# Tema 3

Tuplas, diccionarios y conjuntos

# Tuplas

## ¿Qué es una tupla?

Una tupla es una colección **ordenada** e **inmutable** de elementos. Imagina una lista de cosas que no pueden cambiar una vez creadas, como los días de la semana o las coordenadas de un punto en un espacio.

Las tuplas se definen asignando a una variable un conjunto de valores separados por comas. Los paréntesis son opcionales.

```
tupla = 1, 2, 3

print(tupla)
type(tupla)
```

```
(1, 2, 3)

tuple
```

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

print(tupla)
type(tupla)
```

```
(1, 2, 3)

tuple
```

# Tuplas

## ¿Por qué usar tuplas?

- **Inmutabilidad:** Esto las hace seguras para usar como claves en diccionarios o como elementos de otras estructuras de datos.
- **Eficiencia:** Al ser inmutables, las tuplas suelen ser más eficientes en  memoria y velocidad de acceso que las listas.
- **Claridad:** Representan datos que no deberían cambiar, mejorando la legibilidad del código.

Acceso: Se accede a los elementos de una tupla utilizando índices, igual que en listas. El primer elemento tiene índice 0.

```
mi_tupla = (1, 2, 3)
print(mi_tupla[1])
```

```
2
```

```
mi_tupla = (1, 2, 3)
print(mi_tupla[-1])
```

```
3
```

# Tuplas

**Slicing**: Funciona igual que en listas. El slice es una tupla.

```
mi_tupla = (1, 2, 3)
print(mi_tupla[::-1])
```

```
(3, 2, 1)
```

**Constructor de tuplas:** Usando el mismo nombre de la clase podemos invocar al constructor y pasarle cualquier secuencia para crear una tupla a partir de ella.

```
import numpy as np

mi_array = np.array([1, 2, 3, 4])

mi_tupla = tuple(mi_array)
print(mi_tupla)
```

```
(1, 2, 3, 4)
```

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

mi_tupla = tuple(matriz)
print(mi_tupla)
```

```
([1, 2], [3, 4])
```

# Tuplas: Métodos

Las tuplas son objetos inmutables, no se pueden modificar. Si queremos cambiar algo (añadir elemento, quitar elemento, cambiar un elemento por otro) debemos crear una nueva tupla.

Los únicos métodos que tiene son:

- **count(x):** Cuenta el número de veces que aparece el elemento x en la tupla.
- **index(x):** Devuelve el índice de la primera aparición del elemento x.

```
tupla_numeros = (1, 2, 2, 10, 10)
print(tupla_numeros.count(2))  # Imprime 2
print(tupla_numeros.index(10))  # Imprime 3
```

```
2
3
```

# Tuplas: Desempaquetado

Podemos asignar los elementos de una tupla a múltiples variables en una sola línea (igual que en listas).

```
x, y, z = (10, 20, 30)
print(x)  # Imprime 10
print(y)  # Imprime 20
print(z)  # Imprime 30
```

```
10
20
30
```

Muchas funciones retornan varios valores en una tupla que posteriormente desempaquetamos.

```
def mi_funcion():
    return 4, 5

retorno = mi_funcion()
print(type(retorno))
x, y = retorno
print(x)
print(y)
```

```
<class 'tuple'>
4
5
```

# Tuplas: Intercambiar variables

El intercambio de variables es un caso particular del uso de tuplas

```
a = 2
b = 3

b, a = a, b  # Se crea la tupla y luego se desempaqueta

print(f"{b=}")
print(f"{a=}")
```

```
b=2
a=3
```

# Tuplas

**Casos prácticos de uso de tuplas**

- **Coordenadas:** Representar puntos en un espacio 2D o 3D.
- **Dimensiones:** Definir las dimensiones de una imagen o matriz.
- **Datos inmutables:** Almacenar datos que no deben ser modificados, como constantes o configuraciones.
- **Devolución de múltiples valores:** Una función puede devolver una tupla para retornar múltiples valores.
- **Claves en diccionarios:** Las tuplas son inmutables, por lo que pueden utilizarse como **claves en diccionarios.**

