Conferencia 3: Estructuras de Datos - Listas y Tuplas

Objetivo de hoy: Al final de esta clase, entenderán cómo agrupar y gestionar colecciones de datos en Python. Dominarán las dos estructuras de datos ordenadas más fundamentales: las **listas** y las **tuplas**. Y lo más importante, comprenderán los conceptos de **mutabilidad** y **referencia**, ideas clave en toda su carrera como programadores.

---

1. ¿Por Qué Necesitamos Estructuras de Datos?

Imaginen que queremos guardar las notas de 5 alumnos. Podríamos hacer esto:

```
nota_alumno1 = 8.5
nota_alumno2 = 9.0
nota_alumno3 = 7.2
nota_alumno4 = 10.0
nota_alumno5 = 6.8
```

Funciona, pero es ineficiente. ¿Y si son 1000 alumnos? ¿Cómo calculamos el promedio? ¿Cómo encontramos la nota más alta? Necesitamos una forma de agrupar estos datos en una sola "caja" o "contenedor". Aquí es donde entran las estructuras de datos.

Hoy veremos estructuras **ordenadas**: donde cada elemento tiene una posición específica y podemos contar con que esa posición no cambiará.

---

2. Listas: La Navaja Suiza de Python

Una **lista** es la estructura de datos más común y versátil de Python. Piénsenla como un tren de carga: una colección ordenada de vagones, y en cada vagón podemos meter lo que queramos (números, texto, incluso otras listas) y podemos cambiar la carga de un vagón, añadir vagones nuevos o quitarlos.

Creación de Listas

Se crean usando corchetes `[]` y separando los elementos con comas.

```
# Una lista de números
notas = [8.5, 9.0, 7.2, 10.0, 6.8]

# Una lista de texto (strings)
nombres = ["Ana", "Juan", "Mercedes", "Yosvany"]

# Una lista mixta (totalmente válido)
mixta = ["Hola", 100, True, 3.14]

# Una lista vacía, para llenarla más tarde
tareas_pendientes = []

print(notas)
print(nombres)
```

Accediendo a los Elementos: Indización

La clave de las estructuras ordenadas es el **índice**. En Python, como en muchos lenguajes, la numeración **empieza en 0**. Esto es fundamental.

Importante: Un índice puede ser cualquier expresión que se evalúe como entero. No tiene que ser un número literal.

Imagina la lista `nombres = ["Ana", "Juan", "Mercedes", "Yosvany"]`:

Elemento	"Ana"	"Juan"	"Mercedes"	"Yosvany"
Índice	0	1	2	3

```
# Acceder al primer elemento (índice 0)
primer_nombre = nombres[0]
print(f"El primer nombre es: {primer_nombre}")  # Salida: El primer nombre es: Ana

# Acceder al tercer elemento (índice 2)
tercer_nombre = nombres[2]
print(f"El tercer nombre es: {tercer_nombre}") # Salida: El tercer nombre es: Mercedes

# El índice puede ser una variable o el resultado de una expresión
posicion = 1
segundo_nombre = nombres[posicion]
print(f"El segundo nombre es: {segundo_nombre}") # Salida: El segundo nombre es: Juan

# Para esto necesitamos conocer la función len(). Esta función nos dice cuántos elementos
# tiene una lista (o cualquier colección). Es muy útil para calcular índices dinámicamente.
print(f"La lista tiene {len(nombres)} elementos") # Salida: La lista tiene 4 elementos

# Ahora usamos len() para calcular el índice del último elemento
ultimo_nombre = nombres[len(nombres) - 1]
print(f"El último nombre es: {ultimo_nombre}") # Salida: El último nombre es: Yosvany
```

Nota: Indización Negativa

Python nos permite la indización negativa. Sirve para empezar a contar desde el final. `-1` es el último elemento, `-2` el penúltimo, y así sucesivamente. Es muy útil.

```
# Obtener el último nombre sin saber el tamaño de la lista
ultimo_nombre = nombres[-1]
print(f"El último nombre es: {ultimo_nombre}") # Salida: El último nombre es: Yosvany

# Obtener el penúltimo
penultimo_nombre = nombres[-2]
print(f"El penúltimo es: {penultimo_nombre}") # Salida: El penúltimo es: Mercedes
```

El Operador `in`

El operador `in` nos permite verificar si un elemento específico está presente en la lista. Es como preguntar: "¿Está este elemento en mi lista?"

```python
nombres = ["Ana", "Juan", "Mercedes", "Yosvany"]

# Verificar si "Ana" está en la lista
if "Ana" in nombres:
    print("Ana está en la lista")
else:
    print("Ana no está en la lista")

# También funciona con números
notas = [2, 2, 2, 2, 2, 3, 2, 2, 5, 2, 2]
if 5 in notas:
    print("Hay una nota perfecta")
```

