ChatGPT

# Estructuras de datos en Python

## ¿Qué son las estructuras de datos?

Las **estructuras de datos** son formas de organizar y representar la información de manera estructurada dentro de un programa [1]. En otras palabras, proporcionan formatos específicos para almacenar datos, con el fin de poder acceder y manipular esa información de forma eficiente en la computadora [2]. Por ejemplo, una lista de números, un conjunto de valores únicos o una colección de pares clave-valor son distintas estructuras de datos. Cada estructura de datos define cómo se organizan los datos en memoria y qué operaciones están disponibles para interactuar con ellos.

## ¿Para qué se utilizan las estructuras de datos?

El uso de estructuras de datos adecuadas permite **almacenar y procesar información de manera eficiente** según la naturaleza del problema [2]. Al elegir la estructura correcta, podemos resolver problemas computacionales de forma más sencilla y óptima. Por ejemplo, si necesitamos registrar elementos y luego obtener rápidamente el último agregado, conviene usar una estructura tipo *pila* (LIFO) para apilar y desapilar elementos. En Python, una lista puede funcionar como pila: podemos añadir elementos al final con `append()` y retirarlos con `pop()`, obteniendo siempre el último elemento insertado [3] [4]. En general, las estructuras de datos se utilizan para modelar datos del mundo real (listas de usuarios, conjuntos de resultados únicos, relaciones de mapeo entre claves y valores, etc.) de una forma que haga eficientes las operaciones típicas (búsquedas, inserciones, recorridos, etc.).

## Tipos de estructuras de datos en Python

Python proporciona varias estructuras de datos integradas muy útiles. Las cuatro fundamentales son: **listas**, **tuplas**, **conjuntos (sets)** y **diccionarios (dict)** [5]. Cada una tiene características particulares que la hacen adecuada para ciertos escenarios. A continuación, describimos cada tipo:

### Listas (`list`)

Una *lista* es una colección **ordenada** de elementos, delimitada por corchetes (`[ ]`). Es una estructura **mutable**, lo que significa que sus elementos se pueden modificar, agregar o eliminar. Las listas aceptan elementos de cualquier tipo (incluso combinados) y permiten elementos duplicados. Por ejemplo, `["manzana", 42, True]` es una lista con una cadena, un entero y un booleano. Las listas son muy flexibles: no requieren que todos sus elementos sean del mismo tipo y pueden cambiar de tamaño dinámicamente [6]. Además, mantienen el **orden de inserción** de los elementos; cada elemento ocupa una posición (índice) en la lista. Python usa **indexación base 0**, es decir, el primer elemento está en el índice 0, el segundo en el 1, y así sucesivamente [7]. También soportan **índices negativos** para acceder desde el final (índice -1 es el último elemento, -2 el penúltimo, etc.) [7].

Las listas son ideales para colecciones de tamaño variable donde el orden importa o se van a recorrer frecuentemente los elementos. Por ser mutables, podemos modificarlas en el lugar sin crear nuevas estructuras. Python ofrece operaciones poderosas sobre listas, como el *rebanado* (*slicing*) para obtener sublistas, y las **comprensiones de listas** (*list comprehensions*) para crear listas derivadas de manera

concisa. Por ejemplo, una comprensión `[x**2 for x in range(5)]` genera la lista de cuadrados `[0, 1, 4, 9, 16]` en una sola expresión.

### Tuplas (`tuple`)

Una *tupla* es muy similar a una lista en cuanto a que es una **secuencia ordenada** de elementos, pero con la diferencia clave de que es **inmutable**. Esto significa que, una vez creada la tupla, sus elementos no pueden modificarse (no es posible agregar, cambiar o eliminar elementos individualmente). Se definen con paréntesis `()` (o incluso sin paréntesis, separando los elementos por comas). Por ejemplo, `(3, "hola", True)` es una tupla de tres elementos. Las tuplas suelen usarse para agrupar datos heterogéneos que van juntos y no necesitan cambiar, como por ejemplo las coordenadas de un punto `(x, y)` o los datos retornados por una función. Al ser inmutables, también son **hashables**, lo que permite usar tuplas como claves en diccionarios o elementos en conjuntos (si sus contenidos internos también son inmutables).

Aunque en apariencia las tuplas se parecen a las listas, se utilizan en escenarios distintos. Típicamente, las tuplas contienen una colección *heterogénea* de elementos con significado fijo en cada posición, mientras que las listas suelen ser colecciones *homogéneas* o de tamaño variable [8]. Por ejemplo, podríamos representar la información de una persona con una tupla (`(nombre, edad, ciudad)`), ya que esas propiedades siempre irán juntas y no se requiere modificarlas individualmente, mientras que una lista podría usarse para una serie de números cuya cantidad puede crecer o cambiar. Debido a su inmutabilidad, intentar asignar o modificar un elemento de una tupla producirá un error de tipo (`TypeError`) en tiempo de ejecución [9]. En resumen, las tuplas se usan para **datos que no deben cambiar** durante la ejecución del programa y para representar estructuras estáticas de datos.