**Diferencia con listas**

| Característica | Tuplas | Listas |
|---|---|---|
| Mutabilidad | Inmutables | Mutables |
| Eficiencia | Generalmente más eficientes | Menos eficientes que las tuplas |
| Uso común | Datos que no cambian, claves de diccionarios | Almacenar colecciones de datos que pueden cambiar |

# Tuplas: inmutabilidad de sus elementos

Como sabemos una tupla es inmutable, pero si contiene elementos mutables (una lista) no significa que no podamos cambiar este elemento interno.

```
lista = [1, 2]
mi_tupla = (1, lista, "hola")

print(mi_tupla)
lista[0] = 55   # modifico la posición 0 de la lista almacenada en la tupla
print(mi_tupla)
```

```
(1, [1, 2], 'hola')
(1, [55, 2], 'hola')
```

La tupla en sí misma sigue siendo la misma. El identificador (la dirección de memoria) de la tupla no ha cambiado.

Lo que ha cambiado es el contenido de uno de los objetos a los que la tupla hace referencia. En este caso, hemos modificado el contenido de la lista que forma parte de la tupla.

# Tuplas: métodos

Debido a la inmutabilidad de las tuplas, la clase apenas tiene métodos propios.

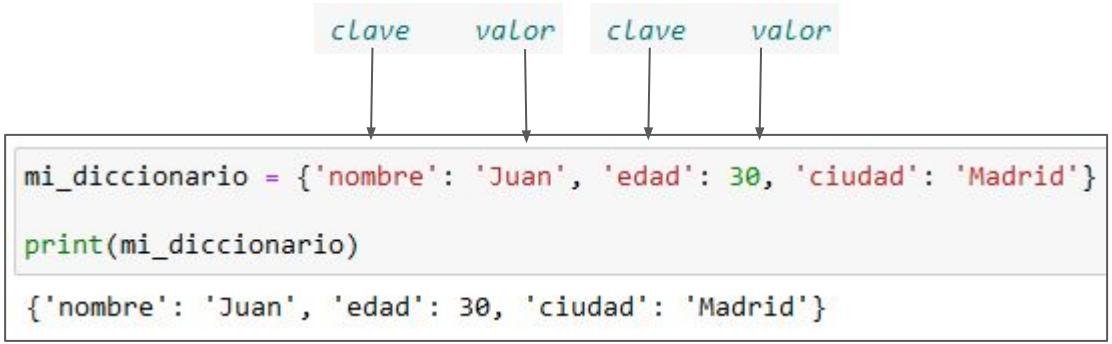
- **count(elemento):** Devuelve el número de veces que un elemento aparece en la tupla.
- **index(elemento):** Devuelve el índice de la primera aparición de un elemento en la tupla. Si el elemento no se encuentra, lanza un ValueError.

Se pueden aplicar muchos operadores al igual que a las listas:

- **Concatenación:** Puedes concatenar dos o más tuplas utilizando el operador **+**.
- **Repetición:** Puedes repetir una tupla un número determinado de veces utilizando el operador *****.
- **Indexar / Slicing:** Puedes extraer porciones de una tupla utilizando la notación de slicing.
- **Operadores de comparación:** Puedes comparar tuplas utilizando los operadores **==, !=, <, >, <=, >=**. La comparación se realiza elemento a elemento.
- **Pertenencia**: **in / not in**

# Diccionarios

Un diccionario en Python es una colección de elementos desordenada, pero accesible, que almacena datos en **pares clave-valor.** Cada **clave debe ser única** dentro del diccionario y **se utiliza para acceder al valor** asociado. Los diccionarios se definen utilizando llaves `{}`.



