

1 - Las Listas en Python son colecciones mutables, lo que significa que sus elementos pueden ser modificados después de su creación (agregar, eliminar o cambiar valores).

- Se crean utilizando corchetes []. Ejemplo: `mi_lista = [1][2][3]`.
- Adecuadas para casos donde los datos almacenados necesitan cambiar durante la ejecución del programa.
- Permiten añadir elementos con `.append()`, eliminar con `.remove()` o `.pop()`, y ordenar con `.sort()`.

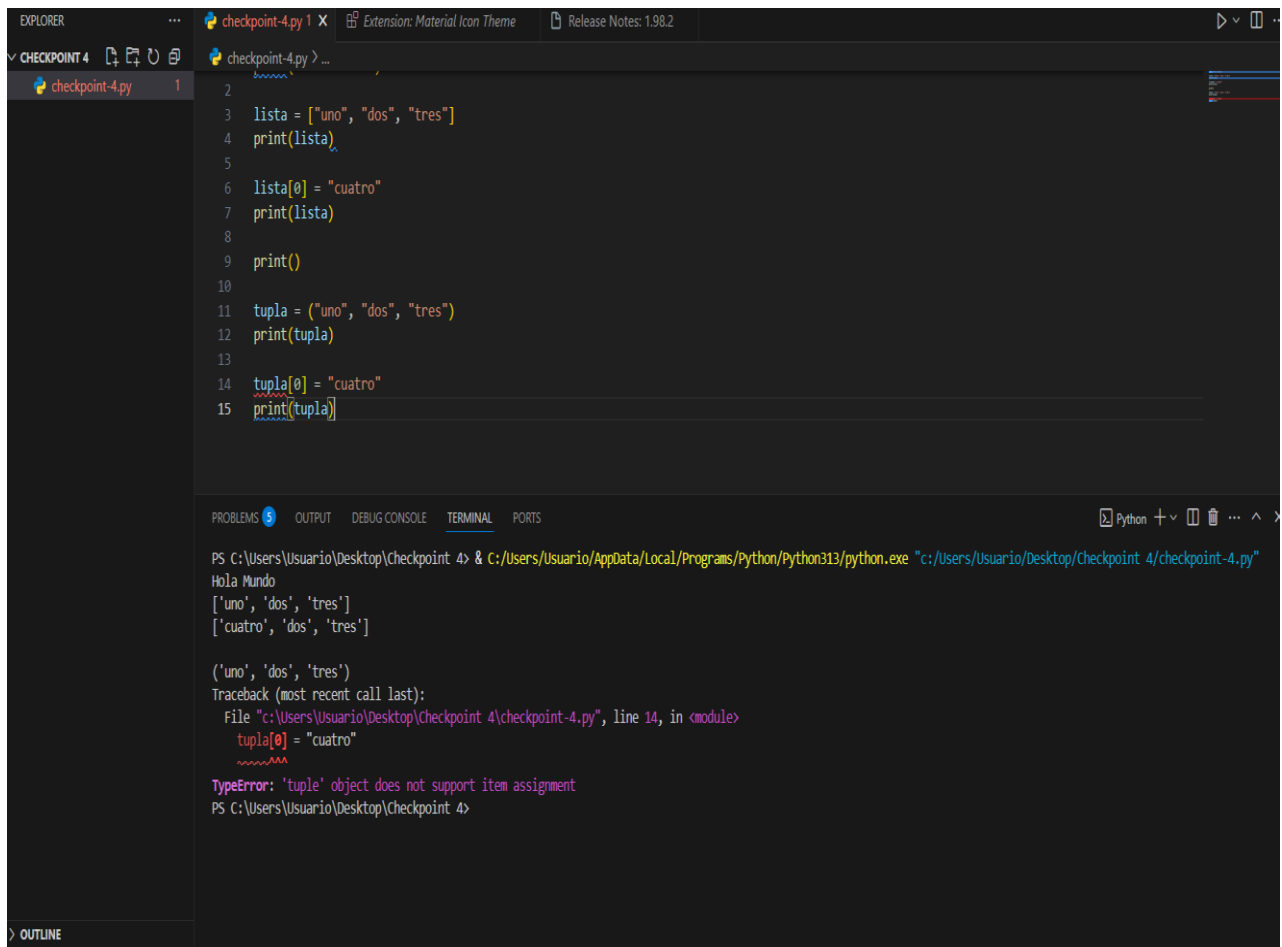
Las tuplas son colecciones inmutables, lo que significa que no se pueden modificar después de su creación.

