«Talento Tech»

Iniciación a la

Programación con Python

Clase 07

Buenos Aires Ciudad

Clase N° 7 | Listas y Tuplas

Temario:

- Listas y tuplas: diferencias y similitudes.
- Uso de índices y slices.
- Métodos comunes de listas (append(), remove(), etc.) y tuplas.
- Iteración y manipulación de listas y tuplas.

Objetivos de la Clase.

En esta clase vas a profundizar tus conocimientos sobre **listas**, aprendiendo cómo usarlas para almacenar, modificar y organizar datos de manera eficiente. También vas a conocer las **tuplas**, una estructura similar a las listas pero con una característica clave: su contenido no se puede modificar después de ser creado.

Aprenderás a identificar las diferencias y similitudes entre listas y tuplas, entendiendo cuándo conviene utilizar una u otra según las necesidades de tu programa. Además, veremos cómo trabajar con **índices** y **slices** para acceder y manipular partes específicas de estas estructuras.

Por último, vas a explorar los métodos más comunes para trabajar con listas, como **append()**, **remove()** y otros que te permitirán realizar operaciones de manera práctica y rápida. También practicaremos cómo recorrer y procesar listas y tuplas utilizando bucles, integrando lo aprendido en las clases anteriores.

Otra mañana en TechLab 🚀

Al revisar tu correo de **TalentoLab** recibís un mensaje de Mariana, la Gerente de Proyectos, asignándote una nueva misión para mejorar tus habilidades en programación:

“Necesitamos implementar un registro ordenado de los nombres de las empresas que trabajan con **TalentoLab** . Tu tarea hoy consistirá en crear una lista, ordenada alfabéticamente, de todas ellas. Más tarde te daré todos los detalles necesarios.“

Luis, el desarrollador senior, te guía y explica la manera de mantener un registro ordenado de los y las clientes y cómo una buena gestión de datos puede mejorar la eficiencia del equipo. ¡Manos a la obra!

Listas

Las listas en Python son muy flexibles. No solo podés acceder a sus elementos, sino también modificarlos, agregar nuevos valores o eliminar los que ya no necesitás. Esta capacidad de cambiar el contenido de una lista es lo que llamamos **mutabilidad**, y es una de las características que hacen a las listas tan útiles para resolver problemas en tus programas.

Método append()

El método **append()** te permite agregar un nuevo elemento al final de una lista. Es muy útil cuando no conocés de antemano cuántos datos vas a manejar, ya que podés ir añadiéndolos uno a uno según sea necesario.

Este método no modifica los elementos existentes, simplemente extiende la lista con el nuevo valor.

A continuación, vamos a ver un ejemplo práctico. Vamos a pedir al usuario que ingrese los montos de ventas realizadas, y vamos a almacenarlos en una lista utilizando append(). Cuando el usuario ingrese un "0", el programa terminará de recopilar datos y mostrará la lista completa.

```
# Creamos una lista vacía para almacenar los montos de las ventas
ventas = []

print("Ingresá los montos de ventas. Escribí '0' para finalizar.")

# Usamos un bucle while para recopilar los datos
while True:
    monto = int(input("Ingresá el monto de la venta: "))
    if monto == 0:
        break  # Finalizamos el bucle si el monto es 0
    ventas.append(monto)  # Agregamos el monto a la lista

# Mostramos la lista completa de ventas
print("\nLista de ventas ingresadas:")
for venta in ventas:
    print(f"- ${venta}")
```

Primero, creamos una lista vacía llamada **ventas**, que se usará para almacenar los datos ingresados. Luego, un bucle **while** solicita al usuario que ingrese montos de ventas. Cada valor ingresado se agrega a la lista utilizando el método append(). Si el usuario ingresa un "0", el programa rompe el bucle con la instrucción **break**.

