de las tuplas es que se pueden desempaquetar, es decir, asignar sus valores directamente a varias variables. Esto es muy práctico para trabajar con funciones que devuelven varios valores.

```
persona = ("Lucía", 28)
nombre, edad = persona
print(nombre)  # Lucía
print(edad)    # 28
```

En general se puede desempaquetar cualquier objeto de tipo secuencia. Podemos utilizar el operador '*' para que asigne todo lo que no "quepa" en las otras variables.

```
lista = (1, 2, 3, 4)
primero, *resto, ultimo = lista
print(primero)   # 1
print(resto)     # [2, 3]
print(ultimo)    # 4
```

Si vemos el tipo de resto, veremos que no es una tupla es una lista <class 'list'>.
Las tuplas, junto con el desempaquetado, son útiles para realizar return de más de un valor.

```
def suma_resta(op1, op2):
    suma = op1 + op2
    resta = op1 - op2
    return (suma, resta)  # Devuelve una tupla con dos valores


suma, resta = suma_resta(10, 5)
print(f"Suma: {suma}, Resta: {resta}")
```

Métodos disponibles para tuplas

Como las tuplas son inmutables, tienen muy pocos métodos, pero existen dos que pueden resultar útiles:

tupla.count(x): Cuenta cuántas veces aparece el valor x en la tupla.
tupla.index(x): Devuelve la posición (índice) de la primera aparición de x.

```
numeros = (1, 2, 2, 3, 4)
print(numeros.count(2))   # Resultado: 2 (el 2 aparece dos veces)
print(numeros.index(3))   # Resultado: 3 (el 3 está en la posición 3)
```

Ejercicios tuplas
Ejercicio 1
Crea una tupla con tres nombres de personas. Accede al segundo nombre y muéstralo por pantalla.

Ejercicio 2
Crea una tupla con los números del 1 al 5. Usa un bucle for para mostrar todos sus elementos uno por uno.

Ejercicio 3
Dada la siguiente tupla: datos = ("Juan", 25, "España")
Desempaquétala en tres variables y muestra un mensaje como:
Juan tiene 25 años y vive en España.

Ejercicio 4
Dada la tupla siguiente: numeros = (4, 2, 4, 1, 4, 3, 2)
Cuenta cuántas veces aparece el número 4 y muestra el resultado por pantalla.

Ejercicio 5
Crea una función "contar_pares_impares" que reciba por parámetro:
- lista: una lista de números enteros
La función debe devolver 2 enteros (una tupla). El primer entero indica la cantidad de números pares y el segundo entero indica la cantidad de números impares de la lista recibida por parámetro.

4. Funciones

Las funciones en Python son bloques de código reutilizables y parametrizables que nos permiten encapsular lógica para realizar tareas específicas. Esto mejora la organización y la mantenibilidad de nuestro código.

Definición de una función

La sintaxis básica para definir una función es:

```
def nombre_de_la_funcion(parametro1, parametro2, ...):
    """
    Documentación de la función
    """
    # Cuerpo de la función
    return valor_de_retorno  # opcional
```

Podemos tener una función sin parámetros:

```
def saludar():
    print("¡Hola, mundo!")
```

O una función con un parámetro, que puede ser reutilizable y llamarse varias veces:

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


saludar("Pepe")
```

```
saludar("Ana")
```

Como podemos ver en este ejemplo, la simple definición de una función no la ejecuta, hay que llamarla. También indicar que primero se define una función y luego se utiliza, si llamamos a la función saludar antes de hacer def saludar(nombre): nos dará error porque no conoce la existencia de la función, esto es porque Python es interpretado y va ejecutando línea a línea y cuando lee la línea saludar("Pepe") aún no ha llegado a def saludar(nombre) y por eso no sabe de su existencia.
Una función puede devolver un valor, que habrá que guardarlo en una variable.

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

Podemos llamar a esta función y recuperando la suma así:

```
resultado = sumar(3, 5)
print("La suma es:", resultado)
```

Documentar funciones en Python

La documentación de funciones es fundamental para mejorar la legibilidad y facilitar el mantenimiento del código. En Python, se utilizan los docstrings para describir qué hace una función, sus parámetros y su valor de retorno.

```
def sumar(a,b):
    """
    Esta función suma dos números y devuelve el resultado.
    """
    suma = a + b
    return suma
```

Ahora al dejar el ratón encima del nombre de la función, abajo te aparece la descripción de para qué vale la función.

Parámetros por defecto

Se puede indicar un valor por defecto a un parámetro de entrada de la función, de modo que si no se le pasa dicho parámetro, coge el valor por defecto:

```
def sumar(a,b=3):
    """
    Esta función suma dos números y devuelve el resultado.
    """
    suma = a + b
    return suma


resultado = sumar(3)
print("La suma es:", resultado)
```

Aquí llamamos a la función suma con el parámetro 3 y la función recibe el parámetro 3 en "a" y el valor por defecto que es 3 en "b".

Parámetros opcionales

Podemos usar el valor por defecto None como valor defecto, para indicar que el parámetro es opcional, por lo que podemos llamar a la función especificando dicho parámetro o no.

```
def saluda(nombre = None):
    if nombre is None:
        print("Hola, ¿cómo te llamas?")
    else:
        print(f"Hola, {nombre}, ¡bienvenido/a!")
# Llamo a la función sin argumentos
saluda()
#Llamo a la función con un argumento
saluda("Pepe")
```

Argumentos por clave o parámetros nombrados

Hasta ahora, hemos visto que una función tiene los parámetros posicionales, si llamo a la función con 3 parámetros, el primer parámetro siempre estará asociado al primer parámetro de la función.

```
def describe_persona(nombre, edad, ciudad): {
    print(f"Nombre: {nombre} | Edad: {edad} | Ciudad: {ciudad}")
}


describe_persona("Pepe", 30, "Madrid")
```

Pero si cambio el orden de los argumentos.

```
describe_persona(30, "Madrid" ,"Pepe")
```

Muestra que nombre s 30, edad “Madrid” y ciudad “Pepe”. Para evitar esto usamos los argumentos por posición.