# Diccionarios: creación

1. Podemos crear un diccionario literalmente asignando los pares clave,valor encerrados entre llaves.

```
mi_diccionario = {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Madrid'}

print(mi_diccionario)
```

```
{'nombre': 'Juan', 'edad': 30, 'ciudad': 'Madrid'}
```

2. Mediante el constructor, pasando como argumento una lista o tupla de pares (clave,valor) en forma de tupla.

```
mi_dict = dict([("nombre", "juan"), ("edad", 8)])
print(mi_dict)
```

```
{'nombre': 'juan', 'edad': 8}
```

# Diccionarios

**Las claves de un diccionario deben ser inmutables.** Esto significa que su valor no puede ser cambiado después de que la clave haya sido creada. Los tipos de datos más comunes que cumplen con este requisito son:

- **Enteros (int) y números de punto flotante (float).**
- **Cadenas (str)**
- **Tuplas:** Colecciones ordenadas e inmutables de elementos (para ser clave, los elementos deben ser inmutables)

**¿Por qué deben ser inmutables?**

- **Hashing:** Las claves de los diccionarios se utilizan para calcular un hash, que es una especie de huella digital única. Esta huella digital se utiliza para localizar rápidamente el valor asociado a la clave. Si las claves fueran mutables, el hash podría cambiar y el diccionario no podría encontrar el valor de forma eficiente.

- **Consistencia:** La inmutabilidad garantiza que las claves no cambien después de ser creadas, lo que evita problemas de inconsistencia en el diccionario.

# Diccionarios

Las tuplas son inmutables y pueden ser claves para un diccionario

```
clave = (2, 3)
mi_dict = {clave: "valor"}
print(mi_dict)
```

```
{(2, 3): 'valor'}
```

Las tuplas pueden tener una lista (mutable) en su interior sin ningún problema. Pero en este caso ya no sirve como clave porque no se puede calcular el hash.

```
clave = (1, [55, 2])
mi_dict = {clave: "valor"}
print(mi_dict)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[53], line 2
      1 clave = (1, [55, 2])
----> 2 mi_dict = {clave: "valor"}
      3 print(mi_dict)

TypeError: unhashable type: 'list'
```

# Diccionarios (hash)

Una forma de comprobar si un objeto es “hasheable” es mediante el siguiente código que calcula el hash de un objeto. Si no se puede calcular el hash lanza una excepción.

```
def es_hashable(obj):
    try:
        hash(obj)
        return True
    except TypeError:
        return False
```

```
mi_tupla = ("Edad", 2)
es_hashable(mi_tupla)
```

```
True
```

```
mi_lista = [1, 2, 3]
es_hashable(mi_lista)
```

```
False
```

# Diccionarios: acceso

Accedemos a los datos del diccionario por la clave.

```
diccionario = {"nombre": "Juan", "apellido": "García", "edad": 45}

# Accedemos a los elementos por la clave
print(diccionario["edad"])
```

```
45
```

Si la clave no existe se genera una excepción.

```
print(diccionario["apellido2"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[62], line 1
----> 1 print(diccionario["apellido2"])

KeyError: 'apellido2'
```

El diccionario es una estructura iterable. Con un bucle **for** recorremos las claves. Con cada clave que nos da el for, indexamos su valor correspondiente.

```
for clave in diccionario:
    print(f"{clave} = {diccionario[clave]}")
```

```
nombre = Juan
apellido = García
edad = 45
```

# Diccionarios: métodos

## Métodos de clase diccionario:

- **keys():** Devuelve una vista de todas las claves del diccionario.
- **values():** Devuelve una vista de todos los valores del diccionario.
- **items():** Devuelve una vista 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.

# Ejemplo recorrer diccionario

El método **diccionario.items()** es ideal si deseamos recorrer un diccionario al completo, tanto las claves como sus valores.

El método devuelve una especie de lista de tuplas con los pares (clave, valor) que podemos recorrer con un for.

```
for clave, valor in diccionario.items():
    print(f"{clave=}, {valor=}")
```

```
clave='nombre', valor='Juan'
clave='apellido', valor='García'
```

# Diccionario: Acceso a elementos

Imagina que tienes una función donde recibes por parámetro una clave que introduce el usuario por teclado y debes acceder a su valor correspondiente en un diccionario.