Después de recopilar todos los datos, un bucle for recorre la lista para mostrar cada monto de manera ordenada y clara. Esto muestra cómo **append()** puede ser combinado con bucles para manejar datos de forma dinámica.

```
Ingresá los montos de ventas. Escribí '0' para finalizar.
Ingresá el monto de la venta: 1000
Ingresá el monto de la venta: 1500
Ingresá el monto de la venta: 800
Ingresá el monto de la venta: 1200
Ingresá el monto de la venta: 0

Lista de ventas ingresadas:
- $1000
- $1500
- $800
- $1200
```

Método remove()

El método **remove()** se utiliza para eliminar el primer elemento de una lista que coincida con un valor específico. Es importante destacar que este método solo elimina la primera aparición del valor, por lo que, si el elemento se repite en la lista, las demás ocurrencias permanecerán intactas. Si el valor no se encuentra en la lista, Python generará un error.

Para entender cómo funciona remove(), vamos a analizar con un ejemplo. En este caso, crearemos una lista de productos vendidos y permitiremos al usuario eliminar productos específicos de la lista, uno por uno, hasta que decida terminar.

Este ejemplo demuestra cómo **remove()** puede ser útil para gestionar dinámicamente los elementos de una lista, permitiendo modificar contenido según sea necesario.

```
# Creamos una lista inicial con productos vendidos
productos = ["pan", "leche", "queso", "pan", "yogur"]

print("Lista de productos actuales:")
print(productos)

print("\nPodés eliminar un producto de la lista. Escribí 'fin' para
salir.")

# Usamos un bucle while para permitir al usuario eliminar productos
while True:
   producto_a_eliminar = input("Producto a eliminar: ").lower()
   if producto_a_eliminar == "fin":
       break  # Finalizamos el bucle si el usuario escribe "fin"
   if producto_a_eliminar in productos:
       # Eliminamos el producto de la lista
       productos.remove(producto_a_eliminar)
       print(f"'{producto_a_eliminar}' fue eliminado. Lista
actualizada:")
       print(productos)
   else:
       print(f"El producto '{producto_a_eliminar}' no está en la
lista.")

# Mostramos la lista final
print("\nLista final de productos:")
print(productos)
```

Primero, definimos una lista inicial llamada productos que contiene algunos artículos repetidos. Luego, el programa muestra esta lista al usuario y le da la opción de eliminar elementos uno por uno.

Usamos un bucle while para solicitar el nombre del producto a eliminar. Si la usuaria escribe "fin", el programa termina. Si el producto está en la lista, utilizamos remove() para eliminar la primera aparición de ese producto. Finalmente, mostramos la lista actualizada después de cada eliminación. Si el producto no está en la lista, se muestra un mensaje de advertencia.

```
Lista de productos actuales:
['pan', 'leche', 'queso', 'pan', 'yogur']

Podés eliminar un producto de la lista. Escribí 'fin' para salir.
Producto a eliminar: pan
'pan' fue eliminado. Lista actualizada:
['leche', 'queso', 'pan', 'yogur']
Producto a eliminar: queso
'queso' fue eliminado. Lista actualizada:
['leche', 'pan', 'yogur']
Producto a eliminar: fin

Lista final de productos:
['leche', 'pan', 'yogur']
```

Métodos comunes para listas en Python

Estos métodos son útiles en diversas situaciones y pueden ayudarte a gestionar datos de manera eficiente. Algunos, como **append()**, **remove()**, **count()** y **sort()**, son especialmente relevantes para abordar los desafíos del TFI, donde tendrás que manipular y analizar listas de datos.

Método	Ejemplo de Uso	Explicación

append()	`ventas = []`
	`ventas.append(100)`
	`print(ventas)`
	`# Salida: [100]`
		Agrega un elemento al final de la lista.

remove()	`productos = ["pan", "leche"]`
	`productos.remove("pan")`
	`print(productos)`
	`# Salida: ["leche"]`
		Elimina la primera aparición de un valor en la lista. Si el valor no existe, genera un error.

pop()	`valores = [10, 20, 30]`
	`x = valores.pop()`
	`print(x)  # Salida: 30`
	`print(valores)  # Salida: [10, 20]`
		Elimina y devuelve el último elemento de la lista (o el índice especificado).