```
describe_persona(edad=30, ciudad="Madrid", nombre="Pepe")
```

Como puedes ver, da igual dónde esté valor, ya ahora que está asociado a un nombre.

Argumentos de longitud variable (*args)

A veces necesitamos que la cantidad de argumentos de una función sea variable:

```
def suma_numeros(*args):
    suma = 0
    for numero in args:
        suma += numero
    return suma
print(suma_numeros(1, 2, 3, 4, 5))
print(suma_numeros(10, 20, 30))
```

Argumentos de clave-valor variable (kwargs)**

En Python, **kwargs permite que una función reciba un número **variable de argumentos con nombre (clave=valor)**. Su nombre viene de *“keyword arguments”* (argumentos con palabra clave), y se utiliza cuando no se sabe de antemano cuántos argumentos nombrados se pasarán a la función

¿Para qué sirve **kwargs?

- Para permitir llamadas a funciones con cualquier número de argumentos nombrados.
- Para escribir funciones más flexibles y reutilizables.
- Se recibe en la función como un **diccionario**, donde las claves son los nombres de los argumentos y los valores sus respectivos contenidos.

Sintaxis:

```
def funcion(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave} = {valor}")
```

Ejemplo:

```
def mostrar_info(**kwargs):
    """
    Muestra información recibida como argumentos nombrados.
    """
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")


mostrar_info(nombre="Ana", edad=30, ciudad="Madrid")
mostrar_info(nombre="Ana", apellido="Lopez", suscrito=True)
```

Ejercicio 12

Crear una función sumaproducto que reciba por parámetro

- Operación: una cadena que indica si se realiza la suma o el producto.
- Un número variable de parámetros que se considerarán como números. Estos parámetros serán los operandos.

La función devuelve la suma o el producto de todos los números recibidos por parámetros. Si la operación no es ni "suma" ni "producto" se imprime un mensaje de error y devuelve None.

Ejercicios

¿Qué devolvería la siguiente función si se llama con find_carnivore_eggs_indices([3, 6, 7, 8, 5, 10], 16)?

```
def find_carnivore_eggs_indices(eggs_list, target_sum):
    """
    Encuentra los índices de los dos primeros huevos de carnívoros (números pares)
    cuya suma sea igual al objetivo.
    """
    # Filtrar solo los huevos de carnívoros (números pares) con sus índices
    carnivore_eggs = [(i, eggs) for i, eggs in enumerate(eggs_list) if eggs % 2 == 0]


    # Buscar el primer par que sume el objetivo
    for i in range(len(carnivore_eggs)):
        for j in range(i + 1, len(carnivore_eggs)):
```

```
    idx1, eggs1 = carnivore_eggs[i]
    idx2, eggs2 = carnivore_eggs[j]
    if eggs1 + eggs2 == target_sum:
      return [idx1, idx2]

  return None
```

¿Qué imprimirá el siguiente código?

```
def count_vowels_consonants(text):
  text = text.lower()
  vowels = "aeiou"
  consonants = "bcdfghjklmnpqrstvwxyz"

  vowel_count = 0
  consonant_count = 0

  for char in text:
    if char in vowels:
      vowel_count += 1
    elif char in consonants:
      consonant_count += 1

  return vowel_count, consonant_count

v, c = count_vowels_consonants("Python")
print(f"Vocales: {v}, Consonantes: {c}")
```

Comprobador de palíndromos

```
def is_palindrome(text):
  text = text.lower()
  return text == text[::-1]

print(is_palindrome("Python"))
```

Como sabemos, una variable que apunte a una lista, lo que tiene dicha variable es una referencia a la posición de memoria que tiene dicha lista, por lo que si le pasamos una lista como parámetro de entrada a una función, cualquier cambio que se realice en la lista dentro de la función, modifica la lista original.

```
def print_models(modelos_a_imprimir, modelos_completados):
    while modelos_a_imprimir:
        diseño_actual = modelos_a_imprimir.pop()
```

```
        print(f"Imprimiendo modelo: {diseño_actual}")
        modelos_completados.append(diseño_actual)

def mostrar_modelos_completados(modelos_completados):
    print("\nLos siguientes modelos se han impreso:")
    for completed_model in modelos_completados:
        print(completed_model)

modelos_a_imprimir = ['phone case', 'robot pendant', 'dodecahedron']
modelos_completados = []
print_models(modelos_a_imprimir, modelos_completados)
mostrar_modelos_completados(modelos_completados)
```

Aquí podemos ver cómo mostrar_modelos_completados modifica la lista modelos_completados que se le pasa como parámetro.

Para prevenir que una función modifique una lista debemos pasarle a la función una copia de la lista, no la referencia a la lista original, por lo que cualquier cambio a la lista dentro de la función se realiza sobre la copia sin afectar a la lista original.

```
function_name(list_name[:])
```

Como ya vimos [:] realiza una copia de la lista.
Nuestro ejemplo anterior quedaría así:

```
def mostrar_modelos_completados(modelos_completados[:]):
```

Almacenando funciones en módulos

En Python, un **módulo** es un archivo con extensión .py que contiene funciones, clases o variables que puedes **importar y reutilizar** en otros programas.
Imagina que tienes un archivo saludos.py con el siguiente contenido:

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

def despedir(nombre):
    print(f"Adiós, {nombre}...")
```

Este archivo define dos funciones (saludar y despedir), y podemos reutilizarlas desde otros scripts **importando el módulo**.

¿Por qué usar módulos?
- **Organización del código** — separa las funciones del programa principal.
- **Reutilización** — puedes usar las mismas funciones en varios proyectos.
- **Colaboración** — otros programadores pueden usar tus módulos sin ver todo tu código.
- **Legibilidad** — tu programa principal se enfoca en la lógica general, no en los detalles.

Importar un módulo completo
Para importar un módulo entero, usa la instrucción:

import nombre_modulo

Ejemplo:

Archivo pizza.py:

```
def make_pizza(size, *toppings):
    """Muestra los ingredientes de una pizza."""
    print(f"\nPreparando una pizza de {size} cm con:")
    for topping in toppings:
        print(f"- {topping}")
```

Archivo hacer_pizza.py:

```
import pizza


pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'champiñones', 'queso extra')
```

Cuando haces import pizza, Python busca el archivo pizza.py, lo carga en memoria y lo pone a disposición del programa.
Ten en cuenta que primero hay que poner el nombre del módulo y luego la función para poder usarla.

