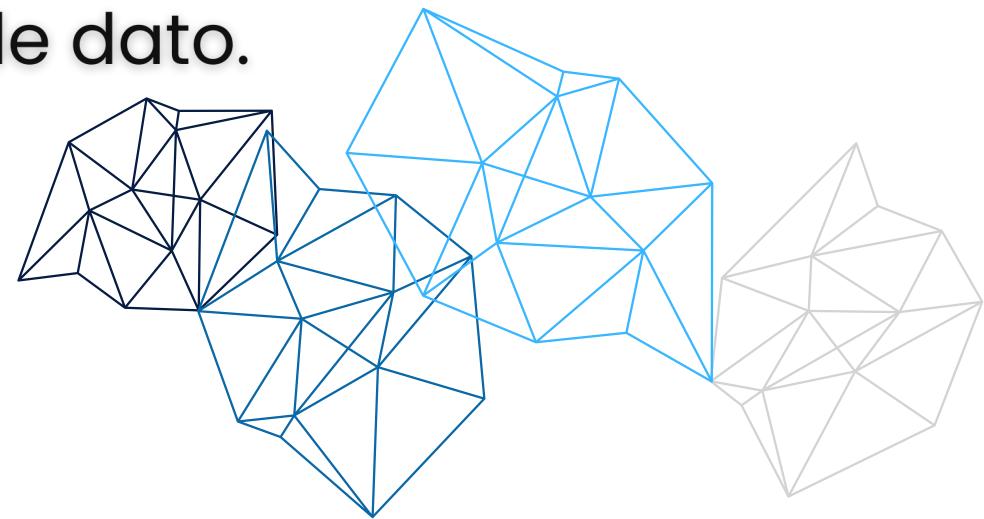


Estructuras de datos

Las estructuras de datos son formas de organizar y almacenar datos en la memoria de un programa de computadora. Son fundamentales en la programación y en la resolución de problemas computacionales, ya que permiten representar y manipular datos de una manera más eficiente y organizada.

En Python, existen varios tipos de estructuras de datos incorporadas en el lenguaje, tales como:

- **Listas (lists):** Son secuencias ordenadas de elementos que pueden ser de cualquier tipo de dato, y que pueden ser modificados.
- **Tuplas (tuples):** Son secuencias ordenadas de elementos que pueden ser de cualquier tipo de dato, pero que no pueden ser modificados una vez creados.
- **Sets:** Son colecciones no ordenadas de elementos únicos e inmutables.
- **Diccionarios (dictionaries):** Son estructuras que permiten almacenar pares clave-valor, donde las claves son únicas y los valores pueden ser de cualquier tipo de dato.



Listas

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo. Son mutables y dinámicas, lo cual es la principal diferencia con los sets y las tuplas.

Crear listas:

Las listas en Python son uno de los tipos o estructuras de datos más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea.

Una lista se crea con [] separando sus elementos con comas , Una gran ventaja es que pueden almacenar tipos de datos distintos.

```
>>lista = [1, "Hola", 3.67, [1, 2, 3]]
```

Algunas propiedades de las listas:

- Son ordenadas, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos arbitrarios
- Pueden ser indexadas con [i].
- Se pueden anidar, es decir, meter una dentro de la otra.
- Son mutables, ya que sus elementos pueden ser modificados.
- Son dinámicas, ya que se pueden añadir o eliminar elementos.