insert()	`nombres = ["Ana", "Luis"]`
	`nombres.insert(1, "Carlos")`
	`print(nombres)`
	`# Salida:`
	`# ["Ana", "Carlos", "Luis"]`
		Inserta un elemento en una posición específica de la lista.

extend()	`lista1 = [1, 2]`
	`lista2 = [3, 4]`
	`lista1.extend(lista2)`
	`print(lista1)`
	`# Salida: [1, 2, 3, 4]`
		Extiende la lista añadiendo los elementos de otra lista o iterable al final.

index()	`colores = ["rojo", "azul", "verde"]`
	`pos = colores.index("azul")`
	`print(pos)`
	`# Salida: 1`
		Devuelve la posición de la primera aparición de un valor en la lista.

count()	`numeros = [1, 2, 2, 3]`
	`ocurrencias = numeros.count(2)`
	`print(ocurrencias)`
	`# Salida: 2`
		Cuenta cuántas veces aparece un valor en la lista.

reverse()	`letras = ["a", "b", "c"]`
	`letras.reverse()`
	`print(letras)`
	`# Salida: ["c", "b", "a"]`
		Invierte el orden de los elementos en la lista.

sort()	`edades = [25, 18, 30]`
	`edades.sort()`
	`print(edades)`
	`# Salida: [18, 25, 30]`
		Ordena los elementos de la lista en orden ascendente (o descendente si se usa reverse=True).

clear()	`datos = [1, 2, 3]`
	`datos.clear()`
	`print(datos)`
	`# Salida: []`
		Elimina todos los elementos de la lista, dejándola vacía.

El siguiente ejemplo muestra cómo un método como **sort()** puede integrarse fácilmente con estructuras de control como **for**, permitiendo gestionar y presentar información de manera eficiente:

```
# Lista inicial con nombres de productos
productos = ["yogur", "pan", "queso", "leche", "manteca"]

# Ordenamos los productos alfabéticamente
productos.sort()

# Mostramos los productos ordenados
print("Lista de productos ordenados alfabéticamente:")
for producto in productos:
   print(f"- {producto}")
```

Primero definimos una lista llamada **productos** que contiene nombres desordenados. Luego, utilizamos el método **sort()** para ordenar los elementos alfabéticamente. Una vez ordenada, recorremos la lista con un bucle **for** para mostrar cada producto en la terminal, presentándolo de forma organizada.

Las tuplas en Python

Hasta ahora, trabajamos con listas y exploramos sus características, como su mutabilidad y los métodos útiles para manipularlas. Sin embargo, Python también ofrece otra estructura similar pero con diferencias importantes: las **tuplas**.

Ambas estructuras permiten almacenar múltiples valores, acceder a ellos mediante índices y ser recorridas con bucles, pero la elección entre listas y tuplas depende del contexto. Si necesitás manipular los datos, una lista es la mejor opción. Si buscás garantizar que los datos permanezcan inalterados, una tupla será más adecuada.

Las tuplas son estructuras de datos similares a las listas, pero con una característica principal que las diferencia: **son inmutables**. Esto significa que, una vez creada, una tupla no se puede modificar. No podés agregar, eliminar ni cambiar los elementos que contiene.

Esta inmutabilidad las hace ideales para representar datos constantes o que no deberían alterarse durante la ejecución de un programa, como coordenadas, configuraciones o valores que necesitan mantenerse seguros.

Creación de tuplas
Las tuplas se definen utilizando paréntesis ( ), aunque también se pueden crear sin paréntesis explícitos, separando los elementos por comas. Por ejemplo:

```
# Creación de una tupla
mi_tupla = (10, 20, 30)

# Creación sin paréntesis
otra_tupla = 40, 50, 60

# Tupla con un solo elemento (requiere una coma final)
tupla_unica = (100,)
```

Acceso a elementos
Al igual que las listas, los elementos de una tupla se acceden mediante índices, comenzando en 0. También podés utilizar índices negativos para acceder desde el final:

```
colores = ("rojo", "verde", "azul")
print(colores[0])  # Salida: rojo
print(colores[-1])  # Salida: azul
```

