

## Checkpoint 4

### ¿Cuál es la diferencia entre una lista y una tupla en Python?

Una tupla es una colección ordenada e inmutable de elementos. Esto significa que una vez que se crea una tupla, no puedes cambiar, agregar o eliminar sus elementos. Las tuplas se escriben con paréntesis (). Por otro lado, una lista es una colección ordenada y mutable, lo que significa que puedes modificar sus elementos después de que ha sido creada. Las listas se escriben con corchetes [].

Algunas de las diferencias entre tuplas y listas son:

**Mutabilidad:** La diferencia más notable es que las listas son mutables, mientras que las tuplas no. Puedes modificar una lista después de su creación, pero una vez que una tupla está creada, no puede ser modificada de ninguna manera.

**Uso de memoria:** Las tuplas, al ser inmutables, son generalmente más rápidas y utilizan menos memoria que las listas. Python optimiza el almacenamiento de tuplas, lo que las hace más eficientes para el acceso a los datos.

**Uso en programas:** Debido a su inmutabilidad, las tuplas son ideales cuando necesitas asegurar que los datos no se modificarán, como, por ejemplo, las claves en un diccionario de Python. Las listas son más adecuadas para datos que necesitan modificarse, como agregar o eliminar elementos.

Ejemplos:

```
Checkpoint4.py > ...
1  #Tupla:
2  coordenadas = (55.0, 15.0)
3
4  #Lista:
5  nombres = ["Melisse", "Carlos", "Enrique"]
6
```

## ¿Cuál es el orden de las operaciones?

El orden de las operaciones se sigue según la regla PEMDAS, que significa:

P: Paréntesis

E: Exponentes

M: Multiplicación

D: División

A: Suma

S: Resta

Esta regla ayuda a recordar el orden de las operaciones.

Además, los operadores en Python siguen un orden de precedencia específico.

Operadores aritméticos (\*\* es el más alto)

Operadores relacionales

Operadores lógicos

Algunos ejemplos de operadores en Python son:

Operadores aritméticos: +, -, \*, /, \*\*

Operadores de comparación: >, <, >=, <=

Operadores lógicos: not, and, or

Operador de módulo: %

Cuando aparecen más de un tipo de operador en una expresión, el procesador de lenguaje sigue una prioridad global que incluye todos los operadores.

Ejemplo:

```
resultado = 10 + 2 * 3 - 4 / 2
```

Primero, realiza la multiplicación y la división de izquierda a derecha.

$2 * 3$  es igual a 6.

$4 / 2$  es igual a 2.

Ahora, la expresión se convierte en:  $10 + 6 - 2$

Luego, realiza la suma y la resta de izquierda a derecha.

$10 + 6$  es igual a 16.

$16 - 2$  es igual a 14.

El resultado de la expresión es 14.

## ¿Qué es un diccionario Python?

Un diccionario en Python es una colección de elementos, donde cada uno tiene una llave key y un valor value. Los diccionarios se pueden crear con paréntesis `{ }` separando con una coma cada par key: value. En el siguiente ejemplo tenemos tres keys que son el nombre, la edad y el documento.

Clave sobre los diccionarios en Python:

Pares clave-valor:

Cada elemento en un diccionario consiste en una clave única y un valor asociado.

Las claves deben ser inmutables (como cadenas, números o tuplas), mientras que los valores pueden ser de cualquier tipo.

Mutables:

Los diccionarios son mutables, lo que significa que puedes agregar, eliminar o modificar elementos después de su creación.

Sintaxis:

Se definen utilizando llaves {}.

Los pares clave-valor se separan por dos puntos :.

Los elementos se separan por comas ,.

Acceso a los valores:

Puedes acceder a un valor utilizando su clave correspondiente entre corchetes [].

También puedes utilizar el método get() para acceder a los valores, lo que te permite manejar casos en los que la clave no existe.

Usos comunes:

Almacenar configuraciones.

Representar registros de datos.

Contar la frecuencia de elementos.

Crear tablas de búsqueda.

Ejemplos:

```
Checkpoint4.py > ...
1  frutas = {
2      "fresa": "rojo",
3      "limon": "amarillo",
4      "uva": "morado",
5      "naranja": "naranja"
6  }
7
8  print(frutas["fresa"]) # Imprime "rojo"
9  print(frutas["limon"]) # Imprime "amarillo"
10
11  frutas["pera"] = "verde" # Agrega un nuevo par clave-valor
12  print(frutas)
13
```

**¿Cuál es la diferencia entre el método ordenado y la función de ordenación?**

Tanto `sort()` como `sorted()` se utilizan para ordenar elementos, pero tienen diferencias clave:

`sort()`

Método:

`sort()` es un método incorporado de las listas.