Si esa clave no existe, el intérprete lanza excepción. Podemos hacer control de excepciones, lo que “ensucia” el código, pero para evitar esto existen alternativas.

1. Comprobar antes que la clave existe ——-->

```
clave = "apellido2"
if clave in diccionario:
    print(diccionario[clave])
else:
    print("no existe")
```

```
no existe
```

2. Usar método get() —>

```
print(diccionario.get("nombre", "no existe"))
```

```
Juan
```

```
print(diccionario.get("apellido2", "no existe"))
```

```
no existe
```

# Conjuntos

## ¿Qué es un conjunto en Python?

Un conjunto en Python sigue la noción de conjunto matemático, es una colección desordenada de elementos únicos. Esto significa que cada elemento aparece solo una vez dentro del conjunto y el orden no importa. Los conjuntos son muy útiles para realizar operaciones matemáticas de conjuntos como unión, intersección, diferencia, etc.

## ¿Para qué son útiles los conjuntos?

- **Eliminar duplicados:** Los conjuntos son ideales para eliminar elementos duplicados de una lista de forma automática.
- **Comprobar pertenencia:** Puedes verificar rápidamente si un elemento está presente en un conjunto.
- **Operaciones de conjunto:** Puedes realizar operaciones como unión, intersección, diferencia, etc., de manera eficiente

# Conjuntos: creación

El constructor de conjuntos es **set(iterable)** y puede recibir cualquier iterable, como: tupla; lista; diccionario (realizará un conjunto con las claves).

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

# Los elementos del conjunto tienen su propio orden
# no tiene elementos repetidos
print(set(mi_lista))
```

```
{1, 2, 3}
```

También podemos crear directamente un conjunto asignando a una variable los valores del conjunto encerrados entre llaves.

```
# Conjunto con elementos
otro_conjunto = {1, "que", 2, 3, "hola"}
print(otro_conjunto)
```

```
{1, 2, 3, 'que', 'hola'}
```

# Conjuntos: métodos (se invocan conjunto.nombre_método()

`add(elem)` Agrega un elemento al conjunto.

`remove(elem)` Elimina un elemento del conjunto. Lanza un error si el elemento no existe.

`discard(elem)` Elimina un elemento del conjunto si existe, de lo contrario no hace nada.

`pop()` Elimina y devuelve un elemento arbitrario.

`clear()`Elimina todos los elementos del conjunto.

`union(otro_set)` Devuelve un nuevo conjunto con todos los elementos de ambos conjuntos.

`intersection(otro_set)`Devuelve un nuevo conjunto con los elementos comunes a ambos.

`difference(otro_set)` Devuelve un nuevo conjunto con los elementos que están en el primer conjunto pero no en el segundo.

`symmetric_difference(otro_set)` Devuelve un nuevo conjunto con los elementos que están en uno u otro conjunto, pero no en ambos.

`issubset(otro_conjunto)` Devuelve `True` si el conjunto es subconjunto del otro.

`issuperset(otro_conjunto)` Devuelve `True` si el conjunto es superconjunto del otro.

# Conjuntos: operadores

Existen operadores equivalentes a las operaciones de conjuntos

- Unión: **|**
- Intersección: **&**
- Diferencia: **-**
- Diferencia simétrica: **^**

# Función zip

La función `zip()` sirve para combinar elementos de múltiples iterables (como listas, tuplas, cadenas, etc.) en un solo iterable de tuplas (Un **generador** de tuplas).

**Características importantes de `zip()`:**

- **Iterador:** Devuelve un iterador, por lo que debes recorrer el resultado con un for, o convertirlo a tupla, lista o diccionario.
- **Longitud:** La longitud del resultado es igual a la longitud del iterable más corto.