Importar funciones específicas

Si solo necesitas una o varias funciones, puedes importarlas directamente:

from nombre_modulo import función

Ejemplo:

```
from pizza import make_pizza


make_pizza(16, 'pepperoni')
make_pizza(12, 'queso', 'tomate')
```

Aquí no necesitas escribir pizza.make_pizza(), porque la función se importa directamente.

También puedes importar varias funciones separadas por comas:

```
from pizza import make_pizza, otra_funcion
```

Usar alias con as

A veces una función tiene un nombre largo, o puede haber conflicto de nombres. Puedes darle un alias con as:

```
from pizza import make_pizza as mp


mp(16, 'jamón', 'queso')
```

Ahora mp() es un alias que hace referencia a make_pizza().

Alias para módulos

También puedes acortar el nombre del módulo completo:

```
import pizza as p


p.make_pizza(12, 'aceitunas', 'queso azul')
```

Usar un alias hace que el código sea más conciso y claro, sobre todo con módulos largos:

```
import matplotlib.pyplot as plt
import numpy as np
```

Impotar todos los elementos de un modulo

Si quieres importar todo el contenido de un módulo, usa el asterisco *:

from pizza import *

Esto hace que todas las funciones estén disponibles directamente:

```
make_pizza(12, 'queso', 'tomate')
```

Desventaja: puede causar conflictos de nombres.
Si el módulo tiene una función con el mismo nombre que otra de tu código, una sobrescribirá a la otra.
Por eso no se recomienda en programas grandes.

5. Diccionarios

Un diccionario es una estructura de datos en Python que permite almacenar información en forma de pares clave-valor. A diferencia de las listas, donde los elementos se acceden por posición (índice), en los diccionarios se accede mediante una clave única, lo que permite organizar y buscar la información de forma rápida y eficiente.
Cada elemento del diccionario tiene la forma:

clave: valor

Las claves de un diccionario deben ser **inmutables**, es decir, que su valor no puede ser cambiado después de que la clave haya sido creada, mientras que los valores pueden ser de cualquier tipo. Los tipos que cumplen este requisito son:
- Enteros(int) y números de punto flotante(float).
- Cadenas(str).
- Tuplas: siempre que no contenga elementos mutables.

Los diccionarios son muy utilizados en programación para representar:

- Información de una persona (nombre, edad, dirección…)
- Configuraciones o parámetros
- Datos que necesitan ser accedidos por nombre

Sintaxis básica:

```
diccionario = {
   "nombre": "Ana",
   "edad": 25,
   "ciudad": "Madrid"
}
```

• Claves: deben ser únicas y de tipo inmutable (como str, int, tuple…).
• Valores: pueden ser de cualquier tipo.

Diccionarios anidados

Un valor de un diccionario puede ser, a su vez, otro diccionario o una lista:

```
persona = {
  "nombre": "Pepe",
  "edad": 25,
  "es_estudiante": True,
  "calificaciones": [7, 8, 9],
  "socials": {
    "twitter": "@pepe",
    "instagram": "@pepe",
    "facebook": "pepe"
  }
}
```

Métodos de clase diccionario:

- keys(): Devuelve una lista de todas las claves del diccionario.
- values(): Devuelve una lista de todos los valores del diccionario.
- items(): Devuelve una lista de tuplas con los pares clave-valor.
- get(key, default): Devuelve el valor correspondiente a la clave, o un valor por defecto si la clave no existe.
- pop(key): Elimina el elemento con la clave especificada y devuelve su valor.
- popitem(): Elimina y retorna el par (clave, valor) que ha sido el último en ser añadido (LIFO, last-in, first-out).
- update(dict2): Actualiza el diccionario con los elementos de otro diccionario.
- clear(): Elimina todos los elementos del diccionario.
- copy(): Crea una copia del diccionario en otra dirección de memoria.
- setdefault(clave, default=None): Si la clave existe, devuelve su valor. Si no existe, la agrega al diccionario con el valor por defecto y luego devuelve el valor.

Acceso a los elementos

Se puede acceder a un valor indicando la clave correspondiente entre corchetes:

```
print(diccionario["nombre"])  # Ana
```

Si usamos el diccionario "persona"(diccionario anidado):

```
print(persona["calificaciones"][2])  #9
print(persona["socials"]["twitter"]) #@Pepe
```

Si la clave no existe, se genera un error (KeyError). Para evitarlo, se recomienda usar el método get():

```
print(diccionario.get("nombre"))       # Ana
print(diccionario.get("profesion"))    # None (no lanza error)
```

También se puede proporcionar un valor por defecto si la clave no existe:

```
print(diccionario.get("profesion", "No especificada"))
```

Modificar y añadir elementos

Los valores asociados a una clave se pueden modificar fácilmente:

```
diccionario["edad"] = 26
persona["calificaciones"][2] = 10
```

Cambia el valor de "edad" y la tercera calificación de persona.

Si se asigna un valor a una clave que no existe, se añade al diccionario:

```
diccionario["profesion"] = "Ingeniera"
```

Eliminar elementos

Se pueden eliminar elementos mediante del o pop():

```
del diccionario["ciudad"]      # Elimina la clave "ciudad"
edad = diccionario.pop("edad")  # Elimina la clave "edad" y devuelve su valor
```

Si se usa la función **popItem**, elimina y devuelve el último par añadido al diccionario.

```
diccionario = {
    'nombre': 'Pepe',
    'edad': 30,
    'profesion': 'Ingeniera'
}
diccionario["nuevo"] = 8
ultimo = diccionario.popitem()
print(ultimo)  # ('nuevo', 8)
```

Combinar o actualizar diccionarios

Para sobreescribir un diccionario con los valores de otro diccionario, usar "**update**", lo que hace es que, si el primer diccionario tiene la misma clave que el segundo diccionario, se modifica el valor para que coincida con el segundo diccionario, si no existe la clave en el primer diccionario, la añade.

```
a = {"nombre": "Pepe", "edad": 25 }
b = {"nombre": "Ana", "es_estudiante": True }
a.update(b)
print(a) #{'nombre': 'Ana', 'edad': 25, 'es_estudiante': True}
```

Si una clave ya existe, su valor se sobrescribe.
Si no existe, se añade.

