

Listas Uso Avanzado



Definición de Tipo de Dato Abstracto

Un **Tipo de Dato Abstracto** (TDA) es un modelo teórico que define un conjunto de datos y las operaciones que se pueden realizar sobre ellos, sin especificar cómo se implementan internamente.



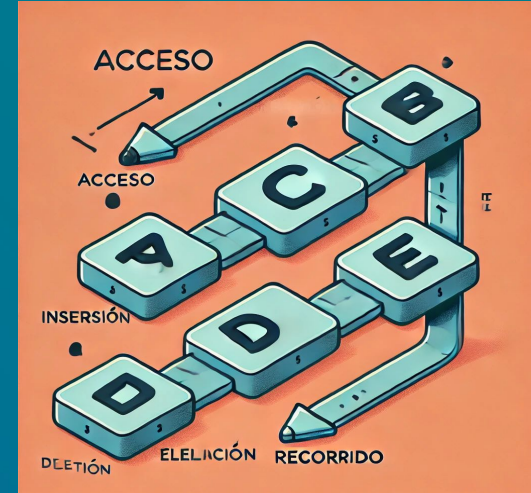
Lista: entre los TDA más comunes

Definición de Lista como TDA:

- Una lista es una colección **ordenada** de elementos, donde cada uno tiene una posición **definida** por un índice.
- **Mutabilidad:** los elementos pueden ser modificados después de su creación.
- **Acceso secuencial:** Se puede acceder a los elementos mediante un **índice**.
- **Tamaño dinámico:** Se pueden agregar o eliminar elementos durante la ejecución.
- Permite almacenar datos **heterogéneos** (de diferentes tipos).

Operaciones Comunes

- **Acceso:** Obtener el valor en una posición específica.
- **Inserción:** Añadir un nuevo elemento en una posición dada.
- **Eliminación:** Remover elementos de la lista.
- **Recorrido:** Iterar sobre todos los elementos de manera secuencial.

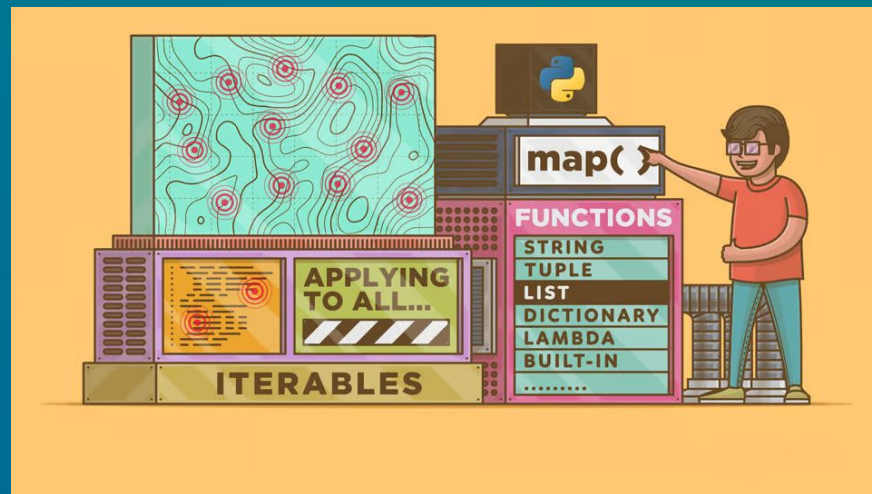


¿Qué es un método?

Un **método** es una función que pertenece a un objeto o clase y que opera sobre los datos que ese objeto contiene. Los métodos definen el comportamiento que un tipo de dato o estructura, permitiendo manipular sus elementos o interactuar con ellos de manera específica.

¿Cómo funciona un Método en listas?

Cuando llamamos a un método sobre una lista, se está invocando una función que pertenece a esa lista y que puede modificarla o devolver información sobre **su contenido**.