Tuplas y métodos

Aunque las tuplas son inmutables, algunas operaciones y métodos aún se pueden aplicar. Por ejemplo:

Conteo de elementos con count():

```
numeros = (1, 2, 2, 3)
print(numeros.count(2))  # Salida: 2
```

Índice de un elemento con index():

```
frutas = ("manzana", "pera", "manzana")
print(frutas.index("pera"))  # Salida: 1
```

Relación con las listas

Aunque listas y tuplas comparten características, como el acceso por índices y la capacidad de ser recorridas con bucles, la decisión entre usar una u otra depende de la necesidad de mutabilidad. Por ejemplo, si estás trabajando con datos que no necesitan cambiar, como una tabla de constantes, una tupla es la mejor opción:

```
# Ejemplo de tupla para datos inmutables
coordenadas = (40.7128, -74.0060)
print(f"Latitud: {coordenadas[0]}, Longitud: {coordenadas[1]}")
```

Por otro lado, si necesitás agregar, eliminar o actualizar elementos, una lista será más adecuada.

Los métodos de las tuplas están limitados comparados con las listas, lo que refleja su naturaleza estática. Como siempre, Luis está atento a tus avances, y te proporciona una tabla que contiene los métodos más comunes y si aplica o no a estos tipos de datos.

Método	Listas	Tuplas
append()	✔️ Se aplica. Agrega un elemento al final de la lista.	❌ No se aplica. Las tuplas son inmutables, por lo que no se pueden agregar elementos.
remove()	✔️ Se aplica. Elimina la primera aparición de un elemento específico.	❌ No se aplica. No se pueden eliminar elementos de una tupla debido a su inmutabilidad.
pop()	✔️ Se aplica. Elimina y devuelve un elemento de una posición específica.	❌ No se aplica. Las tuplas no permiten eliminar elementos.
insert()	✔️ Se aplica. Inserta un elemento en una posición específica.	❌ No se aplica. No se pueden modificar las posiciones de los elementos en una tupla.
extend()	✔️ Se aplica. Agrega múltiples elementos al final de la lista.	❌ No se aplica. Las tuplas no pueden extenderse con nuevos elementos.
index()	✔️ Se aplica. Devuelve el índice de la primera aparición de un elemento.	✔️ Se aplica. Las tuplas permiten buscar el índice de un elemento.

count()	✔️ Se aplica. Cuenta cuántas veces aparece un elemento.	✔️ Se aplica. Las tuplas permiten contar la cantidad de veces que aparece un elemento.
reverse()	✔️ Se aplica. Invierte el orden de los elementos de la lista.	❌ No se aplica. No se puede modificar el orden de los elementos en una tupla.
sort()	✔️ Se aplica. Ordena los elementos de la lista en orden ascendente/descendente.	❌ No se aplica. Las tuplas no pueden modificarse ni ordenarse.
clear()	✔️ Se aplica. Elimina todos los elementos de la lista.	❌ No se aplica. Las tuplas no permiten eliminar elementos ni vaciar su contenido.

Convertir entre listas y tuplas

A veces, podés necesitar convertir una lista en tupla o viceversa. Esto es posible utilizando las funciones tuple() y list():

```
# Conversión de lista a tupla
mi_lista = [1, 2, 3]
mi_tupla = tuple(mi_lista)
print(mi_tupla)  # Salida: (1, 2, 3)

# Conversión de tupla a lista
otra_tupla = (4, 5, 6)
otra_lista = list(otra_tupla)
print(otra_lista)  # Salida: [4, 5, 6]
```

Las tuplas son una herramienta útil para cuando se requiere proteger la integridad de los datos. Su inmutabilidad aporta seguridad y claridad en muchos escenarios, como definir configuraciones o datos que no deben cambiar. A medida que avancemos, veremos cómo combinarlas con listas y otros conceptos para manejar información de manera más eficiente.