Comprobar si existe una clave

Se hace como ya hemos visto anteriormente para comprobar un valor en un diccionario o una cadena.

```
print("name" in persona) # False
print("nombre" in persona) # True
```

Filtrar un diccionario (Dictionary Comprehesion)

```
frutas = {
  "manzana": 3,
```

```
  "banana": 2,
  "naranja": 5
}
# Filtrar frutas con valor mayor que 2
frutas_filtradas = {k: v for k, v in frutas.items() if v > 2}
print(frutas_filtradas)
```

Acceso a claves y valores

Ya sabemos que un diccionario se compone de "clave: valor", podemos obtener no solo el valor como hemos hecho hasta ahora, sino que también podemos acceder a las claves.

Obtener todas las claves

Para obtener todas las claves de un diccionario, se utiliza el método .keys().

```
print("\nClaves del diccionario:")
print(persona.keys())
```

Este método devuelve un objeto tipo dict_keys que contiene todas las claves. Este es un ejemplo de salida:
Claves del diccionario:
dict_keys(['nombre', 'edad', 'ciudad'])

Puedes convertir el resultado en lista si necesitas trabajar con él directamente:

```
claves = list(persona.keys())
print(claves)
```

Obtener todos los valores de un diccionario

El método .values() permite acceder a todos los valores almacenados en el diccionario.

```
print("\nValores del diccionario:")
print(persona.values())
```

Ejemplo de salida:
Valores del diccionario:
dict_values(['Lucía', 30, 'Madrid'])

Obtener pares clave-valor

El método .items() devuelve una lista de pares (clave, valor) en forma de tuplas dentro de un objeto iterable.

```
print("\nPares clave-valor:")
print(persona.items())
```

Ejemplo de salida:
Pares clave-valor:
dict_items([('nombre', 'Lucía'), ('edad', 30), ('ciudad', 'Madrid')])

Este formato es ideal para recorrer el diccionario en un bucle for.

Recorrer un diccionario

Para acceder a todos los elementos se puede usar un bucle for con .items():

```
for clave, valor in diccionario.items():
```

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

Esto imprimirá todas las claves junto a sus valores.
Este es el enfoque más limpio y legible para mostrar la información completa de un diccionario.

setdefault

Es útil para asegurar que una clave existe antes de trabajar con ella.

```
usuario = {"nombre": "Pepe"}
pais = usuario.setdefault("pais", "España")

print(usuario)
# {'nombre': 'Pepe', 'pais': 'España'}

print(pais)
# España
```

Si la clave "pais" ya existiera, no la modificaría, simplemente devolvería su valor actual.

Copiar diccionarios sin vincularlos

En Python, las variables no almacenan directamente los datos, sino referencias a los objetos en memoria.
Esto significa que, cuando asignas un diccionario a otra variable, ambas apuntan al mismo objeto, no se crea una copia nueva.

```
a = {"x": 1, "y": 2}
b = a    # No crea una copia: b apunta al mismo diccionario que a

b["x"] = 99
print(a)  # {'x': 99, 'y': 2}
print(b)  # {'x': 99, 'y': 2}
```

b = a no copia los datos, solo hace que b apunte al mismo lugar en memoria.
Si modificas b, también cambias a, porque en realidad son el mismo objeto.

Podemos comprobarlo con id():

```
print(id(a), id(b))  # Mismo identificador en memoria
```

Copiar correctamente un diccionario

Para crear una copia independiente, puedes usar el método .copy() o la función dict().

Ejemplo con copy():

```
a = {"x": 1, "y": 2}
b = a.copy()   # Crea una copia independiente

b["x"] = 99
print(a)  # {'x': 1, 'y': 2}
print(b)  # {'x': 99, 'y': 2}
```

Ejemplo con dict():

```
a = {"x": 1, "y": 2}
b = dict(a)
b["y"] = 100
print(a)  # {'x': 1, 'y': 2}
print(b)  # {'x': 1, 'y': 100}
```

Ambas formas (.copy() y dict()) crean una copia superficial (shallow copy).

Copia superficial vs copia profunda

Si el diccionario contiene otros diccionarios o listas dentro, .copy() solo copia la estructura externa, no los objetos anidados.
Esto puede generar vínculos inesperados entre el original y la copia.

```
a = {"nombre": "Ana", "notas": [8, 9, 10]}
b = a.copy()

b["notas"][0] = 5  # Modificamos la lista interna desde 'b'

print(a)  # {'nombre': 'Ana', 'notas': [5, 9, 10]}
print(b)  # {'nombre': 'Ana', 'notas': [5, 9, 10]}
```

Aunque usamos .copy(), ambos diccionarios comparten la misma lista interna ("notas").
Esto ocurre porque .copy() no copia en profundidad.

```
import copy

a = {"nombre": "Ana", "notas": [8, 9, 10]}
b = copy.deepcopy(a)

b["notas"][0] = 5

print(a)  # {'nombre': 'Ana', 'notas': [8, 9, 10]}
print(b)  # {'nombre': 'Ana', 'notas': [5, 9, 10]}
```

Ahora sí, a y b son totalmente independientes, incluso en sus elementos internos.

Copia profunda

Para hacer una copia completamente independiente, incluyendo los objetos internos, usamos el módulo copy y su función deepcopy():

Ejercicio 9

Crea un diccionario con información de un libro: título, autor y año. Imprímelo completo.

Ejercicio 10

Dado un diccionario con notas de varios alumnos, muestra por pantalla los que han aprobado (nota mayor o igual a 5).

```
notas = {
```

```
    "Ana": 8,
    "Luis": 4,
    "María": 6,
    "Pedro": 3
}
```

Ejercicio 11
Dado un texto, cuenta cuántas veces aparece cada letra (ignorando espacios y sin distinguir mayúsculas y minúsculas).

6. Retos

Reto 1
¿Está en Equilibrio la Alianza entre Reed Richards y Johnny Storm?
En el universo de los 4 Fantásticos, la unión y el equilibrio entre los poderes es fundamental para enfrentar cualquier desafío. En este problema, nos centraremos en dos de sus miembros:
Reed Richards (Mr. Fantastic), representado por la letra R.
Johnny Storm (La Antorcha Humana), representado por la letra J.
Objetivo:
Crea una función en Python que reciba una cadena de texto. Esta función debe contar cuántas veces aparece la letra R (para Reed Richards) y cuántas veces aparece la letra J (para Johnny Storm) en la cadena.
- Si la cantidad de R y la cantidad de J son iguales, se considera que la alianza entre la mente y el fuego está en equilibrio y la función debe retornar True.
- Si las cantidades no son iguales, la función debe retornar False.
- En el caso de que no aparezca ninguna de las dos letras en la cadena, se entiende que el equilibrio se mantiene (0 = 0), por lo que la función debe retornar True.