### Conjuntos (`set`)

Un *conjunto* es una colección de elementos **únicos** y **no ordenados**. Se define con llaves `{ }` (por ejemplo: `{"apple", "banana", "cherry"}`) o usando la función `set()`. A diferencia de listas y tuplas, un set **no mantiene un orden particular** de los elementos, y además **no permite duplicados**: cualquier elemento aparece a lo sumo una vez [10]. Los conjuntos son útiles cuando necesitamos probar rápidamente la pertenencia de un elemento (opera de forma similar a un conjunto matemático) o eliminar duplicados de una colección [10]. Internamente, los sets están implementados mediante tablas hash, lo que les da una eficiencia muy alta en las operaciones de **búsqueda**, inserción o eliminación (en promedio, tiempo constante *O(1)* para estas operaciones).

En Python, podemos realizar operaciones de teoría de conjuntos con este tipo: **unión**, **intersección**, **diferencia**, etc., ya sea mediante operadores (`|`, `&`, `-`, `^`) o métodos equivalentes (`set.union()`, `set.intersection()`, etc.) [11] [12]. Por ejemplo, `{1,2,3} | {3,4,5}` resulta en `{1,2,3,4,5}` (unión de conjuntos). Cabe destacar que, como los conjuntos no tienen un orden definido, no se accede a sus elementos por índice (no existe algo como `mi_set[0]`); en su lugar, se comprueba si un valor pertenece o no al set mediante el operador `in`. Así, `"banana" in mi_set` devuelve `True` si `"banana"` es un elemento del conjunto, o `False` en caso contrario [13]. *Nota:* para crear un conjunto vacío, se debe usar `set()`, ya que literal `{}` crea un **diccionario** vacío (no un set) [14].

### Diccionarios (`dict`)

Un *diccionario* es una estructura de datos que almacena **pares de clave y valor**. Funciona como una especie de mapa: cada *clave* única se asocia a un *valor*. Por ejemplo, un diccionario podría ser

`{"nombre": "Ana", "edad": 30, "ciudad": "Madrid"}`, donde las claves son `"nombre"`, `"edad"`, `"ciudad"`, y sus respectivos valores `"Ana"`, `30`, `"Madrid"`. A diferencia de las secuencias (listas o tuplas) que usan índices numéricos, en un diccionario se accede a los elementos mediante la clave asociada [15]. Las claves suelen ser de tipo inmutable (por ejemplo, cadenas, números o tuplas de inmutables) y **deben ser únicas** dentro del diccionario [15]. Los valores, en cambio, pueden ser de cualquier tipo (mutable o inmutable) y pueden repetirse.

Los diccionarios son muy eficientes para buscar y recuperar valores a partir de su clave, también gracias a una implementación interna basada en tablas hash. Desde Python 3.7, además, los diccionarios preservan el **orden de inserción** de las claves (es decir, iterar por un diccionario recorrerá las claves en el orden en que fueron añadidas). Para crear un diccionario vacío se usan llaves vacías `{}`. Es posible inicializar un diccionario con elementos colocando pares `clave: valor` separados por comas dentro de las llaves [16]. Por ejemplo: `inventario = {"manzanas": 50, "naranjas": 20}`. En contexto de uso, los diccionarios son ideales para representar estructuras tipo "clave -> valor": por ejemplo, un directorio telefónico (clave = nombre, valor = número), conteo de ocurrencias (clave = elemento, valor = frecuencia) o cualquier estructura de datos donde queramos acceder a un valor especificado a través de un identificador en lugar de mediante una posición numérica.

## ¿Qué problemas se pueden resolver con cada estructura?

Cada estructura de datos es más adecuada para ciertos tipos de problemas. A continuación se resumen *pautas generales* sobre **cuándo conviene usar listas, tuplas, conjuntos o diccionarios** en Python:

- **Listas:** son recomendables cuando necesitas una colección **ordenada y mutable**; por ejemplo, *"usa listas cuando el orden importa y necesitas modificar elementos"* [17]. Si vas a recorrer secuencialmente datos que pueden cambiar (añadir/quitar elementos) o cuyo orden intrínseco es significativo, una lista es la opción por defecto. Ejemplo: una lista para los productos en el carrito de compras de un usuario, donde el orden de añadidura se mantiene y se pueden agregar o quitar productos.