- Se crean utilizando paréntesis (). Ejemplo: `mi_tupla = (1, 2, 3)`. También pueden crearse sin paréntesis separando los elementos con comas. Para una tupla de un solo elemento, se debe incluir una coma: `(1,)`.
- Son ideales para almacenar datos que no deben cambiar, como configuraciones o constantes.
- Al ser inmutables, son más rápidas y consumen menos memoria en comparación con las listas

La principal diferencia entre las listas y las tuplas de Python, y el motivo por el que muchos usuarios solamente utilizan listas, es que las listas son mutables mientras que las tuplas no lo son. Básicamente un objeto mutable se puede modificar una vez creado mientras que uno que no lo es no. Así el contenido de las listas se puede modificar durante la ejecución del programa mientras para las tuplas no es posible alterar su contenido.

El hecho de ser mutable tiene además otras consecuencias. Para ser mutables las listas se almacena en dos bloques de memoria, mientras que las tuplas solo necesitan uno. Lo que provoca que las tuplas ocupen menos memoria que las listas. Además, por el hecho de no ser mutables, es más rápido manejar tuplas que listas.

En este ejemplo podemos ver como conseguimos modificar la Lista, pero cuando lo intentamos hacer con Tupla nos da error.



The screenshot shows a Visual Studio Code editor with a file named 'checkpoint-4.py'. The code in the editor is as follows:

```
1
2
3 lista = ["uno", "dos", "tres"]
4 print(lista)
5
6 lista[0] = "cuatro"
7 print(lista)
8
9 print()
10
11 tupla = ("uno", "dos", "tres")
12 print(tupla)
13
14 tupla[0] = "cuatro"
15 print(tupla)
```

The terminal output at the bottom shows the execution of the script. It prints the list and tuple, then attempts to modify the tuple, resulting in a `TypeError: 'tuple' object does not support item assignment`.

```
PS C:\Users\Usuario\Desktop\Checkpoint 4> & C:\Users\Usuario\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Usuario/Desktop/Checkpoint 4/checkpoint-4.py"
Hola Mundo
['uno', 'dos', 'tres']
['cuatro', 'dos', 'tres']

('uno', 'dos', 'tres')
Traceback (most recent call last):
  File "c:/Users/Usuario/Desktop/Checkpoint 4/checkpoint-4.py", line 14, in <module>
    tupla[0] = "cuatro"
    ~~~~~^
TypeError: 'tuple' object does not support item assignment
PS C:\Users\Usuario\Desktop\Checkpoint 4>
```

2 - En Python, el orden de las operaciones sigue reglas específicas conocidas como precedencia de operadores. Este orden determina cómo se evalúan las expresiones complejas. A continuación, se detalla el orden desde la mayor prioridad hasta la menor:

- Paréntesis (`()`): Todo lo que está dentro de paréntesis se evalúa primero.
- Exponentes (`**`): La potencia tiene la segunda prioridad.
- Signos unarios (`+x`, `-x`, `~x`): Operaciones como negación o cambio de signo.

- Multiplicación, División, División Entera y Módulo (\*, /, //, %): Estas operaciones tienen la misma prioridad y se evalúan de izquierda a derecha.
- Adición y Sustracción (+, -): Se evalúan después de las operaciones anteriores.
- Operadores Relacionales y Comparativos (<, <=, >, >=, ==, !=): Comparaciones entre valores.
- Operadores Lógicos:
  - not: Tiene la mayor prioridad entre los operadores lógicos.
  - and: Evaluado después de not.
  - or: Evaluado al final.

3 - Los diccionarios nos permiten asignar valores a ciertas palabras clave y luego recuperar esos valores más adelante mediante la referencia a las palabras clave. Esto es particularmente útil cuando se trabaja con conjuntos de datos complejos que requieren un almacenamiento específico, o cuando se necesita realizar búsquedas rápidas de valores específicos.