Reto 2
En Jurassic Park, se ha observado que los dinosaurios carnívoros, como el temible T-Rex, depositan un número par de huevos. Imagina que tienes una lista de números enteros en la que cada número representa la cantidad de huevos puestos por un dinosaurio en el parque.
Importante: Solo se consideran los huevos de los dinosaurios carnívoros (T-Rex) aquellos números que son pares.
Objetivo:
Escribe una función en Python que reciba una lista de números enteros y devuelva la suma total de los huevos que pertenecen a los dinosaurios carnívoros (es decir, la suma de todos los números pares en la lista).

Reto 3
Dado un array de números y un número goal, encuentra los dos primeros números del array que sumen el número goal y devuelve sus índices. Si no existe tal combinación, devuelve None.
nums = [4, 5, 6, 2]
goal = 8
find_first_sum(nums, goal) # [2, 3]

7. Repaso

Ejercicio 1
Lea dos números enteros que serán los operandos de la operación (operando1 y operando2).
Lea un tercer número que identificará la operación.
Si operación es 0 calcula la suma de ambos operandos y muestra el resultado.
Si operación es 1 calcula la resta. Si operación es 2 calcula la multiplicación.
Si operación es 3 calcula la división.
Si operación no coincide con ningún valor válido mostrará un mensaje de error.

Ejercicio 2
Realizar un algoritmo para determinar, de N cantidades introducidas por teclado:
- La media aritmética.
- El número más alto.
- El número más bajo.

Ejercicio 3
Este ejercicio consiste en realizar un juego de adivinar un número. El número a adivinar se genera al inicio del juego automáticamente mediante una función de la librería de Python "random" que genera números aleatorios.
Para usar la librería es necesario importarla, recomendado al principio del código: import random.
Para obtener un número entero aleatorio se usa la función "randint" que recibe por parámetro el rango entre el cual generar el número aleatorio. Por ejemplo, un número aleatorio del 1 al 10 sería: random.randint(1,10)
El juego consiste en lo siguiente:
- Pedir al usuario que ingrese la dificultad del juego, esta será un número del 10 al 50. Este número será utilizado para generar el número aleatorio.
- Pedir al usuario que adivine el número generado.
- El usuario tiene un máximo de 4 intentos para adivinar el número.
- Si no ha acertado, informarle del error y de los intentos consumidos. También comunicarle una "pista" diciéndole si el número a adivinar es mayor o menor al introducido.
- Si adivina el número en los 4 intentos, comunica el acierto y el número de aciertos consumidos, por ejemplo: "¡Correcto: has acertado el número en 3 aciertos!"
- Si pasan los 4 intentos y no ha adivinado el número, comunicar que ha perdido el juego.
- Tanto si ha acertado como si ha perdido, darle la oportunidad de jugar de nuevo, seleccionando una nueva dificultad.

Ejercicio 4
Realiza una función que reciba un número entero positivo N y muestre en pantalla un patrón de asteriscos con N filas como en el siguiente ejemplo para N=7

```
*
**
***
****
*****
******
*******
```

8. Conjuntos

Un conjunto (set) es una colección no ordenada, sin elementos duplicados y con búsqueda muy rápida (hashing).
Son muy útiles cuando queremos eliminar duplicados de una lista, comprobar si un elemento pertenece a un grupo o realizar operaciones matemáticas de conjuntos (unión, intersección, diferencia…).
Características:

- No tienen orden: no mantienen el orden en que se añaden los elementos.
- No admiten duplicados: cada valor aparece solo una vez.
- No admiten elementos mutables: no se pueden incluir listas ni diccionarios dentro de un conjunto.
- Permiten operaciones de álgebra de conjuntos como unión o intersección.
- Son muy rápidos para buscar elementos gracias a su implementación mediante tablas hash.

Además, existe una versión inmutable de los conjuntos llamada **frozenset**, que veremos al final.

Crear conjutos

Para crear un conjunto, puedes usar llaves {} o la función set().

```
# Con llaves
vocales = {"a", "e", "i", "o", "u"}


# Desde un iterable (elimina duplicados)
numeros = set([1, 2, 2, 3, 3, 3])  # {1, 2, 3}


# Conjunto vacío (¡no uses {} porque eso crea un dict!)
vacio = set()


# Conjunto de tuplas (válido, son inmutables)
puntos = {(0, 0), (1, 2), (0, 0)}  # {(0, 0), (1, 2)}
```

No se pueden meter listas ni diccionarios dentro de un conjunto, ya que no son tipos inmutables:

```
# Error:
conjunto = {[1, 2, 3]}  -> TypeError: unhashable type: 'list'
```

Ventaja: al crear un conjunto a partir de una lista, los elementos duplicados se eliminan automáticamente.

Convertir otras estructuras en conjuntos

Cualquier iterable (lista, tupla, cadena, etc.) se puede convertir en conjunto con set().

```
lista = [1, 2, 2, 3, 3, 3]
conjunto = set(lista)
```

```
print(conjunto)
```
Salida: {1, 2, 3}
Muy útil para eliminar duplicados.

Acceso y recorrido de conjuntos
No se puede acceder por índice (conjunto[0] da error), pero sí se puede recorrer con un bucle for:
```
frutas = {"manzana", "pera", "naranja"}
for fruta in frutas:
    print(fruta)
```
Salida posible (ya que al no estar ordenado puede variar entre ejecuciones):
naranja
manzana
pera

Comprobar pertenencia
Usa in o not in para saber si un elemento está dentro del conjunto:
```
frutas = {"manzana", "pera", "naranja"}
print("pera" in frutas)     # True
print("kiwi" not in frutas) # True
```

Añadir y eliminar elementos
Se añaden elementos al conjunto utilizando add()
```
numeros = {1, 2, 3}
numeros.add(4)
print(numeros)
```
Salida: {1, 2, 3, 4}

Se eliminan elementos del conjunto utilizando remove() o discard()
```
numeros = {1, 2, 3}
numeros.remove(2)   # Elimina 2
print(numeros)
```
Salida: {1, 3}
Si el elemento no existe, remove() lanza error. Si no queremos que lance un error, usar discard()
```
numeros.discard(10)  # No hace nada si el elemento no existe
```

Eliminar y obtener un elemento aleatorio: pop()
Como los conjuntos no tienen orden, pop() elimina un elemento aleatorio.
```
frutas = {"pera", "manzana", "naranja"}
elemento = frutas.pop()
print("Eliminado:", elemento)
print("Queda:", frutas)
```

Si queremos vaciar un conjunto, utilizar clear()

```
frutas.clear()
print(frutas)  # set()
```

Operaciones matemáticas con conjuntos

Python permite realizar operaciones típicas de teoría de conjuntos: unión, intersección, diferencia, etc.