```
impares=(1, 3) # 2 elementos
pares=(4, 6, 8) # 3 elementos

print(list(zip(impares, pares)))
```

```
[(1, 4), (3, 6)]
```

```
# Crear un diccionario a partir de listas de claves y valores
claves = ['nombre', 'edad', 'ciudad']
valores = ['Ana', 25, 'Madrid']
mi_diccionario = dict(zip(claves, valores))
print(mi_diccionario)
```

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

# Recordando desempaquetar tuplas/listas

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)
print(resto)
print(ultimo)
```

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

Es una tupla

resto es tipo lista, no tupla

```
# Desempaquetar siempre devuelve una lista si es más de un elemento
tupla = (1, 2, 3, 4)
primero, *resto = tupla
print(primero)
print(resto)
print(type(resto))
```

```
1
[2, 3, 4]
<class 'list'>
```

# Otras estructuras de datos

Python provee otras estructuras de datos adicionales disponibles a través de la librería estándar. Algunas de ellas son:

- deque:  Cola doble que mejora las listas para insercción/borrado por los extremos
- Counter: Generador de Contador de frecuencias.
- defaultdict: Diccionario con valores por defecto automáticos.
- namedtuple: Tuplas nombradas (los campos de cada tupla tienen una clave)
- heapq: Librería para implementar una cola de prioridad sobre una lista
- array: Lista tipada más eficente que la clase list.

# Otras estructuras de datos

**deque**: Cola doble más eficiente que una lista para insertar y eliminar por ambos extremos. Disponible a través de librería collections (`from collections import deque`)

Útil para implementar: Colas FIFO, Pilas LIFO, Historiales (undo/redo), Buffers.

**Constructor** deque(iterable, maxlen=n). **Métodos**:

- append(x): Añade a la derecha
- appendleft(x): Añade a la izquierda
- pop(x): Elimina y devuelve desde la derecha
- popleft(x): Elimina y devuelve desde la izquierda.
- rotate(n): A la derecha o izquierda n elementos (si n es negativo->izquierda)
- El parámetro **maxlen** proporciona un tamaño máximo. Al insertar un elemento, si se supera el máximo, se elimina por el lado contrario.

# Otras estructuras de datos

**Counter**: Es una subclase de diccionario, disponible a través del módulo collections. Está especializada en contar frecuencias de objetos “hashables” (elementos inmutables, al igual que las claves del diccionario).

Es decir, partiendo de una estructura iterable (como una lista), un Counter es un diccionario donde las claves son los elementos y los valores la frecuencia de aparición.

**Constructor**: Counter(iterable). Genera el dict con las frecuencias. Métodos:

- most_common(n): Lista con los N elementos más frecuentes con sus frecuencias.
- elements(): Genera un interador que repite cada elemento su frecuencia.
- update(iterable): Actualiza el contador con los elementos de otra estructura.
- subtract(iterable): Resta conteos.

# Otras estructuras de datos

**defaultdict**:  En un dict, cuando se accede a una clave que no existe, se lanza una excepción KeyError.  En defaultdict se crea automáticamente esa clave con un valor inicial. Está disponible a través de librería collections.

**Constructor**: defaultdict(valor_inicial). El valor inicial típicamente es: list, set, tuple o int (valor 0).

No añade métodos nuevos. Es un diccionario pero con claves iniciadas automáticamente si no existen.

# Otras estructuras de datos

defaultdict es útil cuando agrupamos, así evitamos comprobar si ya existe o no la clave.

```python
from collections import defaultdict
alumnos = [
    ("1A", "Ana"),
    ("1A", "Luis"),
    ("1B", "Marta"),
]
# El valor inicial de la clave será una lista
grupos = defaultdict(list)

for clase, nombre in alumnos:
    # No es necesario que compruebe si la clave existe
    grupos[clase].append(nombre)

print(grupos["1A"])  # ['Ana', 'Luis']
print(grupos["1B"])  # ['Marta']
```

# Otras estructuras de datos

**namedtuple**: Es una tupla normal (inmutable, indexable), pero con el extra de que cada posición tiene un nombre (una “clave” como un diccionario). Se puede acceder a cada valor a través de la clave, con tupla.nombre en lugar de tupla[index]. Aporta legibilidad al código.

