📘 Clase 2: Apuntes sobre Colecciones y Procesamiento de Datos en Python

Listas y manejo de memoria en Python

En Python, cuando trabajamos con listas, es importante entender cómo se manejan en la memoria. Esto afecta cómo se copian y modifican las listas, lo que puede llevar a resultados inesperados si no se tiene en cuenta la referencia a la memoria. Vamos a desglosar cómo funciona este proceso:

📋 Copiar una lista en Python

Cuando haces una copia de una lista, en realidad estás copiando **una referencia** al mismo lugar de memoria. Esto significa que cualquier modificación en la lista original también se reflejará en la copia.

Ejemplo:

```
a = [1, 2, 3, 4, 5]
b = a  # Copiando la lista 'a' en 'b'
```

En este caso, las variables `a` y `b` **apuntan al mismo espacio en memoria**. Si modificamos `a`, los cambios también se verán reflejados en `b`.

🔍 Verificar la referencia de memoria:

Para verificar si dos listas apuntan al mismo espacio de memoria, podemos utilizar la función `id()`, que nos devuelve el identificador único de un objeto en memoria.

```
print(id(a))  # Muestra el ID de 'a'
print(id(b))  # Muestra el ID de 'b', que será el mismo
```

✂️ Uso de `slice` para copiar una lista sin compartir memoria

Si no quieres que las dos listas apunten al mismo espacio en memoria, puedes usar **slicing** para crear una copia independiente de la lista.

```
a = [1, 2, 3, 4, 5]
c = a[:]  # Copia independiente usando slice
```

Aquí, `c` tendrá una copia de los elementos de `a`, pero estará **almacenada en un espacio de memoria diferente**.

Verificación del espacio de memoria:

```
print(id(a))  # ID de 'a'
print(id(c))  # ID de 'c', que será diferente al de 'a'
```

🔄 Modificar listas y efectos en la memoria

Al modificar una lista que comparte la referencia de memoria, los cambios se reflejarán en ambas variables. Pero si usas *slicing* para copiar la lista, las modificaciones solo afectarán a la lista original.

Ejemplo:

```
a = [1, 2, 3, 4, 5]
b = a  # Copia referenciada
```

```
b.append(6)  # Añadir un elemento a 'b'
print(a)  # [1, 2, 3, 4, 5, 6]
print(b)  # [1, 2, 3, 4, 5, 6]
```

Ambas listas han sido modificadas porque apuntan al mismo espacio de memoria.

En cambio, si usamos *slicing*:

```
a = [1, 2, 3, 4, 5]
c = a[:]  # Copia independiente usando slice
c.append(6)  # Añadir un elemento a 'c'
print(a)  # [1, 2, 3, 4, 5]
print(c)  # [1, 2, 3, 4, 5, 6]
```

Aquí, solo `c` se ha modificado, mientras que `a` permanece intacta, porque las dos listas **no comparten el mismo espacio de memoria**.

💡 Importancia en el mundo laboral

En muchos lenguajes de programación, las colecciones como las listas están limitadas a un solo tipo de dato, pero en Python, las listas pueden contener **diferentes tipos de datos** en la misma colección. Esta flexibilidad es sumamente útil en el ámbito laboral, ya que permite manejar datos heterogéneos con facilidad. Por ejemplo, podrías tener una lista que contenga números enteros, cadenas de texto y objetos personalizados.

📚 Ejercicios prácticos

Ejercicio 1: Copia de listas con y sin *slice*

Problema:
Imagina que estás desarrollando un sistema para una biblioteca donde los libros están organizados en listas. Necesitas crear una copia de la lista de libros actuales, pero deseas asegurarte de que cualquier cambio en la lista original no afecte la copia.

1. Crea una lista con los siguientes libros: `["Don Quijote", "Cien años de soledad", "La Odisea"]`.
2. Haz una copia referenciada de la lista.
3. Haz una copia independiente usando slicing.
4. Agrega un nuevo libro solo a la lista original y verifica que no afecte la copia independiente.

Solución paso a paso:

```
# Paso 1: Crear la lista original
libros_originales = ["Don Quijote", "Cien años de soledad",
"La Odisea"]

# Paso 2: Hacer una copia referenciada
libros_referencia = libros_originales

# Paso 3: Hacer una copia independiente con slicing
libros_copia = libros_originales[:]

# Paso 4: Agregar un nuevo libro a la lista original
libros_originales.append("El Principito")

# Verificación
print("Libros originales:", libros_originales)
print("Libros referenciados:", libros_referencia)
print("Libros copiados:", libros_copia)
```

Resultado esperado:

```
Libros originales: ['Don Quijote', 'Cien años de soledad', 'L
a Odisea', 'El Principito']
Libros referenciados: ['Don Quijote', 'Cien años de soledad',
'La Odisea', 'El Principito']
```

```
Libros copiados: ['Don Quijote', 'Cien años de soledad', 'La
Odisea']
```

Ejercicio 2: Modificar una lista sin afectar la copia

Problema:
Estás desarrollando una aplicación de recetas. Cada receta tiene una lista de ingredientes. Quieres asegurarte de que si haces cambios en la lista de ingredientes de una receta, las demás recetas no se vean afectadas.

1. Crea una lista de ingredientes para una receta de "Pizza Margarita".
2. Haz una copia de la lista utilizando slicing.
3. Añade un nuevo ingrediente solo a la lista original y verifica que la copia no se vea afectada.

Solución paso a paso:

```
# Paso 1: Crear la lista de ingredientes para Pizza Margarita
ingredientes_pizza = ["harina", "tomate", "queso", "albahac
a"]

# Paso 2: Hacer una copia con slicing
ingredientes_pizza_copia = ingredientes_pizza[:]

# Paso 3: Agregar un nuevo ingrediente a la lista original
ingredientes_pizza.append("aceite de oliva")

# Verificación
print("Ingredientes Pizza Original:", ingredientes_pizza)
print("Ingredientes Pizza Copia:", ingredientes_pizza_copia)
```

Resultado esperado:

```
Ingredientes Pizza Original: ['harina', 'tomate', 'queso', 'a
lbahaca', 'aceite de oliva']
```

```
Ingredientes Pizza Copia: ['harina', 'tomate', 'queso', 'alba
haca']
```

🤓 Conclusión

Python nos ofrece mucha flexibilidad al trabajar con listas, especialmente en cómo se manejan en la memoria. Saber cuándo usar una copia referenciada y cuándo hacer una copia independiente mediante *slicing* es crucial para evitar errores inesperados en tu código. Los conceptos de referencia de memoria y *slicing* son fundamentales para trabajar eficientemente con colecciones en Python, y dominar estos temas será de gran ayuda tanto en proyectos personales como en el entorno laboral