Métodos de Inserción

append

El método **append** agrega un elemento al final de la lista:

```
frutas = ["manzana", "plátano"]  
frutas.append("naranja")  
print(frutas) # ['manzana', 'plátano', 'naranja']
```

Métodos de Inserción

insert

El método **insert** inserta un elemento en la posición especificada:

```
frutas = ["manzana", "plátano"]  
frutas.insert(1, "kiwi")  
print(frutas) # ['manzana', 'kiwi', 'plátano']
```


Métodos de Inserción

extend

El método **extend** permite añadir una lista a la lista inicial.

```
frutas = ["manzana", "plátano"]
otras_frutas = ("pera", "uva")

# Extender la lista con una tupla
frutas.extend(otras_frutas)
print(frutas) # ['manzana', 'plátano', 'pera', 'uva']
```

Métodos de Eliminación

remove

El método **remove** elimina la primer ocurrencia del elemento especificado:

```
frutas = ["manzana", "plátano", "manzana"]  
frutas.remove("manzana")  
print(frutas) # ['plátano', 'manzana']
```

Métodos de Eliminación

pop

El método **pop** elimina y devuelve el elemento en la posición dada. Si no se especifica un índice, se elimina y devuelve el último elemento.

```
frutas = ["manzana", "plátano", "naranja"]  
fruta = frutas.pop(1)  
print(fruta) # 'plátano'  
print(frutas) # ['manzana', 'naranja']
```

Métodos de Eliminación

clear

El método **clear** elimina **todos los elementos de la lista**, dejándola vacía.

```
frutas = ["manzana", "plátano", "naranja"]  
frutas.clear()  
print(frutas) # []
```

Métodos de Acceso

index

El método **index** devuelve el índice de la primera ocurrencia del elemento especificado.

```
frutas = ["manzana", "plátano", "naranja", "kiwi", "plátano"]  
posicion = frutas.index("plátano")  
print(f"El plátano está en la posición: {posicion}")
```

Métodos de Acceso

index

El método **index** tiene parámetros opcionales. **lista.index(elemento, inicio, fin)**.
Inicio: Desde que posición comenzar a buscar. **fin:** Hasta que posición buscar

```
frutas = ["manzana", "plátano", "naranja", "kiwi", "plátano"]  
posicion = frutas.index("plátano", 2) # Empieza a buscar desde el índice 2  
print(f"El segundo plátano está en la posición: {posicion}")
```

Métodos de Ordenamiento

sort

El método **sort** ordena la lista en orden ascendente.

```
numeros = [3, 1, 4, 2]  
numeros.sort()  
print(numeros) # [1, 2, 3, 4]
```

Métodos de Ordenamiento

reverse

El método **reverse** invierte el orden de los elementos de la lista

```
frutas = ["manzana", "plátano", "naranja"]  
frutas.reverse()  
print(frutas) # ['naranja', 'plátano', 'manzana']
```


Otra forma de Eliminación del lista[índice]

La keyword **del lista[índice]** elimina **el elemento de la posición indicada** usando la palabra clave del. (No es un método)

```
frutas = ["manzana", "plátano", "naranja"]  
del frutas[0]  
print(frutas) # ['plátano', 'naranja']
```

Métodos de Eliminación del lista[inicio: fin]

También se puede utilizar para eliminar **varios elementos** con **slicing**:

```
numeros = [0, 1, 2, 3, 4, 5]
del numeros[1:4]
print(numeros) # [0, 4, 5]
```

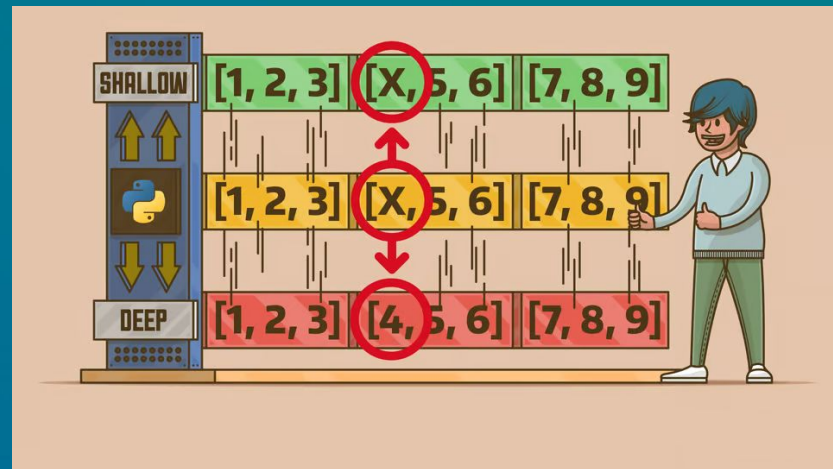
Copiar listas

Recordemos que no es posible copiar una lista simplemente escribiendo

lista_2 = lista_1, que en ese caso **lista_2** solo será **una referencia** a **lista_1**.