La Palabra Clave `del`: Eliminar por Índice

La palabra clave **`del`** nos permite eliminar un elemento específico de la lista usando su índice. Es como decir: "Elimina el elemento que está en esta posición".

```python
alumnos = ["Ana", "Juan", "Mercedes", "Yosvany", "Yoliesky"]
print(f"Lista original: {alumnos}")

# Eliminar el elemento en la posición 2 (Juan)
del alumnos[2]
print(f"Después de eliminar índice 2: {alumnos}")

# También podemos eliminar el último elemento
del alumnos[-1]
print(f"Después de eliminar el último: {alumnos}")
```

Slicing en Listas

El **slicing** (rebanado) nos permite extraer una porción de una lista usando la sintaxis **`[inicio:fin]`**. Es como cortar un pedazo de pastel: especificas dónde empezar y dónde terminar.

Sintaxis Básica

```python
lista = ["A", "B", "C", "D", "E", "F"]
#        0    1    2    3    4    5

# Extraer elementos del índice 1 al 3 (sin incluir el 4)
porcion = lista[1:4]  # Resultado: ["B", "C", "D"]
print(f"Porción extraída: {porcion}")
```

Reglas del Slicing

- **`inicio`**: Índice donde empezar (incluido)
- **`fin`**: Índice donde terminar (NO incluido)
- Si omites `inicio`, empieza desde el principio: `lista[:3]`
- Si omites `fin`, va hasta el final: `lista[2:]`

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

# Primeros 3 elementos
primeros = numeros[:3]    # [10, 20, 30]
print(f"Primeros tres: {primeros}")

# Últimos 3 elementos
ultimos = numeros[-3:]    # [40, 50, 60]
print(f"Últimos tres: {ultimos}")

# Del medio
medio = numeros[2:4]      # [30, 40]
print(f"Del medio: {medio}")

# Toda la lista
todos = numeros[:]        # [10, 20, 30, 40, 50, 60]
print(f"Toda la lista: {todos}")
```

Slicing con Pasos

Puedes especificar un "paso" para saltar elementos: `[inicio:fin:paso]`

```python
letras = ["A", "B", "C", "D", "E", "F", "G", "H"]

# Cada segundo elemento
pares = letras[::2]       # ["A", "C", "E", "G"]
print(f"Cada segundo elemento: {pares}")

# Invertir la lista
reversa = letras[::-1]    # ["H", "G", "F", "E", "D", "C", "B", "A"]
print(f"Lista invertida: {reversa}")

# Del índice 1 al 6, cada 2 elementos
seleccion = letras[1:7:2] # ["B", "D", "F"]
print(f"Selección con paso: {seleccion}")
```

Slicing con Expresiones Enteras

Los índices pueden ser **cualquier expresión que resulte en un entero**. Esto es muy poderoso:

```python
datos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Usando variables
inicio = 2
fin = 8
paso = 2
resultado = datos[inicio:fin:paso]  # [3, 5, 7]
print(f"Con variables: {resultado}")

# Usando operaciones matemáticas
primera_mitad = datos[:len(datos) // 2]       # [1, 2, 3, 4, 5]
print(f"Primera mitad: {primera_mitad}")
```

---

3. Tuplas

Una **tupla** es casi idéntica a una lista: es una colección ordenada de elementos. La diferencia radica en que una tupla es **inmutable**.

Piénsenla como una caja de cristal sellada. Podemos ver lo que hay dentro y en qué orden está, pero una vez cerrada, no podemos cambiar, añadir o quitar nada.

Creación de Tuplas

Se crean usando paréntesis `()` y separando los elementos con comas.

```
# Una tupla de coordenadas (un caso de uso muy común)
punto_3d = (10, 20, 5)

# Una tupla con los días de la semana (datos que no deberían cambiar)
dias_semana = ("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")

# Ojo con las tuplas de un solo elemento: necesitan una coma
tupla_solitaria = (42,) # Sin la coma, Python pensaría que es solo el número 42 entre paréntesis
no_es_una_tupla = (42)

print(type(tupla_solitaria)) # Salida: <class 'tuple'>
print(type(no_es_una_tupla)) # Salida: <class 'int'>
```

Accediendo a los Elementos: Indización (Igual que en las listas)

La indización positiva y negativa funciona exactamente de la misma manera que en las listas.