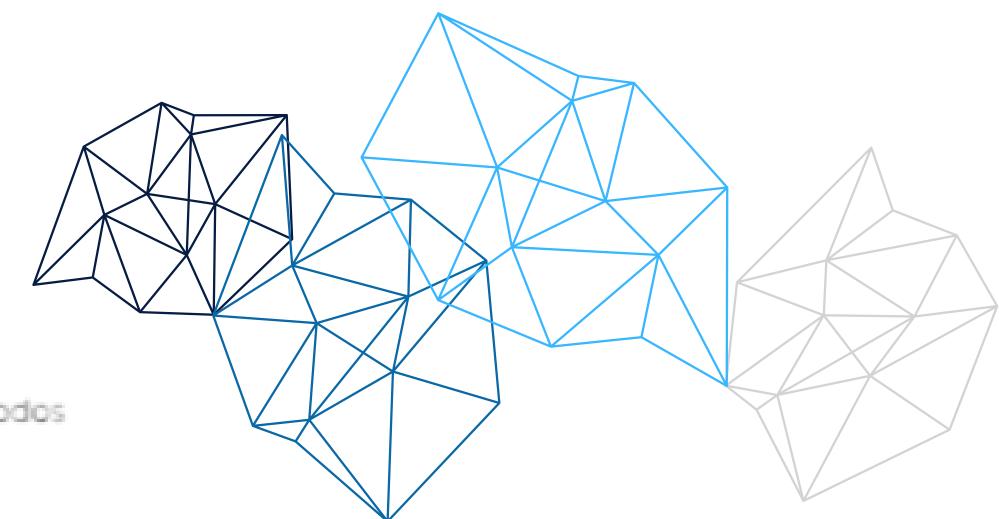
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Acceder, modificar y eliminar elementos de las listas

Si tenemos una lista `a` con 3 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde 0 a $n-1$ siendo n el tamaño de la lista.

```
>>a = [90, "Python", 3.87]
```

```
>>print(a[0]) #90
```

Se puede también acceder al último elemento usando el índice `[-1]`.

```
>>a = [90, "Python", 3.87]
```

```
>>print(a[-1]) #3.87
```

De la misma manera, al igual que `[-1]` es el último elemento, podemos acceder a `[-2]` que será el penúltimo.

```
>>print(a[-2]) #Python
```

Y si queremos modificar un elemento de la lista, basta con asignar con el operador `=` el nuevo valor.

```
>>a[2] = 1
```

```
>>print(a) #[90, 'Python', 1]
```

Un elemento puede ser eliminado con `del` y la lista con el índice a eliminar.

```
>>l = [1, 2, 3, 4, 5]
```

```
>>del l[1]
```

```
>>print(l) #[1, 3, 4, 5]
```



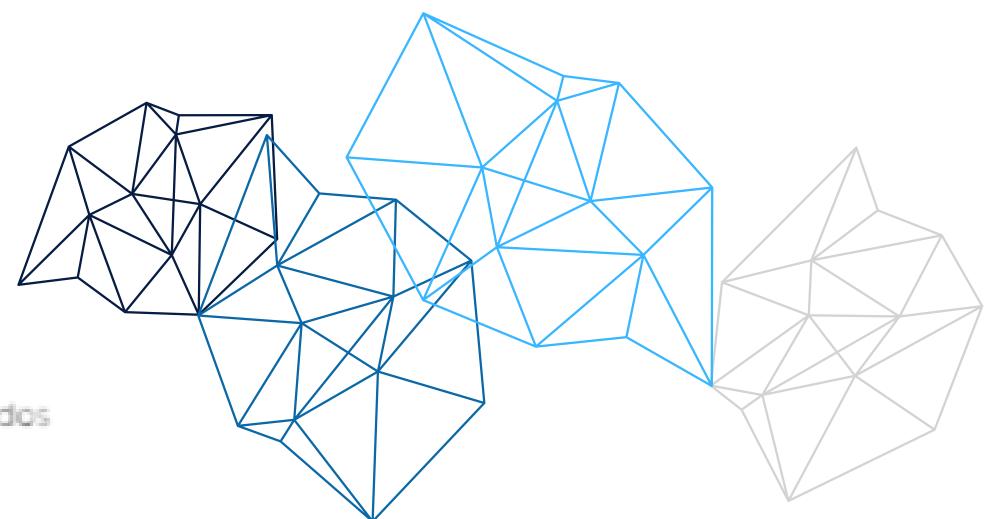
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Tuplas