En términos de sintaxis, los diccionarios se definen utilizando llaves en lugar de corchetes. Por ejemplo, supongamos que queremos crear un diccionario de nombres de personas y sus edades:

```
personas = {'Juan': 31, 'Pedro': 25, 'Maria': 42}
```

En este caso, las llaves ('Juan', 'Pedro', 'Maria') son las palabras clave y los valores (31, 25, 42) son las edades correspondientes de las personas.

Una vez que tenemos nuestro diccionario definido, podemos acceder a los valores correspondientes de varias maneras. Por ejemplo, para obtener la edad de Juan, simplemente tenemos que escribir:

```
print(personas['Juan'])
```

Esto imprimirá el valor 31 en la consola.

En un diccionario, las **claves** son únicas e inmutables, lo que significa que no pueden ser duplicadas o borradas, mientras que los **valores** pueden ser de cualquier tipo de datos y, a diferencia de las claves, pueden ser modificados.

La sintaxis básica para declarar un diccionario es la siguiente:

```
mi_diccionario = {'clave1': valor1, 'clave2': valor2, 'clave3': valor3}
```

En este ejemplo, se creó un diccionario llamado `mi_diccionario` con tres pares clave-valor. Podemos acceder a estos valores a partir de sus claves de esta manera:

```
print(mi_diccionario['clave1'])
```

La salida sería el valor que corresponde a la clave 'clave1'.

Podemos agregar elementos a un diccionario de esta manera:

```
mi_diccionario['nueva_clave'] = nuevo_valor
```

También podemos modificar el valor de cualquier clave existente en un diccionario de esta manera:

```
mi_diccionario['clave1'] = nuevo_valor
```

Y finalmente, para borrar una clave y su valor asociado de un diccionario, podemos utilizar el método **del** de esta manera:

```
del mi_diccionario['clave1']
```

Los diccionarios también pueden ser utilizados en combinación con otros conceptos de Python, como las funciones. Por ejemplo, podemos crear una función que tenga como argumento un diccionario y que devuelva el valor correspondiente a una clave específica:

```
def consulta_valor(diccionario, clave):  
    return diccionario[clave]
```

De esta manera, si tenemos un diccionario llamado **mi\_diccionario** que contenga una clave 'clave3' con el valor 10, podemos llamar a esta función de la siguiente manera:

```
valor = consulta_valor(mi_diccionario, 'clave3')  
print(valor)
```

La salida en este caso sería el valor 10.

En Python, los **diccionarios** son una estructura de datos muy útil para almacenar información de manera organizada y flexible. Una de las ventajas principales de los diccionarios es que se pueden acceder a sus valores utilizando su clave en lugar de su posición en la estructura, lo cual facilita enormemente la búsqueda y manipulación de los datos.

Para acceder a los valores de un diccionario mediante su clave, se utiliza la notación de corchetes []. Por ejemplo, si tenemos un diccionario que contiene la información de una persona:

```
persona = {'nombre': 'Juan', 'edad': 25, 'altura': 1.75}
```

Podemos acceder al valor correspondiente a la clave 'nombre' de la siguiente manera:

```
nombre = persona['nombre']  
print(nombre) # Output: 'Juan'
```

También es posible modificar los valores de un diccionario utilizando su clave:

```
persona['edad'] = 26  
print(persona['edad']) # Output: 26
```

O añadir nuevos pares clave-valor al diccionario:

```
persona['peso'] = 70  
print(persona) # Output: {'nombre': 'Juan', 'edad': 26, 'altura': 1.75, 'peso': 70}
```

Es importante destacar que cada clave en un diccionario debe ser única, ya que no puede haber dos valores asociados a la misma clave. Si intentamos añadir un nuevo valor para una clave ya existente, el valor anterior será sobrescrito:

```
persona['edad'] = 27  
print(persona) # Output: {'nombre': 'Juan', 'edad': 27, 'altura': 1.75, 'peso': 70}
```

También es posible verificar si una clave determinada existe en un diccionario utilizando el operador 'in':

```
if 'nombre' in persona:  
    print('La clave "nombre" existe en el diccionario')  
else:  
    print('La clave "nombre" no existe en el diccionario')
```

Los diccionarios son una de las estructuras de datos más utilizadas en Python, y en general en programación, debido a su flexibilidad y facilidad de uso. Una de las características más importantes de los diccionarios en Python es que son mutables, lo que significa que se pueden agregar, modificar o eliminar elementos fácilmente.