Unión

Devuelve un conjunto con todos los elementos de ambos.

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

Salida: {1, 2, 3, 4, 5}

Intersección

Devuelve los elementos comunes a ambos conjuntos.

```
print(a & b)
print(a.intersection(b))
```

Salida: {3}

Diferencia

Devuelve los elementos de a que no están en b.

```
print(a - b)
print(a.difference(b))
```

Salida: {1, 2}

Diferencia simétrica

Devuelve los elementos que están en un conjunto u otro, pero no en ambos.

```
print(a ^ b)
print(a.symmetric_difference(b))
```

Salida: {1, 2, 4, 5}

Subconjuntos y superconjuntos

issuperset(): Comprueba si un conjunto contiene a otro.
issubset(): Comprueba si un conjunto es subconjunto de otro.

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


print(b.issubset(a))   # True → b está contenido en a
print(a.issuperset(b)) # True → a contiene a b
```

Conjuntos inmutables: frozenset

Python también tiene una versión inmutable del conjunto: frozenset.

```
f = frozenset([1, 2, 3])
```

```
f.add(4) # Error: no se puede modificar
print(f)
```

Se usa cuando necesitas un conjunto que no pueda cambiar, por ejemplo, como clave de un diccionario.

update(iterable)

update modifica el conjunto en el sitio añadiendo todos los elementos de uno o varios iterables (lista, tupla, otro set, rango, etc.).
Es equivalente a una unión in-place.

```
A = {1, 2}
A.update([2, 3, 4])     # añade 3 y 4; 2 ya estaba
print(A)                # {1, 2, 3, 4}

# Varios iterables a la vez
B = {4, 5}
A.update(B, (5, 6), range(6, 9))
print(A)                # {1, 2, 3, 4, 5, 6, 7, 8}
```

La diferencia con unión es que unión devuelve un conjunto nuevo, update cambia el original.

isdisjoint(otro)

Devuelve True si no comparten ningún elemento; es decir, si la intersección es vacía.

```
A.isdisjoint(B)  # True si A y B no comparten elementos
```

Ejemplo:

```
A = {1, 2}
B = {3, 4}
C = {2, 5}

print(A.isdisjoint(B))  # True  (no hay elementos comunes)
print(A.isdisjoint(C))  # False (comparten el 2)
```

No hace falta que el parámetro “otro” sea un set; puede ser cualquier iterable.

```
prohibidas = {"rm", "reboot"}
entrada = ["ls", "cat"]

print(prohibidas.isdisjoint(entrada))  # True
print(prohibidas.isdisjoint(["mv", "rm"]))  # False
```

También existen operadores equivalentes:

```
A <= B   # Subconjunto
A < B    # Subconjunto propio
B >= A   # Superconjunto
B > A    # Superconjunto propio
```

Comprensiones de conjuntos

Las comprensiones de conjuntos permiten crear conjuntos a partir de expresiones o bucles, de forma compacta.

```
# Cuadrados de 0 a 9
cuadrados = {n*n for n in range(10)}  # {0,1,4,9,16,25,36,49,64,81}


# Filtrar pares
pares = {n for n in range(20) if n % 2 == 0}
print(pares)
```

Es similar a las comprensiones de listas, pero con llaves {}.

Casos de uso frecuentes

Eliminar duplicados de una lista

```
nombres = ["Ana", "Luis", "Ana", "Marta", "Luis"]
unicos = list(set(nombres))
print(unicos)  # ['Luis', 'Marta', 'Ana']
```

Comprobar pertenencia rápida

```
prohibidas = {"rm", "shutdown", "reboot"}
cmd = input("Comando: ").strip()
if cmd in prohibidas:
    print("Comando no permitido.")
```

Intersección de etiquetas o habilidades

```
skills_user = {"python", "git", "docker"}
skills_job = {"python", "fastapi", "docker"}
comunes = skills_user & skills_job
print(comunes)  # {'python', 'docker'}
```

Conjuntos inmutables: frozenset

Un frozenset es una versión inmutable de un conjunto, es decir, no se puede modificar una vez creado.

```
A = frozenset({1, 2, 3})
B = frozenset({3, 4, 5})