Las tuplas en Python, son muy similares a las listas, la principal diferencia es que las tuplas son inmutables, es decir, no se puede modificar o reordenar su contenido. En Python los elementos de una tupla se separan por coma y pueden ir encerrados por paréntesis (No es necesario, pero muy habitual verlas de esta manera). Ejemplos de tuplas:

```
>> tupla = "a", "b", 1, 3, True
```

o lo que es igual:

```
>> tupla = ("a", "b", 1, 3, True)
```

Si queremos crear una tupla con un solo elemento, se debe poner una coma al final sino python lo va interpretar como el elemento en sí y no un elemento de una tupla

```
>> tupla = 5,  
>> print(tupla) #(5,  
>> otra_tupla = (3,)  
>> print(tupla) #(3,)
```

ejemplo de como NO:

```
>> tres = (3)  
>> print(tres) #3
```



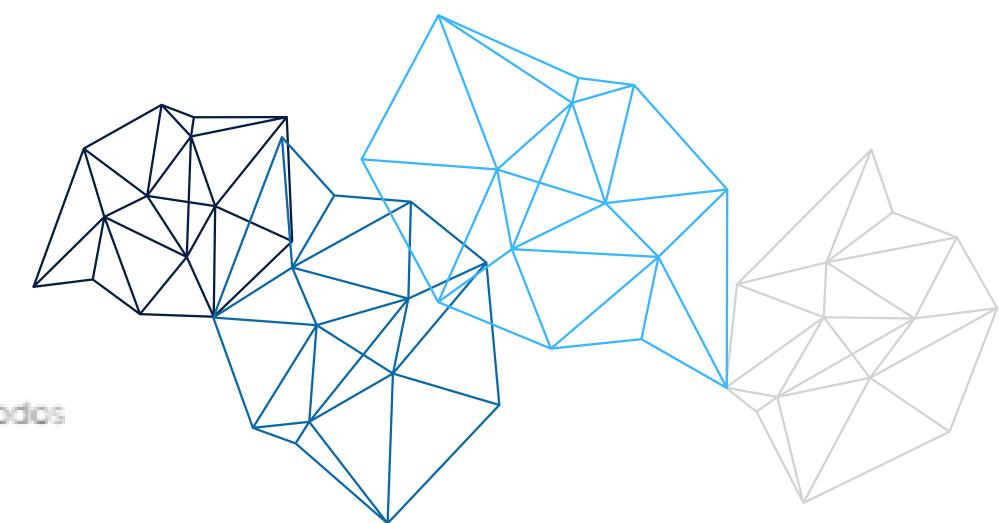
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Tuplas: Métodos

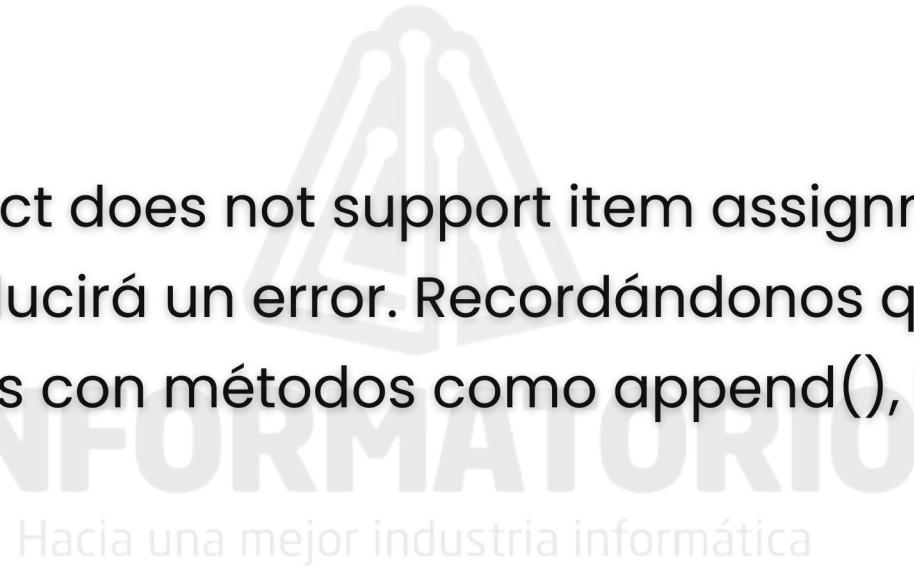
También contamos con los métodos `count()` e `index()`

```
>> colores = ("azul", "verde", "rojo", "amarillo", "azul")
>> colores.count("azul") #2
>> colores.index("amarillo") >#3
```

La principal diferencia que veremos frente a las listas será cuando tratemos modificar algún elemento de la tupla.

```
>> tupla = ("a","b","c")
>> tupla[1] = "y" #TypeError: 'tuple' object does not support item assignment
```