Personalmente, hemos utilizado los diccionarios en varios proyectos en los que necesitábamos almacenar información compleja en una estructura fácilmente accesible. Por ejemplo, en un proyecto para una tienda en línea, almacenamos la información del catálogo de productos en un diccionario, donde cada clave era el nombre del producto y cada valor era un objeto que contenía la descripción, el precio, el inventario disponible y otros detalles relevantes.

La mutabilidad de los diccionarios es especialmente útil cuando se necesita actualizar la información en tiempo real. Por ejemplo, si el inventario de un producto cambia, podemos actualizar fácilmente el valor correspondiente en el diccionario. También podemos agregar nuevos productos o eliminar productos obsoletos con facilidad.

Para agregar un nuevo elemento a un diccionario, simplemente especificamos una nueva clave y su correspondiente valor. Por ejemplo, si nuestro diccionario de productos aún no incluye “Taza de café con logotipo”, podemos agregarla con el siguiente código:

```
catalogo_productos = {  
    "Camiseta de algodón": {"descripcion": "Camiseta cómoda de algodón", "precio": 20, "inventario": 50},  
    "Sudadera con capucha": {"descripcion": "Sudadera suave y abrigadora", "precio": 40, "inventario": 30},  
    "Paquete de stickers": {"descripcion": "Stickers divertidos para decorar laptops, cuadernos, etc.", "precio": 5, "inventario": 100}  
}  
  
catalogo_productos["Taza de café con logotipo"] = {"descripcion": "Taza de cerámica con el logotipo de la tienda", "precio": 10, "inventario": 20}
```

En este ejemplo, hemos agregado una nueva clave "Taza de café con logotipo" con un diccionario como valor que contiene la descripción, el precio y el inventario.

Para modificar un elemento existente en un diccionario, simplemente asignamos un nuevo valor a la clave correspondiente. Por ejemplo, si queremos actualizar el precio de las camisetas a \$25, podemos hacer lo siguiente:

```
catalogo_productos["Camiseta de algodón"]["precio"] = 25
```

Este código actualiza el valor correspondiente al precio de las camisetas en el diccionario de productos.

Para trabajar con diccionarios en Python, es importante saber cómo utilizar la función **len**. Esta función devuelve la cantidad de elementos que contiene el diccionario. Por ejemplo, si queremos saber cuántos elementos hay en el siguiente diccionario:

```
mi_dict = {'nombre': 'Juan', 'edad': 25, 'ciudad': 'Mexico'}
```

Simplemente podemos utilizar la función **len**:

```
print(len(mi_dict))
```

La salida de este código sería 3, porque el diccionario `mi_dict` contiene tres elementos.

También se puede utilizar la función `len` en un diccionario vacío. Por ejemplo:

```
mi_dict_vacio = {}  
print(len(mi_dict_vacio))
```

La salida de este código sería 0, ya que el diccionario `mi_dict_vacio` no contiene elementos.

Es importante notar que la función `len` solo devuelve la cantidad de elementos en el diccionario, no la cantidad de pares clave-valor. Por ejemplo, si tenemos un diccionario como el siguiente:

```
mi_dict = {'a': 1, 'b': 2, 'c': 3, 'd': {'e': 4, 'f': 5}}
```

La función `len` devolverá 4, ya que el diccionario contiene cuatro elementos. No importa que un par clave-valor dentro del diccionario sea en sí mismo otro diccionario (`{'e': 4, 'f': 5}`).

En el mundo de la programación, la manipulación de datos es una tarea imprescindible. Y una herramienta que resulta muy útil para este propósito son los diccionarios en Python.

Particularmente, en el desarrollo de un proyecto, en ocasiones es necesario almacenar grandes cantidades de datos y operar con ellos de manera eficiente, rápida y sencilla. Afortunadamente Python nos provee de diversas estructuras para lograrlo, y entre ellas destacan los diccionarios.

Un diccionario en Python es una estructura que permite almacenar datos de una forma muy particular. Funciona similar a un diccionario de la vida real, donde cada elemento se encuentra asociado con una palabra clave o llave. Dicha llave sirve para identificar y acceder a un valor en particular.