¿Diccionario o namedtuple? Un namedtuple es inmutable y ocupa menos que un diccionario.

```
1  from collections import namedtuple
2
3  # Se crea la clase Persona con 2 claves, nombre y edad
4  Persona = namedtuple("Persona", ["nombre", "edad"])
5  # Creamos un objeto Persona con los 2 valores de la tupla
6  p = Persona("Ana", 30)
7  # Se indexa por posición o por clave
8  print(p[0])
9  print(p.nombre)
```

# Otras estructuras de datos

**array**: Array de valores del mismo tipo más similar a lenguajes como C (se reserva memoria de forma contigua). Es más eficiente que una lista. Es una alternativa de la librería estándar a usar ndarrays de NumPy si no queremos dependencias.

- Uso: Debemos indicar el tipo de los elementos del array y podemos pasar una lista de valores iniciales.

```
1  from array import array
2
3  # Crear un array de enteros ('i')
4  mi_array = array('i', [1, 2, 3, 4])
```

- Tipos: Los tipos son numéricos enteros o flotantes. Admite un tipo unicode para caracteres independientes (no cadenas como str). https://docs.python.org/3/library/array.html

# Otras estructuras de datos

Métodos:

- Dispone de métodos básicos de las listas como append, extend, insert, pop, remove, count…
- Propios:
  - tolist(): Devuelve una lista normal. Útil para una impresión más limpia.
  - fromlist(lista): Añade los elementos desde una lista.
- Atributos:
  - itemsize: Número de bytes por elemento.
  - typecode: Tipo del array.

Se pueden indexar, hacer slice y recorrer con un for, hacer len, min, max, sum.

# Otras estructuras de datos

**heapq:** Este módulo implementa una estructura de datos denominada montículo. Proporciona funciones para usar sobre una lista. La diferencia, es que nos permite tener acceso al elemento más pequeño (o más grande) forma eficiente, sin necesidad de ordenar la lista constantemente.

Es ideal cuando no es necesario ordenar toda la lista, sino solamente acceder al valor mínimo (o máximo). El elemento más pequeño siempre está en la posición [0]. Insertar y extraer en un heap es más eficiente que usar una lista invocando a sort cada vez que insertamos.

Usos:

- **Colar de prioridad**: Guardar una lista de tareas por prioridad. El heapq no ordena la lista, sino que te permite acceder al más prioritario.
- **Obtener los N elementos más grandes o pequeños** (más eficiente que ordenar la lista y hacer slice).
- Algoritmos para Grafos (Dijkstra).

Está disponible a través del módulo heapq -> import heapq

# Otras estructuras de datos

**Se puede trabajar con un Min-Heap (valores mínimos) o Max-Heap (valores máximos). El comportamiento por defecto es un Min-Heap. Las funciones para Min-Heap son:**

- `heapq.heapify(lista)` Convierte una lista normal en un heap *in-place* (sin crear una nueva).
- `heapq.heappush(heap, item):` Agrega un elemento al heap manteniendo el min-heap.
- `heapq.heappop(heap):`Extrae y devuelve el elemento **más pequeño** y configura un nuevo mín. Para acceder al mínimo sin extraer indexar la posición [0]
- `heapq.heapreplace(heap, item):` Saca el más pequeño e inserta uno nuevo en un solo paso más eficiente que hacer pop y luego push.
- `heapq.heappushpop(heap, item):` Apila el item en el heap, y luego extrae y devuelve el elemento más pequeño del montículo. Es más eficiente que hacer push y luego pop.

Las funciones para Max-Heap son iguales, acabando el nombre en “_max”. Además están las funciones:

- heapq.merge(*iterables). Retorna un generador para combinar en orden 2 iterables ordenados sin fusionar ambos previamente ni consumir memoria.
- heapq.nlargest(n, iterable). Retorna una lista con los n elementos más grandes del iterable.
- heapq.nsmallest(n, iterable). Retorna una lista con los n elementos más pequeños de iterable.