Tratar de modificar algún valor nos producirá un error. Recordándonos que las tuplas no son mutables, no se pueden modificar. Tampoco contaremos con métodos como `append()`, `insert()` o `sort()` que si teníamos en las listas que sí podían ser modificadas



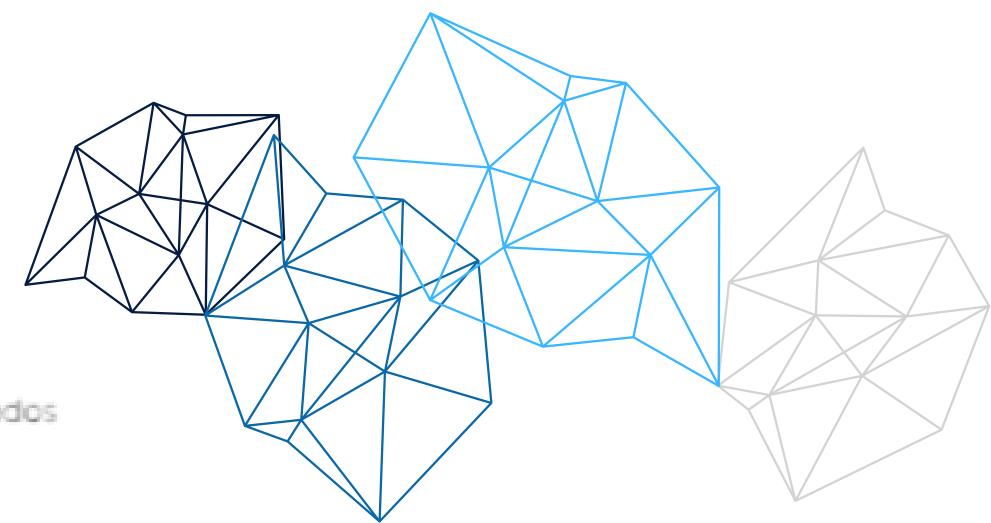
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos



Empaquetar y Desempaquetar Tuplas

Así como podemos asignar varios elementos en una tupla a una variable (empaquetar)

```
>> tupla = 10, 20, 30
```

También podemos desempaquetar una tupla en distintas variables

```
>> x, y, z = tupla #Cuanto creen que valdrá la variable "y"?
```

Para desempaquetar una tupla, la cantidad de variables tiene que ser la misma cantidad que elementos haya en la tupla de lo contrario nos dará error.

Podríamos usar tuplas por ejemplo para intercambiar valores entre variables, sin necesidad de usar una variable auxiliar: por ejemplo si queremos el valor de una variable "a" y pasarla a una variable "b", y el contenido de "b" pasarlo a "a"

usando una variable temporal se podría hacer:

```
>> temp = a  
>> a = b  
>> b = temp
```

pero con tuplas podríamos crear una tupla con los valores (a,b) y desempaquetarlo en las variables "b" y "a" de la siguiente manera:

```
>> b, a = (a, b)
```



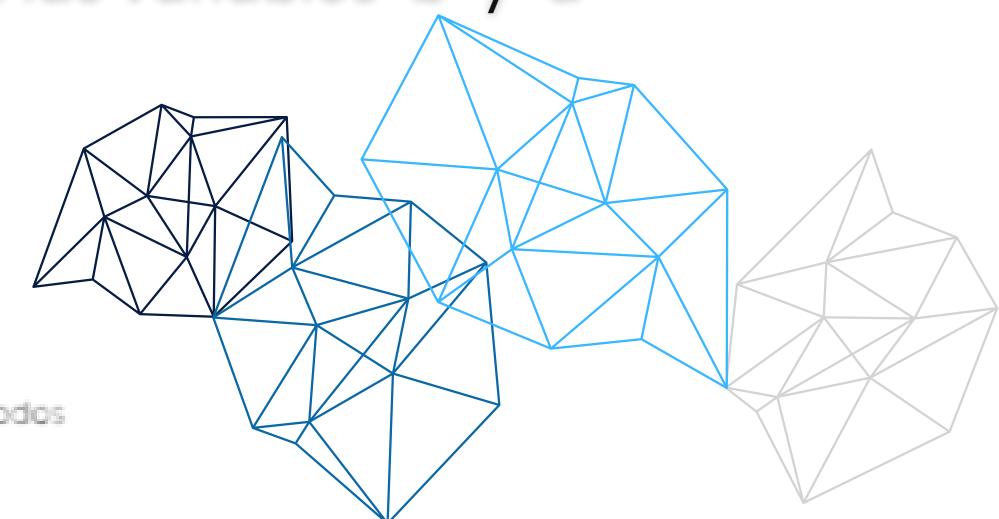
Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