Existen dos tipos de copias posibles:

- **Superficial** (Shallow copy)
- **Profunda** (deep copy)



Superficial (shallow copy)

Una **shallow copy** (copia superficial) crea **una nueva lista** que contiene referencias a los mismos elementos que la lista original. Esto significa que, si modificamos los **objetos internos** de la nueva lista, los cambios se verán también en la lista original, porque ambos comparten los mismos objetos.

Superficial (shallow copy): Ejemplo

Resultado:

Ambas listas (la original y la copia) reflejan los mismos cambios porque se realizó una copia superficial: los elementos anidados no fueron duplicados, sino que ambas listas comparten las mismas referencias.

```
import copy

# Lista original con objetos anidados
lista_original = [[1, 2], [3, 4], [5, 6]]

# Copia superficial de la lista
lista_copia = copy.copy(lista_original)

# Modificamos un elemento dentro de un objeto anidado en la copia
lista_copia[0][0] = 99

# Imprimimos ambas listas
print("Lista Original:", lista_original) # [[99, 2], [3, 4], [5, 6]]
print("Lista Copia:", lista_copia)      # [[99, 2], [3, 4], [5, 6]]
```

Deep Copy (Copia Profunda)

Una **deep copy** (copia profunda) crea una nueva lista **independiente**, duplicando todos los objetos, incluso los anidados. A diferencia de la **shallow copy**, en una deep copy los cambios en los elementos internos de la lista copiada **no** afectan a la lista original.

Deep Copy (Copia Profunda)

Resultado

- deepcopy() crea una copia **completamente independiente de la lista**, incluyendo los objetos anidados.
- En este caso, al modificar un elemento en la lista copiada, **la lista original no cambia**, ya que ambos objetos son totalmente separados.

```
import copy

# Lista original con objetos anidados
lista_original = [[1, 2], [3, 4], [5, 6]]

# Crear una copia profunda de la lista
lista_copia_profunda = copy.deepcopy(lista_original)

# Modificamos un elemento dentro de la copia profunda
lista_copia_profunda[0][0] = 99

# Imprimimos ambas listas para ver las diferencias
print("Lista Original:", lista_original) # [[1, 2], [3, 4], [5, 6]]
print("Lista Copia Profunda:", lista_copia_profunda) # [[99, 2], [3, 4], [5, 6]]
```

Comparación entre Shallow Copy y Deep Copy

Tipo de Copia	Descripción	Efecto de Objetos Internos
Shallow Copy	Copia solo las referencias a los objetos internos	Comparte los mismos objetos
Deep Copy	Crea una copia completa, duplica todos los objetos	Independencia completa

Función enumerate

La función `enumerate()` agrega un contador (índice) a un iterable, como una lista, devolviendo un objeto enumerado. Esto es especialmente útil cuando necesitas acceder a los índices de los elementos mientras iteras sobre ellos.

```
frutas = ["manzana", "plátano", "naranja"]  
  
# Usar enumerate para mostrar el índice y el valor  
for indice, fruta in enumerate(frutas):  
    print(f"Índice: {indice}, Fruta: {fruta}")
```

```
Índice: 0, Fruta: manzana  
Índice: 1, Fruta: plátano  
Índice: 2, Fruta: naranja
```

Función zip

Permite iterar múltiples listas a la vez

```
nombres = ["Ana", "Luis", "Pedro"]  
edades = [25, 30, 35]  
  
for nombre, edad in zip(nombres, edades):  
    print(f"{nombre} tiene {edad} años.")
```

```
Ana tiene 25 años.  
Luis tiene 30 años.  
Pedro tiene 35 años.
```

Conclusión

- Las listas son una **estructura fundamental** por su flexibilidad y facilidad de uso. Los métodos asociados permiten **manipular y gestionar** los elementos de forma eficiente, facilitando el trabajo con conjuntos de datos dinámicos.
- El uso adecuado de los métodos de listas en Python permite resolver **problemas complejos** de forma eficiente y clara. Estos métodos simplifican el desarrollo de aplicaciones que manejan datos dinámicos, como inventarios, registros, listas de tareas o procesamiento de información.
- Aprender y dominar estos métodos es fundamental para **mejorar la calidad del código** y asegurar que las soluciones sean **escalables y mantenibles** en el tiempo.