print(A | B)  # Unión
# A.add(6) -> ❌ Error: no se pueden modificar los frozenset
```

Su utilidad principal es permitir que un conjunto pueda ser usado como clave en un diccionario, ya que es inmutable.

Buenas prácticas

- No confíes en el orden de los elementos: los conjuntos no están ordenados.
- Si necesitas mostrar los elementos ordenados, usa sorted(set).

• Usa set() para eliminar duplicados rápidamente de una lista.
• Usa copy() si necesitas clonar un conjunto.
• Usa frozenset cuando necesites un conjunto inmutable (por ejemplo, como clave de diccionario).
• Las operaciones in y not in son mucho más rápidas con conjuntos que con listas.

Ejercicios conjuntos

Ejercicio 1
Dada una lista de números, elimina los duplicados y ordénala.
numeros = [3, 5, 3, 2, 2, 9, 1]

Ejercicio 2
Comprueba si un usuario tiene los permisos mínimos requeridos.
usuario = {"read", "write"}
requeridos = {"read"}

Ejercicio 3
Muestra qué productos hay en una tienda, pero no en la otra.
tienda1 = {"raton", "teclado", "monitor"}
tienda2 = {"monitor", "altavoces"}

Ejercicio 4 – Contador de palabras
Crea una función llamada contar_palabras que reciba un texto y cuente la frecuencia de cada palabra en el texto. Debes eliminar primero los signos de puntuación.
La función Retorna un diccionario que almacena las palabras como claves y su frecuencia como valores.
Una vez obtienes el resultado, muestra la frecuencia de cada palabra recorriendo el diccionario con un for.

Ejercicio 5 – Lotería.
Crea una función llamada "premios" para simular un sorteo de lotería de 2 cifras. Se harán 20 tiradas. Se debe utilizar el módulo random para generar las tiradas. Cada tirada es un número, todas las tiradas se almacenan en un conjunto y se retornan. No puede haber 2 números iguales, es decir, si ya ha salido el 54, no puede volver a salir...

Crea otra función llamada "apuesta" con 5 tiradas aleatorias. Estas serán los números que juega el jugador.

Crea otra función llamada "comprobación" que compara ambos conjuntos para saber cuántos números ha acertado. Debes hacer la comprobación mediante operaciones de conjuntos. Debes retornar una tupla con la cantidad de números acertados y la cantidad de números no acertados, es decir (acertados, no_acertados).

Ejercicio 6 – Agenda telefónica
Implementa una agenda telefónica simple utilizando un diccionario. Las claves serán los nombres y los valores serán tuplas con el número de teléfono y la dirección.

Debes crear las siguientes funciones:
- Introducir un nuevo contacto

- Buscar un contacto por nombre y mostrar su teléfono y dirección.
- Eliminar contacto por nombre
- Mostrar toda la agenda
- Eliminar toda la agenda

Ejercicio 7 – Poker
Vamos a representar una mano de poker utilizando una tupla de cinco elementos, donde cada elemento es una tupla (valor, palo).

Por ejemplo: (('9', 'picas'), ('3', 'corazones'), ('8', 'diamantes'), ('9', 'tréboles'), ('5', 'tréboles'))

La función proporcionada genera una mano aleatoria.

Debes implementar varias funciones para evaluar si tiene poker, escalera de color, escalera o color. Estas funciones retornan true/false si cumple los criterios. Es decir, si la mano recibida como parámetro es poker, la función "es_poker" retorna True.

Una vez tengas las funciones por separado, crea una nueva función que reciba la mano y devuelva la combinación de mejor puntuación en forma de string, es decir, retorna uno de estos valores:

- 'poker'
- 'escalera color'
- 'escalera'
- 'color'
- none Si no tiene ninguna combinación válida.

9. Función zip

La función zip() en Python se utiliza para combinar (empaquetar) dos o más iterables (como listas, tuplas o cadenas) elemento a elemento, creando pares o tuplas con los valores que ocupan la misma posición en cada iterable.
Sintaxis:
zip(iterable1, iterable2, ...)

• Cada posición del resultado será una tupla con los elementos correspondientes de cada iterable.
• Si los iterables tienen longitudes diferentes, zip() se detiene en el más corto.
• Devuelve un objeto iterable (de tipo zip), que normalmente se convierte en lista o tupla para visualizarlo.
Ejemplo básico:

```
nombres = ["Ana", "Luis", "María"]
edades = [20, 25, 30]

resultado = zip(nombres, edades)
print(list(resultado))
```

Salida: [('Ana', 20), ('Luis', 25), ('María', 30)]

Su funcionamiento lo habremos deducido:

• Toma los elementos por posición (índice 0 con índice 0, índice 1 con índice 1, etc.).
• Si las secuencias no tienen la misma longitud, se detiene en la más corta.
• Devuelve un iterador (no una lista directamente), por lo que puedes recorrerlo o convertirlo.

Ejemplo con diferentes longitudes:

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

print(list(zip(a, b)))
```

Salida: [(1, 'a'), (2, 'b')]

Ejemplo práctico: sumar listas

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]

suma = [a + b for a, b in zip(lista1, lista2)]
print(suma)
```

Salida: [5, 7, 9]

Crear un diccionario con zip()

```
claves = ["nombre", "edad", "ciudad"]
valores = ["Ana", 25, "Madrid"]

persona = dict(zip(claves, valores))
print(persona)
```

Salida: {'nombre': 'Ana', 'edad': 25, 'ciudad': 'Madrid'}

Recorrer varias listas a la vez

```
nombres = ["Ana", "Luis", "María"]
notas = [8, 9, 7]

for nombre, nota in zip(nombres, notas):
    print(f"{nombre} ha sacado un {nota}")
```

Salida: Ana ha sacado un 8
Luis ha sacado un 9
María ha sacado un 7
Es más limpio que usar índices.

Desempaquetar con zip(*)
También puedes invertir el proceso con zip(*iterable).
Sirve para desempaquetar una lista de tuplas en listas separadas.

```
pares = [(1, 'a'), (2, 'b'), (3, 'c')]
numeros, letras = zip(*pares)
```

```
print(numeros)
print(letras)
```

Salida: (1, 2, 3)
('a', 'b', 'c')

10. Type hints

Los type hints (anotaciones de tipo) en Python permiten indicar el tipo de dato que se espera en parámetros y retornos de funciones (también puede hacerse en variables).
No cambian el comportamiento del código (Python sigue siendo tipado dinámico). El intérprete ignorará las anotaciones de tipos.
Ayudan a la legibilidad y mantenimiento (actúan como documentación) y facilitan la detección de errores mediante herramientas y plugins de VSCode.
Ejemplo simple:

```
nombre: str = "Ana"
edad: int = 25
activo: bool = True
```

Aquí se está indicando que:
- nombre es de tipo str
- edad es de tipo int
- activo es de tipo bool

Python no obliga a cumplir estos tipos, solo los sugiere.
Si luego haces edad = "veinticinco", el programa funcionará, pero un buen editor te mostrará un aviso.

Type hints en funciones
Podemos especificar tanto el tipo de los parámetros como el tipo del valor de retorno de una función.
Sintaxis:

```
def nombre_funcion(parametro: tipo) -> tipo_retorno:
    # código
    return valor
```

Ejemplo de función con anotaciones:

```
def suma(a: int, b: int) -> int:
    return a + b
```

Indica que:
- a y b deben ser enteros (int)
- La función devuelve también un entero

Ejemplo de funciones con cadenas:

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

Salida: Hola, Ana

Type Hints: tipos de datos básicos
int → edad: int = 25
float → precio: float = 19.99

str → nombre: str = "Juan"
bool → activo: bool = True
list → numeros: list = [1, 2, 3, 4]
tuple → coordenadas: tuple = (19.43, -99.13)
set → colores: set = {"rojo", "verde", "azul"}
dict → edades: dict = {"Ana": 30, "Luis": 25}

Type Hints: tipos de datos
Lista indicando el tipo → numeros: List[int] = [1, 2, 3, 4]
Tupla indicando el tipo → coordenadas: Tuple[float, float] = (19.43, -99.13)
Tupla de longitud variable con solo enteros: valores_random: Tuple[int, ...] = (1, 5, 9, 2)
Conjunto indicando el tipo → colores: Set[str] = {"rojo", "verde", "azul"}
Diccionario indicando tipos clave y valor → edades: Dict[str, int] = {"Ana": 3, "Luis":2}
None → respuesta: None = None
Indicar 2 opciones posibles → dato: int | str = "Texto"
Indicar que un valor es opcional (tipo o None) → apellido: str | None = None
Indicar 2 opciones en listas → números: List[int | str] = [1, "Juan", 3]

La librería Typing
Python es un lenguaje de tipado dinámico, lo que significa que no necesitas declarar el tipo de una variable antes de usarla. Sin embargo, la librería estándar **typing** permite añadir anotaciones de tipo (type hints) para mejorar la claridad, la mantenibilidad y la robustez del código.
Algunos tipos avanzados (como TypedDict, Protocol, Literal, etc.) solo están en typing

Si queremos usar la librería para los tipos de datos, habrá que importarla de esta manera:

```
from typing import List, Dict, Tuple, Set
```

Ejemplo: Tipos literales (Literal) para restringir valores posibles.

```
from typing import Literal
def elegir_color(color: Literal['rojo', 'verde', 'azul']) -> str:
    return f"Has elegido el color {color}."
elegir_color('rojo')
```

Un type alias es simplemente un nombre alternativo para un tipo (o combinación de tipos).
Se usa para:

- Hacer el código más legible.
- Evitar repetir anotaciones largas o complejas.
- Documentar mejor la intención del tipo.

```
from typing import List


# Sin alias
def procesar_datos(datos: List[tuple[str, int]]) -> None:
    ...
```

```python
# Con alias
Registro = tuple[str, int]
def procesar_datos(datos: list[Registro]) -> None:
    ...
```

En este caso, Registro es un alias que representa una tupla de (nombre, edad).

A partir de **Python 3.10**, puedes declarar un alias de tipo de forma **explícita y segura** con la constante TypeAlias del módulo typing.

```python
from typing import TypeAlias

# Declaramos un alias de tipo
UserID: TypeAlias = int
```

Esto le dice a los analizadores de tipo (como pylance) que UserID no es una variable, sino un nombre de tipo.
Ejemplo:

```python
from typing import TypeAlias

UserID: TypeAlias = int
UserName: TypeAlias = str

def get_user_name(user_id: UserID) -> UserName:
    return f"Usuario_{user_id}"

print(get_user_name(42))
```

UserID es un alias de int.
UserName es un alias de str.
A efectos de ejecución, no cambia nada.
Pero para **type checkers**, UserID y int se tratan como equivalentes en tipo, aunque el alias mejora la semántica y la legibilidad.

Ejemplo con tipos complejos

Los type aliases brillan cuando los tipos son largos o repetitivos:

```python
from typing import TypeAlias, Tuple, Union, Dict

# Alias para un tipo complejo
JSONValue: TypeAlias = Union[str, int, float, bool, None, Dict[str, 'JSONValue'], list['JSONValue']]

def parse_json(data: JSONValue) -> None:
    ...
```

Aquí JSONValue describe recursivamente un valor JSON posible (cadena, número, booleano, diccionario o lista).
El alias hace que la función sea mucho más legible.

TypeAlias vs simple asignación

A veces la gente hace esto:

```
Vector = list[float]
```

Y también funciona, pero **no indica de forma explícita** que es un alias de tipo.
Si un analizador de tipos muy estricto no lo deduce correctamente (raro, pero puede pasar), conviene usar TypeAlias:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

Así evitas que un IDE o checker crea que Vector es una variable asignada a list[float] en tiempo de ejecución.
Ejemplo completo:

```
from typing import TypeAlias, Dict, List


Producto: TypeAlias = Dict[str, float]
Catalogo: TypeAlias = List[Producto]


def total_catalogo(catalogo: Catalogo) -> float:
    """Suma los precios de todos los productos."""
    return sum(p["precio"] for p in catalogo)


# Uso
catalogo_ejemplo: Catalogo = [
    {"nombre": 10.0, "precio": 9.99},
    {"nombre": 5.0, "precio": 5.50},
]


print(total_catalogo(catalogo_ejemplo))  # 15.49
```

Tipado de colecciones (listas, diccionarios…)

Desde Python 3.9, se puede indicar el tipo de los elementos dentro de una colección directamente, sin importar de qué módulo provengan.
Ejemplo con lista de enteros:

```
numeros: list[int] = [1, 2, 3, 4]
```

Ejemplo con diccionario con claves str y valores int

```
edades: dict[str, int] = {"Ana": 25, "Luis": 30}
```

Ejemplo con lista de tuplas

```
coordenadas: list[tuple[int, int]] = [(2, 3), (4, 5)]
```

Unión de tipos

A veces, una variable o parámetro puede aceptar más de un tipo.
Para eso, se usa el operador | (pipe).

Ejemplo:

```
def convertir_a_str(dato: int | float) -> str:
```

```
    return str(dato)
```

Valores opcionales (Optional)

Si un parámetro puede ser de un tipo o ser None, se usa Optional.

```
from typing import Optional


def saludar(nombre: Optional[str] = None) -> str:
    if nombre:
        return f"Hola, {nombre}"
    return "Hola, visitante"
saludar()
```

Salida: Hola, visitante

Type hints en funciones que no devuelve nada

Si una función no devuelve ningún valor (solo realiza acciones), se indica con -> None

```
def mostrar_mensaje() -> None:
    print("Esto no devuelve nada.")
```

Type hints en funciones con argumentos variables

```
def sumar_todo(*args: int) -> int:
    return sum(args)
print(sumar_todo(1, 2, 3, 4))  # 10
```

*args: int indica que todos los argumentos deben ser enteros.

En VSCode, para que las anotaciones sean útiles, se recomienda activar la comprobación estática.

1. Abre la paleta de comandos (Ctrl + ,)
2. Busca “type checking mode”
3. Selecciona "strict"

Así el editor marcará los errores de tipo sin detener el programa.