```
coordenada_x = punto_3d[0] # 10
primer_dia = dias_semana[0] # "Lunes"
ultimo_dia = dias_semana[-1] # "Domingo"

print(f"La coordenada X es {coordenada_x} y el último día de la semana es {ultimo_dia}")
```

Conceptos de Listas que También Aplican en Tuplas

Muchas de las operaciones y conceptos que vimos con las listas también funcionan con las tuplas:

El operador `in` funciona igual:

```
dias_semana = ("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")

if "Viernes" in dias_semana:
    print("Viernes está en la tupla")
```

La función `len()` también funciona:

```
print(f"La tupla tiene {len(dias_semana)} elementos")  # Salida: La tupla tiene 7 elementos
```

El slicing (rebanado) funciona exactamente igual:

```python
# Primeros 3 días
primeros_dias = dias_semana[:3]  # ("Lunes", "Martes", "Miércoles")

# Últimos 2 días
ultimos_dias = dias_semana[-2:]  # ("Sábado", "Domingo")

# Días de trabajo (lunes a viernes)
dias_trabajo = dias_semana[:5]   # ("Lunes", "Martes", "Miércoles", "Jueves", "Viernes")
```

Descomposición de Tuplas

Una de las características más elegantes y poderosas de las tuplas es la **descomposición** (también llamada "unpacking"). Es la capacidad de extraer todos los elementos de una tupla en variables individuales de una sola vez.

Sintaxis Básica de Descomposición

```python
# Tupla con coordenadas 3D
coordenadas = (10, 20, 5)

# Descomposición: extraer cada elemento en su propia variable
x, y, z = coordenadas

print(f"X: {x}, Y: {y}, Z: {z}")  # Salida: X: 10, Y: 20, Z: 5
```

¿Por Qué es Tan Útil?

La descomposición hace que el código sea más legible y expresivo. Comparemos:

Sin descomposición (más verboso):

```python
punto = (3, 7)
coordenada_x = punto[0]
coordenada_y = punto[1]
print(f"El punto está en ({coordenada_x}, {coordenada_y})")
```

Con descomposición (más elegante):

```python
punto = (3, 7)
x, y = punto
print(f"El punto está en ({x}, {y})")
```

Casos de Uso Comunes

Intercambio de Variables (Swapping)

```python
# Intercambiar dos variables sin variable temporal
a = 10
b = 20
print(f"Antes: a={a}, b={b}")

# La magia de la descomposición
a, b = b, a  # Python crea una tupla temporal (b, a) y la descompone
print(f"Después: a={a}, b={b}")  # Salida: Después: a=20, b=10
```

Trabajar con Datos Estructurados

```
# Información de un estudiante: (nombre, edad, nota)
estudiante = ("Ana García", 20, 8.5)

# Descomposición para trabajar con cada campo
nombre, edad, nota = estudiante
print(f"{nombre} tiene {edad} años y su nota es {nota}")

# También funciona con listas (aunque no es tan común)
datos_lista = ["Juan", 22, 9.0]
nombre, edad, nota = datos_lista
print(f"Lista descompuesta: {nombre}, {edad}, {nota}")
```

Descomposición Parcial con *

A veces no necesitas todos los elementos. Python te permite "recoger" el resto con *:

```
# Tupla con muchos elementos
datos = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Solo queremos el primero y el último, el resto no nos importa
primero, *medio, ultimo = datos
print(f"Primero: {primero}, Último: {ultimo}")
print(f"El medio: {medio}")  # Salida: El medio: [2, 3, 4, 5, 6, 7, 8, 9]

# También funciona al revés
primero, segundo, *resto = datos
print(f"Primeros dos: {primero}, {segundo}")
print(f"Resto: {resto}")  # Salida: Resto: [3, 4, 5, 6, 7, 8, 9, 10]
```

---

4. Mutabilidad vs. Inmutabilidad

Aquí está el concepto más importante de la clase de hoy.

- **Mutable** (del latín *mutabilis*, "que puede cambiar"). Un objeto mutable **puede ser modificado** después de su creación. **Las listas son mutables.**
- **Inmutable** (que no puede cambiar). Un objeto inmutable **NO puede ser modificado** después de su creación. **Las tuplas, los números, los strings y los booleanos son inmutables.**

Demostración con Listas (Mutables)

Con una lista, podemos cambiar sus elementos, añadir nuevos o eliminarlos.

```
alumnos = ["Ana", "Juan", "Mercedes", "Yosvany"]
print(f"Lista original: {alumnos}")

# Cambiar un elemento
alumnos[2] = "Carlos" # Pedro es reemplazado por Carlos
print(f"Después de cambiar: {alumnos}")
```

```
# Eliminar un elemento
del alumnos[1] # Adiós, Juan
print(f"Después de eliminar: {alumnos}")
```

La lista original `alumnos` ha cambiado su contenido. Es el mismo "tren", pero con vagones diferentes.

Demostración con Tuplas (Inmutables)

Intentemos hacer lo mismo con una tupla.