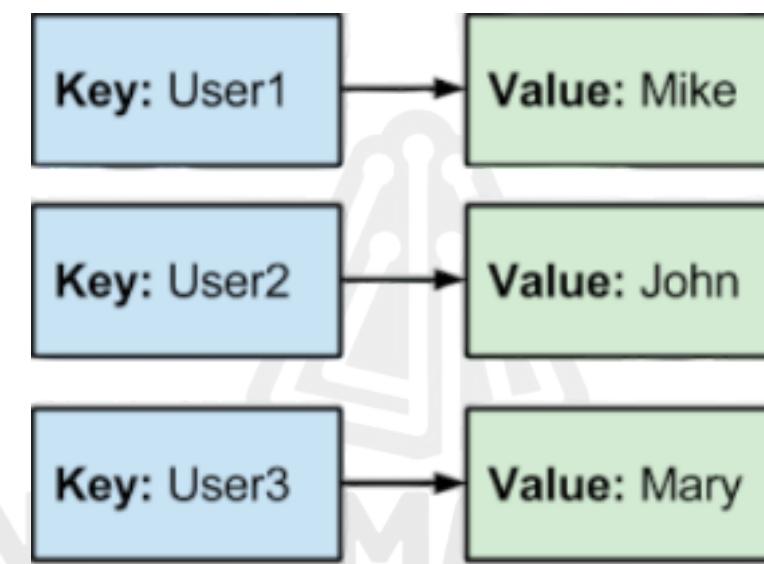
CHACO
Gobierno de todos



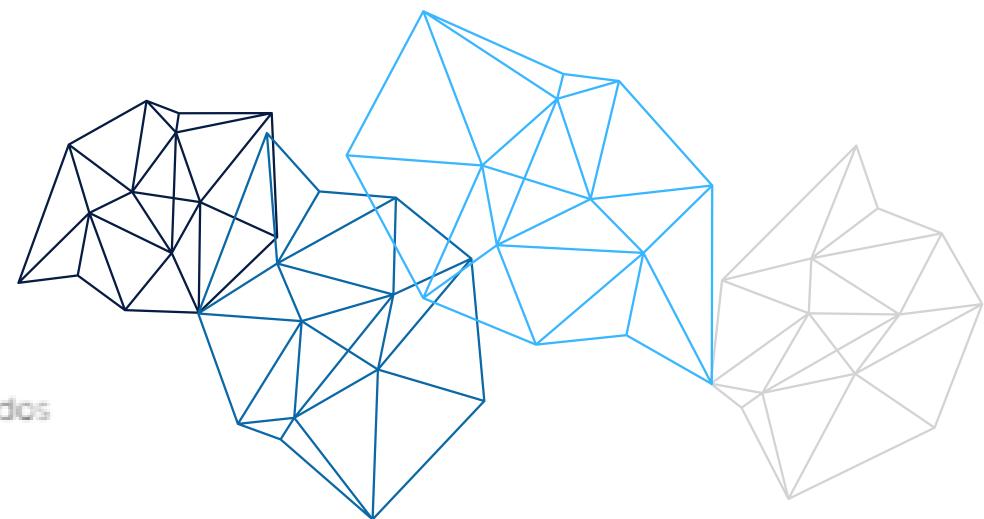
Diccionarios

En Python, un diccionario es una colección no ordenada de valores que son accedidos a través de keys (llaves o claves) a diferencia de las listas o tuplas, que se acceden a los elementos mediante el índice numérico.

Las keys o llaves deben ser únicas, si se asigna un valor a una llave ya existente, se reemplaza el valor anterior.