Modifica la lista original "in place", es decir, la ordena directamente sin crear una nueva lista.

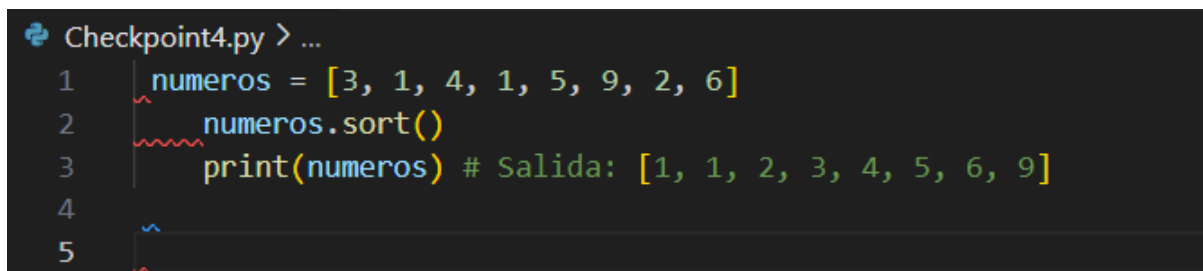
Devuelve `None`.

Uso:

Se utiliza cuando no necesitas conservar la lista original y quieres ordenarla directamente.

Es ligeramente más eficiente en términos de memoria, ya que no crea una copia de la lista.

`sorted()`

A screenshot of a Python code editor with a dark background. The code is as follows:

```
Checkpoint4.py > ...
1  numeros = [3, 1, 4, 1, 5, 9, 2, 6]
2  numeros.sort()
3  print(numeros) # Salida: [1, 1, 2, 3, 4, 5, 6, 9]
4
5
```

Función:

`sorted()` es una función incorporada de Python.

Crea una nueva lista ordenada a partir de cualquier iterable (listas, tuplas, diccionarios, etc.).

Devuelve la nueva lista ordenada.

Uso:

Se utiliza cuando necesitas conservar la secuencia original sin modificarla.

Es más flexible, ya que puede ordenar cualquier iterable.

Ejemplo:

```
Checkpoint4.py > ...  
1  
2     numeros = [3, 1, 4, 1, 5, 9, 2, 6]  
3     numeros_ordenados = sorted(numeros)  
4     print(numeros) # Salida: [3, 1, 4, 1, 5, 9, 2, 6]  
5     print(numeros_ordenados) # Salida: [1, 1, 2, 3, 4, 5, 6, 9]
```

Diferencias:

Modificación:

sort() modifica la lista original.

sorted() crea una nueva lista ordenada.

Devolución:

sort() devuelve None.

sorted() devuelve la nueva lista ordenada.

Aplicación:

sort() solo se aplica a listas.

sorted() se aplica a cualquier iterable.

## ¿Qué es un operador de reasignación?

Los operadores de asignación o assignment operators nos permiten realizar una operación y almacenar su resultado en la variable inicial. Podemos ver como realmente el único operador nuevo es el =. El resto son abreviaciones de otros operadores que habíamos visto con anterioridad.

Un operador de reasignación es una forma abreviada de realizar una operación aritmética o bit a bit y asignar el resultado a una variable al mismo tiempo.

En lugar de escribir  $x = x + 8$ , puedes escribir  $x += 8$ . Ambas expresiones hacen

lo mismo: agregan 8 al valor actual de x y almacenan el resultado nuevamente en x.

Operadores de reasignación:

`+=` (suma y asignación)

`-=` (resta y asignación)

`*=` (multiplicación y asignación)

`/=` (división y asignación)

`%=` (módulo y asignación)

`//=` (división entera y asignación)

`**=` (exponente 1 y asignación)

`&=` (AND bit a bit y asignación)

`|=` (OR bit a bit y asignación)

`^=` (XOR bit a bit y asignación)

`>>=` (desplazamiento a la derecha y asignación)

`<<=` (desplazamiento a la izquierda y asignación)

Ejemplo:

Checkpoint4.py > ...

```
1  x = 12
2  x += 8  # Equivalente a x = x + 8
3  print(x)  # Imprime 20
4
```