```
rgb_color = (255, 0, 128) # Un color fucsia

# Intentemos cambiar el componente azul
rgb_color[2] = 200 # Esto va a fallar
```

Python nos lanza un `TypeError: 'tuple' object does not support item assignment`. Este error es nuestro amigo, nos está diciendo: "Has creado una tupla, un pacto de que estos datos no cambiarían, y estás intentando romperlo".

¿Por qué usar algo inmutable?

1. **Seguridad:** Garantiza que datos importantes (como unas coordenadas, una configuración, los roles de un usuario) no sean modificados por error en otra parte del programa.
2. **Optimización:** Python puede trabajar más rápido con objetos que sabe que nunca cambiarán.
3. **Uso especial:** Solo los objetos inmutables pueden ser usados como claves en diccionarios (lo veremos en la próxima clase).

Regla general: Empieza usando una lista. Si te das cuenta de que tienes una colección de datos que no debería cambiar nunca, conviértela a una tupla.

---

5. Referencia vs. Copia

Este es un concepto fundamental que diferencia a los programadores novatos de los que entienden cómo funciona la memoria.

Imaginemos esta situación:

```
# Creamos una lista de notas
notas_originales = [10, 9, 8]

# Queremos una "copia" para trabajar con ella, así que hacemos:
notas_trabajo = notas_originales
```

Pregunta para la clase: ¿Qué creen que pasa si ahora modifico `notas_trabajo`?

```
notas_trabajo[0] = 5
print(f"Notas de trabajo: {notas_trabajo}") # Salida esperada: [5, 9, 8]
```

Eso tiene sentido. Pero ahora, la sorpresa…

```
print(f"CUIDADO: Notas originales: {notas_originales}")
# Salida: CUIDADO: Notas originales: [5, 9, 8]
```

¿Qué ha pasado? Hemos modificado la lista original sin querer.

La Analogía de la Dirección de Casa

Cuando haces `notas_trabajo = notas_originales`, **NO estás creando una nueva lista**. Estás creando una nueva "etiqueta" o "apodo" (`notas_trabajo`) que apunta a la **misma dirección de memoria** donde vive la lista original.

Es como si yo tengo una casa en la "Calle Falsa 123" (`notas_originales`). Si te doy un papel que dice "mi casa" (`notas_trabajo`) y en él escribo "Calle Falsa 123", no te he dado una casa nueva. Ambos tenemos la misma dirección. Si tú vas y pintas la puerta de verde, cuando yo vuelva a mi casa, la puerta será verde.

Esto ocurre porque las listas son **mutables**. La asignación (`=`) para tipos mutables crea una **referencia** (una flecha a la misma caja), no una copia.

¿Cómo se hace una copia real?

Si de verdad queremos un duplicado, una fotocopia, una casa nueva e independiente, debemos pedirlo explícitamente. Hay dos formas comunes:

Método 1: Usando `.copy()`

```
notas_originales = [10, 9, 8]
notas_copiadas = notas_originales.copy() # Esta es la clave

# Ahora modificamos la copia
notas_copiadas[0] = 5

print(f"Notas copiadas: {notas_copiadas}") # Salida: [5, 9, 8]
print(f"Notas originales: {notas_originales}") # Salida: [10, 9, 8] -> La original está a salvo
```

Método 2: Usando Slicing `[:]`

Una técnica muy pythonica es usar un "slice" o rebanada que cubra la lista entera.

```
notas_originales = [10, 9, 8]
notas_copiadas_slice = notas_originales[:] # Crea una copia de todos los elementos

notas_copiadas_slice[0] = 5

print(f"Notas copiadas (slice): {notas_copiadas_slice}") # Salida: [5, 9, 8]
print(f"Notas originales: {notas_originales}") # Salida: [10, 9, 8] -> Intacta
```

Importante: Con los tipos **inmutables** (como tuplas, números, strings), no tenemos este problema. Si haces `b = a` donde `a` es 5, y luego haces `b = 6`, no estás cambiando el 5 original. Estás haciendo que la etiqueta `b` apunte a un nuevo valor, el 6.

Resumen de la Clase

1. **Listas `[]`**:
   - Contenedores **ordenados** y **mutables** (modificables).
   - Son el caballo de batalla para colecciones de datos que necesitan cambiar.
2. **Tuplas `()`**:
   - Contenedores **ordenados** e **inmutables** (no modificables).
   - Úsalas para datos que deben permanecer constantes, garantizando la integridad.
3. **Indización**:
   - Se accede a los elementos con corchetes `[ ]`.
   - Empieza en `0`.
   - La indización negativa (`-1`) empieza desde el final.
4. **Mutabilidad**:
   - Es la capacidad de un objeto de ser cambiado después de su creación.
   - Las listas son el ejemplo clave de objeto mutable.
5. **Referencia vs. Copia**:
   - La asignación (`=`) de objetos mutables crea una **referencia**, no una copia. Ambas variables apuntan al mismo objeto.
   - Para crear una copia independiente de una lista, usa `.copy()` o `[:]`.