En resumen, un diccionario está conformado por pares clave-valor, donde cada llave puede asociarse con un valor determinado. Así, los diccionarios en Python resultan muy útiles para almacenar grandes cantidades de datos que puedan ser accedidos de forma más rápida y ordenada.

Existen muchas formas de manipular y operar sobre los datos almacenados en un diccionario utilizando Python. Por ejemplo, se puede acceder a un valor específico utilizando la llave correspondiente, o bien, utilizar una estructura de control como los ciclos para acceder iterativamente a todos los elementos.

Otra de las grandes ventajas de los diccionarios es que permiten la incorporación de datos de manera sencilla y eficiente. Además, Python provee una gran variedad de funciones y métodos para la manipulación y modificación de diccionarios.

4 - En Python, la diferencia entre el método sort y la función sorted radica principalmente en cómo operan y en su ámbito de aplicación.

- El método sort ordena los elementos de la lista en su lugar, es decir, altera directamente la lista original y no retorna ningún valor, y Solo funciona con listas

```
mi_lista = [5, 2, 3, 1, 4]
mi_lista.sort()
print(mi_lista) # Output: [1, 2, 3, 4, 5]
```

- La función Sorted retorna una nueva lista ordenada sin alterar el iterable original y puede trabajar con listas, tuplas, cadenas y otros tipos de iterables.

```
mi_lista = [5, 2, 3, 1, 4]
nueva_lista = sorted(mi_lista)
print(nueva_lista) # Output: [1, 2, 3, 4, 5]
print(mi_lista)    # Output: [5, 2, 3, 1, 4] (sin cambios)
```

Aspecto	sort()	sorted()
Modifica el original	Sí	No
Retorno	None	Nueva lista ordenada
Tipo de entrada	Solo listas	Cualquier iterable
Eficiencia	Más eficiente para listas	Menos eficiente (crea copia)

5 - Los operadores de reasignación en Python permiten realizar una operación entre una variable y un valor (o expresión) y asignar el resultado nuevamente a la misma variable. Estos operadores son una forma abreviada de escribir operaciones comunes y hacen el código más legible y eficiente.

```
contador = 10

# Ejemplo de uso de operadores de reasignación

contador += 5 # Equivalente a contador = contador + 5
print(contador) # Salida: 15

contador *= 2 # Equivalente a contador = contador * 2
print(contador) # Salida: 30

contador //= 3 # Equivalente a contador = contador // 3
print(contador) # Salida: 10
```

Operador	Ejemplo	Equivalente	Descripción
=	<code>x = 7</code>	<code>x = 7</code>	Asigna un valor a una variable.
+=	<code>x += 2</code>	<code>x = x + 2</code>	Suma y asigna el resultado.
-=	<code>x -= 2</code>	<code>x = x - 2</code>	Resta y asigna el resultado.
*=	<code>x *= 2</code>	<code>x = x * 2</code>	Multiplica y asigna el resultado.
/=	<code>x /= 2</code>	<code>x = x / 2</code>	Divide y asigna el resultado.
%=	<code>x %= 2</code>	<code>x = x % 2</code>	Calcula el módulo y asigna el resultado.
//=	<code>x //= 2</code>	<code>x = x // 2</code>	Realiza la división entera y asigna el resultado.
**=	<code>x **= 2</code>	<code>x = x ** 2</code>	Calcula la potencia y asigna el resultado.
&=	<code>x &amp;= 2</code>	<code>x = x &amp; 2</code>	Realiza una operación AND bit a bit y asigna el resultado.
`	<code>=`</code>	<code>`x</code>	<code>= 2`</code>
^=	<code>x ^= 2</code>	<code>x = x ^ 2</code>	Realiza una operación XOR bit a bit y asigna el resultado.
>>=	<code>x &gt;&gt;= 2</code>	<code>x = x &gt;&gt; 2</code>	Desplaza los bits a la derecha y asigna el resultado.
<<=	<code>x &lt;&lt;= 2</code>	<code>x = x &lt;&lt; 2</code>	Desplaza los bits a la izquierda y asigna el resultado.

El uso de operadores de reasignación nos permite reducir la longitud del código al evitar escribir expresiones completas, nos facilita la lectura del código al expresar claramente la intención de modificar la variable en lugar de repetirla y también son útiles para operaciones repetitivas sobre una misma variable, especialmente en bucles.