Los diccionarios son mutables, por lo tanto se pueden modificar, agregar o quitar elementos. Para crear diccionarios encerramos los elementos y sus keys entre { }, los elementos tendría la estructura **clave: valor** y cada uno de los elementos deben estar separados por coma. Las **keys/claves** pueden ser de cualquier tipo **inmutable**, y los **valores** pueden ser de cualquier tipo



Diccionarios

Ejemplos:

```
>>coordenadas = {"x":34, "y": 23, "z":45}  
>>alumno = {  
    "nombre": "Marcos",  
    "materias":["biología", "matemáticas", "diseño"]  
}
```

Para acceder los elementos utilizamos la clave del diccionario:

```
>> coordenadas["x"] #34
```

Al igual que con lista, podemos declarar diccionarios vacíos de dos maneras:

```
>>> diccionario = {}  
>> diccionarios = dict()
```

Para modificar un valor, de manera similar a las listas pero en lugar de usar el índice usamos la clave:

```
>>coordenadas["x"] = 36
```

Para agregar claves a un diccionario existente, realizamos la acción similar a como modificamos el valor de una clave existente pero con la nueva clave:

```
coordenadas["a"] = 70
```

Otra manera de insertar elementos es usando el método `setdefault()`, al cual se le pasa la clave y el elemento a agregar. Pero solo se agregara el elemento si la llave no existía, si ya se encontraba en el diccionario no agrega el nuevo valor.



Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos

Diccionarios: Métodos

Para recorrer un diccionario podemos hacerlo de varias maneras:

```
>> diccionario = {"a":11, "b":22, "c":33, "d":44}  
>> for key in diccionario:  
    print("la clave es: ", key)  
    print("su valor: ", diccionario[key])
```

Pero también tenemos un método `items()` en los diccionarios, que nos devuelve una lista de tuplas que contienen el par `key:value`.

```
>>print(diccionario.items())  
>>dict_items([('a', 11), ('b', 22), ('c', 33), ('d', 44)])
```

Podemos usar esto para recorrer un diccionario de la siguiente manera

```
>> for key, value in diccionario.items():  
    print(key,":",value)
```

Para verificar si una clave se encuentra en el diccionario, podemos usar el operador **`in`**

```
>>"a" in diccionario #True
```

Para eliminar algún par `key:value` de un diccionario podemos utilizar la palabra reservada **`del`**

```
>>del diccionario["a"]
```

También podemos utilizar el método `pop()` que vimos anteriormente en listas. Este método nos quitará el elemento que le indiquemos en la clave y nos retorna el valor quitado.

```
>>quitado = diccionario.pop("a") #11
```



Sets

En Python, un set es una estructura de datos mutable que representa una colección desordenada de elementos únicos y no duplicados. Los elementos de un set pueden ser de cualquier tipo de datos inmutable, como números, cadenas de texto y tuplas.

Los sets se definen utilizando llaves ({}) o la función set(). Por ejemplo:

Definición de un set con llaves :

```
>>my_set = {1, 2, 3, 4, 5}
```

Definición de un set con la función set()

```
>>my_set = set([1, 2, 3, 4, 5])
```

Es posible **agregar** elementos a un set utilizando el método add() y **eliminar** elementos utilizando el método remove(). Por ejemplo:

```
>>my_set.add(6)
```

```
>>my_set.remove(2)
```



Subsecretaría de
Empleo



Ministerio de
Producción, Industria y Empleo



CHACO
Gobierno de todos

Sets: Métodos

Los sets en Python tienen varios métodos útiles para realizar operaciones de conjunto y modificar su contenido. Algunos de los métodos más comunes son:

- `discard()`: Elimina un elemento del set. Si el elemento no existe, no arroja ninguna excepción.
- `pop()`: Elimina un elemento aleatorio del set y lo devuelve.
- `clear()`: Elimina todos los elementos del set.
- `union()`: Devuelve un nuevo set que contiene todos los elementos de los dos sets originales.
- `intersection()`: Devuelve un nuevo set que contiene los elementos que se encuentran en ambos sets originales.
- `difference()`: Devuelve un nuevo set que contiene los elementos que se encuentran en el set original pero no en el otro set.
- `symmetric_difference()`: Devuelve un nuevo set que contiene los elementos que se encuentran en uno u otro set, pero no en ambos.

En resumen, los **sets** en Python son estructuras de datos útiles para almacenar elementos únicos y realizar operaciones de conjunto como la unión, la intersección y la diferencia.