Slices en listas y yuplas

El concepto de **slices** (ya utilizado con las cadenas de caracteres) permite acceder a porciones específicas de **listas** o **tuplas**, seleccionando un rango de elementos. Esto es útil cuando necesitás trabajar con subconjuntos de datos, como extraer valores específicos para un análisis o realizar operaciones sobre una parte de la estructura.

Sintaxis de slicing
La sintaxis básica para crear un slice es:

```
estructura[inicio:fin:paso]
```

- **inicio**: Es el índice donde comienza el slice (incluido).
- **fin**: Es el índice donde termina el slice (excluido).
- **paso**: Indica cuántos elementos avanzar en cada iteración (opcional).

Si no especificás alguno de estos valores, Python usará los predeterminados:

- **inicio**: El primer elemento de la estructura.
- **fin**: El último elemento.
- **paso**: 1 (recorre de a uno).

Ejemplo con Listas
Vamos a trabajar con una lista para extraer partes específicas de la misma:

```
numeros = [10, 20, 30, 40, 50, 60]

# Obtener los primeros tres elementos
print(numeros[:3])  # Salida: [10, 20, 30]

# Obtener los últimos dos elementos
print(numeros[-2:])  # Salida: [50, 60]

# Obtener todos los elementos con un paso de 2
print(numeros[::2])  # Salida: [10, 30, 50]
```

En estos ejemplos, podés ver cómo los slices permiten acceder fácilmente a diferentes partes de la lista sin necesidad de recorrerla manualmente con bucles.

Ejemplo con tuplas
El slicing funciona de la misma manera en las tuplas, ya que comparten la misma lógica de indexación:

```
colores = ("rojo", "verde", "azul", "amarillo")

# Obtener los dos primeros colores
print(colores[:2])  # Salida: ("rojo", "verde")

# Obtener los colores desde el tercero hasta el final
print(colores[2:])  # Salida: ("azul", "amarillo")

# Obtener los colores en orden inverso
print(colores[::-1])  # Salida: ("amarillo", "azul", "verde", "rojo")
```

Diferencias entre slicing y acceso directo

El slicing permite trabajar con múltiples elementos a la vez, mientras que el **acceso directo** con índices individuales (estructura[indice]) se limita a un solo elemento por operación. Esto hace que los slices sean más eficientes para procesar rangos de datos. En general, vas a usar slices para:

- **Dividir datos:** Podés dividir listas grandes en partes más pequeñas, como separar los datos de ventas en trimestres.
- **Invertir datos:** Usar slices con pasos negativos ([::-1]) es una forma eficiente de invertir una lista o tupla.
- **Extraer valores específicos:** Seleccionar subconjuntos de datos con una lógica clara, cómo extraer los valores impares de una lista.

Los slices son una herramienta fundamental para manejar listas y tuplas, evitando bucles innecesarios y haciendo el código más limpio y legible, que amplían significativamente las posibilidades de manipulación de datos en Python.

Ejemplo práctico: Análisis de datos con slices

Luis te propone crear un programa que permita analizar los datos de ventas semanales almacenados en una lista. El programa extraerá las ventas de los primeros tres días de la semana, las ventas de los últimos dos días, y calculará el total de ventas de los días

intermedios utilizando slices. Esto te mostrará cómo trabajar con diferentes partes de una lista.

```
# Lista que representa las ventas diarias de una semana
ventas_semanales = [150, 200, 250, 300, 100, 180, 220]

# Extraer las ventas de los primeros tres días
primeros_dias = ventas_semanales[:3]
print(f"Ventas de los primeros tres días: {primeros_dias}")

# Extraer las ventas de los últimos dos días
ultimos_dias = ventas_semanales[-2:]
print(f"Ventas de los últimos dos días: {ultimos_dias}")

# Calcular el total de ventas de los días intermedios (excluyendo los
primeros tres y últimos dos días)
dias_intermedios = ventas_semanales[3:-2]
total_intermedios = sum(dias_intermedios)
print(f"Ventas de los días intermedios: {dias_intermedios}")
print(f"Total de ventas de los días intermedios: ${total_intermedios}")
```