- **Tuplas:** úsalas cuando requieres una **secuencia inmutable** de datos, típicamente de diferentes tipos, que no necesite cambios. En resumen, *"usa tuplas cuando necesitas una secuencia inmutable (ej: valores de retorno de funciones)"* [18]. Son útiles para *paquetes* de datos constantes, como coordenadas fijas, configuraciones o registros que no van a modificarse. También cuando quieres asegurar que una colección de valores permanezca intacta (p. ej., las constantes de configuración de una aplicación, o los días de la semana).

- **Conjuntos:** ideales para colecciones **no ordenadas de elementos únicos**, donde interese sobre todo la pertenencia y evitar duplicados. En otras palabras, *"usa conjuntos cuando necesitas búsquedas rápidas y unicidad"* [19]. Si necesitas comprobar rápidamente si un elemento ya existe (como palabras ya vistas, IDs usados, etc.) o modelar operaciones de conjuntos matemáticos, esta es la estructura apropiada. Ejemplo: un conjunto para almacenar los IDs únicos de usuarios activos en un sistema, permitiendo verificar instantáneamente si un ID dado está activo o no.

- **Diccionarios:** son la mejor elección cuando se necesita una **asociación rápida de clave a valor**, es decir, *"usa diccionarios cuando mapeas claves a valores de forma eficiente"* [20]. Siempre que tengas datos etiquetados o estructurados (estilo JSON), un diccionario permite acceder al valor mediante una clave descriptiva en lugar de un índice. Ejemplo: un diccionario para guardar las preferencias de configuración de un usuario, donde cada opción tiene un nombre (clave) y un valor correspondiente.

En resumen, la elección de la estructura correcta facilita la representación natural del problema: por ejemplo, una cola de tareas podría implementarse con una lista (o mejor, con *deque* para mayor eficiencia en pop/append por ambos extremos), un registro inmutable como los datos de una persona podría representarse con una tupla, un conjunto sirve para agrupar elementos únicos como palabras en un documento para eliminar duplicados, y un diccionario sirve para relaciones directas como traducciones de palabras (clave en un idioma, valor en otro).

## Creación y acceso a elementos de una estructura

**Sintaxis de creación:** Cada estructura tiene su forma literal de ser creada en Python:

```
# Ejemplos de creación de estructuras de datos:
lista = [1, 2, 3]                   # Lista con tres enteros
tupla = (1, 2, 3)                   # Tupla con tres enteros
conjunto = {1, 2, 3}                # Conjunto con tres enteros
diccionario = {"uno": 1, "dos": 2}  # Diccionario con dos pares clave-valor
```

Como se ve, las listas usan corchetes `[ ]`, las tuplas paréntesis `( )` (aunque pueden crearse tuplas sin paréntesis usando comas, o con `tuple()`), los conjuntos usan llaves `{ }`, y los diccionarios llaves con pares `clave:valor`. **Importante:** `{}` crea un diccionario vacío; para crear un conjunto vacío se debe usar `set()` [14].

**Acceso a los elementos:** Una vez creada la estructura, necesitamos formas de acceder a sus datos:

- En **listas y tuplas**, los elementos se acceden por **índice numérico**. Se escribe el nombre de la estructura seguido del índice entre corchetes, por ejemplo `mi_lista[2]` devuelve el tercer elemento (recordando que el índice inicia en 0) [7]. Si se usa un índice fuera del rango válido, Python lanza un error `IndexError`. Las listas y tuplas también admiten *slicing* (rebanado) usando rangos de índices `inicio:fin` para obtener una sub-secuencia de elementos.

- En **diccionarios**, se accede a cada valor mediante su **clave**. En lugar de posiciones numéricas, usamos la sintaxis `mi_diccionario[clave]` para obtener el valor asociado a esa clave [21]. Por ejemplo, dada `persona = {"nombre": "Ana", "edad": 30}`, `persona["edad"]` devolverá `30`. Si la clave no existe, se lanza un `KeyError`. Para evitar el error, se puede usar `mi_diccionario.get(clave)` que retorna `None` u otro valor por defecto si la clave no está presente [22].

- En **conjuntos**, al no haber indices, no existe acceso posicional. La manera típica de *acceder* o comprobar elementos es mediante operaciones de pertenencia: por ejemplo, usando `x in mi_conjunto` para saber si `x` está en el set (True/False). También se puede iterar sobre el conjunto para recorrer sus elementos (en orden arbitrario). No obstante, no podemos hacer algo como `mi_conjunto[i]` para obtener "el i-ésimo" elemento (no hay concepto de orden índice).

**Contar elementos:** Para todas estas estructuras, Python proporciona la función incorporada `len()`, que devuelve el número de elementos contenidos. Por ejemplo, `len(mi_lista)` retorna la longitud de `mi_lista` [23], y de forma similar `len(mi_tupla)`, `len(mi_conjunto)` o `len(mi_diccionario)` dan el tamaño de cada uno (en el diccionario, cuenta cuántas parejas clave-valor hay).

## Iterar sobre los elementos de una estructura

Es muy común **recorrer** los elementos de una estructura de datos para procesarlos. En Python, todas las estructuras mencionadas son *iterables*, lo que significa que podemos usar un bucle `for` para iterar automáticamente por sus elementos:

- **Listas/tuplas:** la iteración recorrerá los elementos en el orden en que están en la secuencia. Ejemplo: `for x in mi_lista: print(x)` imprimirá cada elemento en orden. También se puede iterar por índice usando `for i in range(len(mi_lista)):` si es necesario el índice, aunque generalmente se prefiere iterar directamente por los valores.

- **Conjuntos:** al ser no ordenados, iterar sobre un set recorrerá sus elementos en un orden *arbitrario* (internamente dependiente del hash de los elementos). Por ejemplo, `for elem in mi_conjunto:` procesará cada elemento en algún orden indeterminado. Esto es adecuado cuando el orden no importa. Si necesitas un orden específico, tendrías que convertir el set a una lista ordenada primero, por ejemplo.

- **Diccionarios:** al iterar directamente un diccionario con `for clave in mi_diccionario:`, obtienes cada clave en el orden de inserción. Luego puedes usar esa clave para obtener el valor. Python ofrece métodos especiales para diccionarios que facilitan diferentes recorridos: `mi_diccionario.keys()` produce un iterable de todas las claves, `mi_diccionario.values()` de todos los valores, y `mi_diccionario.items()` de pares (clave, valor). Por ejemplo, para recorrer pares clave-valor:

  ```
  for clave, valor in mi_diccionario.items():
      print(clave, "->", valor)
  ```

  De esta manera, en cada iteración se obtienen simultáneamente la clave y su valor asociado [24].

La iteración es fundamental para aplicar operaciones a cada elemento, buscar elementos que cumplan cierta condición, sumar valores, etc. Gracias a que estas estructuras son iterables, se integran fácilmente con las construcciones de control de flujo de Python.

## Agregación, modificación y eliminación de elementos

Finalmente, es esencial saber cómo **agregar, cambiar o quitar** elementos de cada estructura de datos (en aquellas que lo permiten):

- **En listas:** al ser mutables, las listas soportan múltiples operaciones de modificación. Podemos **modificar** el valor de un elemento existente asignando en su índice: por ejemplo `mi_lista[0] = 42` cambia el primer elemento al valor 42 [25]. Para **agregar** elementos, el método más usado es `mi_lista.append(x)`, que añade `x` al final de la lista. Si se quiere insertar en una posición específica, se utiliza `mi_lista.insert(i, x)` donde `i` es el índice en el que insertar el nuevo elemento [26]. También es posible concatenar listas con el operador `+` o con `extend()` para añadir múltiples elementos a la vez. Para **eliminar** elementos, existen varias opciones: `mi_lista.remove(valor)` elimina la primera aparición de *valor* en la lista (si no existe ese valor, lanza un `ValueError`) [27]; el método `pop(i)` elimina y devuelve el elemento en el índice `i` (por defecto, el último si no se especifica `i`) [28]; y la instrucción

`del mi_lista[i]` elimina el elemento en la posición dada sin retornarlo. Además, `mi_lista.clear()` vacía la lista por completo. Estas operaciones permiten gestionar dinámicamente el contenido de la lista.

- **En tuplas:** dado que las tuplas son inmutables, **no es posible** agregar, modificar ni remover elementos una vez creada la tupla. No existen métodos como append o remove para tuplas. La única manera de cambiar una tupla es creando una tupla nueva. Por ejemplo, para "agregar" un elemento podríamos construir una tupla concatenada: `tupla_nueva = tupla_original + (nuevo_elem,)`. Si intentamos asignar a un índice de tupla, obtendremos un error de tipo (`'tuple' object does not support item assignment`) [29]. En resumen, las operaciones de modificación no aplican a tuplas; su contenido permanece fijo.

- **En conjuntos:** los sets son mutables, por lo que podemos **agregar** o **eliminar** elementos. Para añadir un elemento al conjunto se utiliza `mi_conjunto.add(x)`, que inserta el elemento *x* (si ya estaba en el set, no ocurre nada nuevo porque no puede haber duplicados) [30]. Para añadir múltiples elementos, se puede usar `mi_conjunto.update(otro_iterable)` que incorpora todos los elementos de otro conjunto o lista. Para **eliminar**, el método principal es `mi_conjunto.remove(x)`, que elimina el elemento *x*; si *x* no está en el set, lanza un `KeyError` [31]. Existe también `mi_conjunto.discard(x)` que elimina *x* si está presente, y si no, no hace nada (no lanza error). Otra operación útil es `mi_conjunto.pop()`, que remueve y devuelve *un elemento arbitrario* del conjunto (típicamente el "último" en la hash interna, pero esencialmente no sabemos cuál a priori) [32]. Finalmente, `mi_conjunto.clear()` vacía todo el conjunto dejándolo sin elementos. Con estas operaciones podemos gestionar el contenido del set, sabiendo que no tenemos control sobre el orden pero sí garantizamos la unicidad de elementos.

- **En diccionarios:** los diccionarios también son mutables y permiten agregar, cambiar y quitar pares. Para **agregar** una nueva entrada o **modificar** una existente, se usa la asignación por clave: `mi_dict[nueva_clave] = valor`. Si la clave no existía, se crea ese par clave-valor nuevo; si ya existía, esta operación sobreescribe el valor antiguo con el nuevo [22]. Alternativamente, el método `update()` permite incorporar múltiples pares de otro diccionario o iterable de tuplas. Para **eliminar** elementos, se puede usar la instrucción `del mi_dict[clave]` para remover la entrada de esa clave [22]. También el método `pop(clave)` elimina la clave dada y devuelve su valor asociado, y `popitem()` elimina y devuelve el último par insertado (útil para tratar la estructura como una pila de pares). Al igual que en listas y sets, `mi_dict.clear()` elimina todos los ítems del diccionario dejándolo vacío. Por ejemplo, dado `tel = {"Ana": 1111, "Bob": 2222}`, ejecutar `del tel["Bob"]` removerá a Bob del diccionario. Si intentamos acceder o eliminar una clave que no existe, Python lanzará un `KeyError` (a menos que usemos métodos seguros como `get()` o `pop()` con valor por defecto). Gracias a estas operaciones, podemos mantener actualizado el contenido de un diccionario – añadiendo nuevas asociaciones o eliminando las que ya no se necesitan – durante la ejecución de un programa.

En conclusión, Python ofrece un rico conjunto de estructuras de datos básicas, cada una con operaciones eficientes para distintos propósitos. Comprender las **características** de listas, tuplas, conjuntos y diccionarios – su mutabilidad, ordenamiento, restricciones y métodos disponibles – nos permite modelar soluciones más limpias y eficaces a los problemas, eligiendo la estructura adecuada para cada situación. Con práctica, sabremos identificar rápidamente si un problema se beneficia de una lista (secuencia ordenada modificable), una tupla (datos fijos agrupados), un set (colección de únicos sin

orden) o un diccionario (mapeo rápido de claves a valores), y aplicaremos las operaciones pertinentes para manipular esos datos en Python de forma óptima. [33] [34]

---


[1] [33] GitHub - LuisAlejandroSalcedo/Estructuras-De-Datos: Implementaciones de Estructuras de Datos en Python.
https://github.com/LuisAlejandroSalcedo/Estructuras-De-Datos

[2] [5] [6] [7] [23] [25] Estructuras de datos: listas, tuplas, conjuntos y diccionarios — Repositorio de PAD
https://copa-uniandes.github.io/PAD-web-tutorials/Laboratorio-Computacional-de-Analytics/S1%20-%20Bienvenida%2C%20estructuras%20de%20datos%20y%20de%20control/S1.TU2/TUTORIAL%20-%20Estructuras%20de%20datos.html

[3] [4] [8] [9] [10] [11] [12] [13] [14] [15] [16] [21] [22] [26] [27] [28] [29] 5. Estructuras de datos — documentación de Python - 3.15.0a2
https://docs.python.org/es/dev/tutorial/datastructures.html

[17] [18] [19] [20] [34] ¿Cuándo debería usar una lista, diccionario, tupla o conjunto en Python? : r/learnpython
https://www.reddit.com/r/learnpython/comments/1j4ia9n/when_should_i_use_a_list_dictionary_tuple_or_set/?tl=es-419

[24] Cómo usar listas, diccionarios, tuplas y sets en Python
https://cosasdedevs.com/posts/como-usar-listas-diccionarios-tuplas-y-sets-en-python/

[30] [31] [32] Set Python | El Libro De Python
https://ellibrodepython.com/sets-python