Esta es la salida del programa:

```
Ventas de los primeros tres días: [150, 200, 250]
Ventas de los últimos dos días: [180, 220]
Ventas de los días intermedios: [300, 100]
Total de ventas de los días intermedios: $400
```

Veamos que ocurre en este código. Primero, creamos una lista llamada **ventas_semanales** que contiene los montos de ventas diarias de una semana. Utilizamos slices para dividir esta lista en partes específicas según las necesidades:

1. Para obtener las ventas de los primeros tres días, usamos [:3], que selecciona desde el inicio hasta el índice 2 (excluyendo el índice 3). Este resultado se almacena en primeros_dias y se muestra en la pantalla.
2. Las ventas de los últimos dos días se extraen utilizando [-2:], que selecciona desde el penúltimo hasta el último elemento de la lista. Este resultado se guarda en ultimos_dias y se imprime.

3. Finalmente, para analizar los días intermedios, usamos el slice [3:-2], que selecciona los elementos entre el índice 3 (inclusive) y el índice -2 (excluido). Luego, calculamos la suma de estos valores utilizando la función sum() y mostramos tanto la lista de días intermedios como el total calculado.

Este ejemplo muestra cómo los slices facilitan la manipulación de datos en listas, evitando la necesidad de recorrer la lista manualmente o crear múltiples bucles. También muestra cómo combinarlos con otras funciones como **sum()** para realizar cálculos útiles.

Ejercicio práctico.

Mariana te ha escrito un correo donde te explica claramente la tarea que debe resolver el programa a desarrollar:

¡Hola!
En **TalentoLab** necesitamos llevar un registro ordenado de los nombres de los nuevos clientes y clientas que se van incorporando. Queremos asegurarnos de que los datos ingresados sean válidos y estén bien organizados. Tu tarea es escribir un programa en Python que haga lo siguiente:

1. Solicite al usuario o usuaria los nombres de los clientes y clientas uno por uno y valide que cada nombre no esté vacío. Si se deja el campo vacío, mostrale un mensaje de advertencia y volvé a pedir el nombre.
2. Guarde cada nombre válido en una lista, asegurándote de agregarlo con el método .append().
3. Permití que la persona finalice la carga de nombres escribiendo la palabra "fin".
4. Una vez finalizada la carga, ordená alfabéticamente los nombres en la lista y mostrá la lista ordenada de nombres utilizando un bucle for.

¡Es hora de demostrar tu crecimiento como especialista en desarrollo de software! 💪😊

Materiales y Recursos Adicionales:

Artículos:

Sitio oficial: Todo sobre las listas en Python

Pablo Londoño: Listas en Python: qué son, cómo crearlas y ordenarlas

El libro de Python: Listas en Python

Videos:

Byspel: Listas en Python - Parte 1 y Listas en Python - Parte 2
Sergio Castaño Giraldo: Listas en Python

Preguntas para reflexionar:

1. ¿Por qué es importante conocer los métodos más comunes de las listas, como append() o remove()? ¿En qué situaciones prácticas podrían ayudarte estos métodos en el desarrollo del TFI?
2. ¿Cómo pueden los slices simplificar la manipulación de datos en tus programas? ¿Qué beneficios tienen frente a otras formas de acceder a los elementos de una lista o tupla?
3. ¿En qué escenarios el uso de una tupla puede ser preferible al de una lista? Pensá en ejemplos donde la inmutabilidad sea una ventaja.

Próximos pasos:

En la próxima clase, vas a descubrir una estructura de datos fundamental que lleva el manejo de información a otro nivel: los **diccionarios**. Aprenderás cómo crearlos y manipularlos, cómo acceder a sus elementos mediante claves específicas y utilizar métodos esenciales para agregar, modificar y eliminar datos.

Los diccionarios serán un aporte poderoso a tu caja de herramientas de programación, especialmente para tareas que requieran manejar información organizada y etiquetada. Este nuevo conocimiento te acercará aún más a completar con éxito el **Trabajo Final Integrador (TFI)**


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad