Introducción a los Arreglos

En Python, los arreglos (también conocidos como listas o arrays) son estructuras de datos que permiten almacenar colecciones de elementos, como números, cadenas o incluso otras listas. Un **arreglo de una dimensión** es una secuencia lineal de elementos, donde cada uno se accede a través de un índice.

Características de los Arreglos de una Dimensión:

- Son **dinámicos**, es decir, su tamaño puede cambiar durante la ejecución del programa.
- Los índices en un arreglo comienzan en **0**.
- Pueden almacenar elementos de **diferentes tipos**, aunque generalmente se utilizan para almacenar datos del mismo tipo. A diferencia de lenguajes como Java, donde los arrays deben ser de un solo tipo, en Python puedes combinar números, cadenas, booleanos, e incluso otras listas dentro de una misma lista.

```
mi_lista = [1, "hola", 3.14, True]
print(mi_lista)  # Salida: [1, 'hola', 3.14, True]
```

Comprobación de tipos:

Si necesitas verificar el tipo de un elemento antes de realizar una operación, puedes usar la función type() o isinstance().

```
mi_lista = [1, "hola", 3.14, True]

for elemento in mi_lista:
   print(f"{elemento} es de tipo {type(elemento)}")
```

Crear y Acceder a un Arreglo

En Python, los arreglos de una dimensión se implementan utilizando listas (list).

Crear un arreglo vacío:

```
arreglo = []
```

Crear un arreglo con elementos:

```
arreglo = [1, 2, 3, 4, 5]
```

Acceder a un elemento del arreglo:

```
print(arreglo[0])  # Salida: 1

print(arreglo[3])  # Salida: 4
```

Es importante recordar que el índice del primer elemento es 0. Acceder a un índice fuera del rango generará un error.

Modificar Elementos del Arreglo

Se pueden cambiar los valores de los elementos de un arreglo accediendo a ellos por su índice.

```
arreglo[2] = 10  # Cambia el tercer elemento

print(arreglo)   # Salida: [1, 2, 10, 4, 5]
```

Operaciones Comunes con Arreglos

1. **Agregar Elementos:**
   - Se puede agregar un elemento al final del arreglo utilizando el método append()

```
arreglo.append(6)

print(arreglo)  # Salida: [1, 2, 10, 4, 5, 6]
```

2. **Eliminar Elementos:**

- Se puede eliminar un elemento del arreglo por su valor con remove() o por su índice con del.

```
arreglo.remove(10)  # Elimina el valor 10

del arreglo[2]      # Elimina el tercer elemento

print(arreglo)      # Salida: [1, 2, 4, 5, 6]
```

3. **Longitud del Arreglo:**

Para obtener la cantidad de elementos en un arreglo, se usa la función len()

```
print(len(arreglo))  # Salida: 5
```

4. **Iterar sobre un Arreglo:**

- Se puede recorrer un arreglo con un ciclo for para realizar operaciones en cada uno de los elementos.

```
for elemento in arreglo:

  print(elemento)
```

Slicing en Arreglos

Python permite extraer subarreglos mediante **slicing**, una técnica para obtener una porción del arreglo original.

Sintaxis básica:

```
subarreglo = arreglo[inicio:fin]
```

inicio: Índice del primer elemento incluido.

fin: Índice del primer elemento **no incluido**.

Ejemplo:

```
subarreglo = arreglo[1:4]

print(subarreglo)  # Salida: [2, 4, 5]
```

Si inicio no se especifica, se asume 0. Si fin no se especifica, se toma hasta el final del arreglo.

List Comprehension

Es una manera concisa de crear o modificar arreglos en Python.

Ejemplo: Crear un arreglo con los cuadrados de los números del 1 al 5:

```
cuadrados = [x**2 for x in range(1, 6)]

print(cuadrados)  # Salida: [1, 4, 9, 16, 25]
```

Ejemplo: Filtrar los elementos pares de un arreglo:

```
pares = [x for x in arreglo if x % 2 == 0]

print(pares)  # Salida: [2, 4, 6]
```

Funciones y Métodos Útiles para Arreglos

1. **sum()**: Retorna la suma de todos los elementos.

```
suma = sum(arreglo)

print(suma)  # Salida: 18
```

2. **max() y min()**: Encuentran el valor máximo y mínimo del arreglo.

```
print(max(arreglo))  # Salida: 6
```

```
print(min(arreglo))  # Salida: 1
```

3. **sort()**: Ordena el arreglo en orden ascendente.

```
arreglo.sort()
print(arreglo)  # Salida: [1, 2, 4, 5, 6]
```

4. **reverse()**: Invierte el orden de los elementos en el arreglo.

```
arreglo.reverse()
print(arreglo)  # Salida: [6, 5, 4, 2, 1]
```

Conclusión

Los arreglos de una dimensión en Python son fundamentales para manipular y gestionar datos de forma eficiente. A través de listas y arrays de NumPy, se pueden realizar diversas operaciones como agregar, eliminar, ordenar y filtrar elementos, entre otras. Además, el uso de estructuras como list comprehensions y las herramientas avanzadas de bibliotecas como NumPy mejoran la flexibilidad y el rendimiento del código, facilitando el manejo de grandes cantidades de datos en aplicaciones más complejas.

TP Listas de una dimensión

Ejercicios con Listas de una dimensión

Ejercicio 1: Suma de Elementos

Escribe un programa que permita al usuario ingresar una lista de números y calcule la suma de todos los elementos en la lista.

Ejercicio 2: Encontrar el Mayor y el Menor

Escribe un programa que pida al usuario una lista de números y encuentre el mayor y el menor de ellos.

Ejercicio 3: Invertir una Lista

Escribe un programa que permita al usuario ingresar una lista y la invierta.

Ejercicio 4: Contar Elementos Pares e Impares

Escribe un programa que pida al usuario una lista de números y cuente cuántos de ellos son pares y cuántos son impares.

Ejercicio 5: Multiplicar Elementos por un Valor

Escribe un programa que multiplique cada elemento de una lista de números por un valor ingresado por el usuario.

Ejercicio 6: Eliminar Duplicados

Escribe un programa que permita al usuario ingresar una lista de números y elimine los elementos duplicados.

Pista:

- Utiliza la función set().

Ejercicio 7: Promedio de una Lista

Escribe un programa que permita al usuario ingresar una lista de números y calcule el promedio de los elementos.

Ejercicio 8: Encontrar Elementos Repetidos

Escribe un programa que identifique y muestre los elementos que se repiten en una lista.

Pista:

- Utiliza un diccionario o un conjunto (set) para hacer el seguimiento de los elementos.

Ejercicio 9: Lista de Números Primos

Escribe un programa que permita al usuario ingresar una lista de números y filtre los números primos.

Pista:

- Usa una función para verificar si un número es primo.

Ejercicio 10: Eliminar un Elemento por su Índice

Escribe un programa que permita al usuario ingresar una lista de números y eliminar un elemento en un índice especificado.

Ejercicio 11: Contar Ocurrencias de un Elemento

Escribe un programa que permita al usuario ingresar una lista y un número, y cuente cuántas veces aparece ese número en la lista.

Ejercicio 12: Sumar Listas Elemento por Elemento

Escribe un programa que sume dos listas de números elemento por elemento. Las listas deben tener la misma longitud.

Ejercicio 13: Explique y ejemplifique la librería NumPy para trabajar con matrices y arrays

Listas Bidimensionales en Python

Introducción a las Listas Bidimensionales

Una **lista bidimensional** en Python es una lista que contiene otras listas como elementos, formando una estructura de "filas" y "columnas", lo que permite organizar datos en una matriz. Es equivalente a un array de dos dimensiones en otros lenguajes como Java o C.

En Python, las listas bidimensionales se construyen usando listas anidadas, donde cada sublista representa una fila.

Creación de una Lista Bidimensional

Para crear una lista bidimensional en Python, puedes simplemente anidar listas dentro de una lista principal:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

En este ejemplo, matriz es una lista bidimensional con 3 filas y 3 columnas. Cada sublista representa una fila de la matriz.

Acceder a Elementos en una Lista Bidimensional

Para acceder a un elemento en una lista bidimensional, se utilizan dos índices: uno para la fila y otro para la columna. El primer índice selecciona la fila, y el segundo índice selecciona el elemento dentro de esa fila (columna).

```
# Acceder al elemento en la primera fila, segunda columna
elemento = matriz[0][1]
print(elemento)  # Salida: 2
```

Aquí, matriz[0] selecciona la primera fila ([1, 2, 3]), y matriz[0][1] selecciona el segundo elemento en esa fila, que es 2.

Modificar Elementos en una Lista Bidimensional

Para modificar un elemento en una lista bidimensional, accede a él por sus índices y asígnale un nuevo valor.

```
matriz[1][1] = 10

print(matriz)

# Salida: [[1, 2, 3], [4, 10, 6], [7, 8, 9]]
```

En este ejemplo, el valor en la segunda fila, segunda columna se ha cambiado de 5 a 10.

Recorrer una Lista Bidimensional

Para recorrer una lista bidimensional, se suelen utilizar bucles anidados. El primer bucle recorre las filas, y el segundo bucle recorre los elementos de cada fila (las columnas).

```
for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print()
```

Salida:

```
1 2 3

4 10 6

7 8 9
```

En este ejemplo, el programa recorre cada fila de la matriz y luego imprime cada elemento de esa fila.

Creación Dinámica de una Lista Bidimensional

Puedes crear listas bidimensionales de forma dinámica usando bucles. Esto es útil cuando necesitas inicializar una matriz vacía con un tamaño específico.

Ejemplo: Crear una matriz de 3x3 con todos los valores inicializados en 0:

```
filas = 3

columnas = 3
```

```
matriz = [[0 for _ in range(columnas)] for _ in range(filas)]

print(matriz)

# Salida: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

En este ejemplo, usamos una comprensión de listas para generar una matriz de 3x3 llena de ceros.

Lista irregular o matriz irregular.

En Python puedes tener listas bidimensionales en las que las sublistas (filas) pueden tener tamaños diferentes. Esto se conoce como una **lista irregular** o **matriz irregular**. A diferencia de lenguajes más estrictos como Java o C, Python permite que las listas anidadas no tengan el mismo tamaño.

Ejemplo de una Lista Bidimensional Irregular:

```
lista_irregular = [
   [1, 2, 3],
   [4, 5],
   [6, 7, 8, 9]
]

print(lista_irregular)
```

En este caso, la lista lista_irregular tiene:

- Una primera fila con 3 elementos ([1, 2, 3]),
- Una segunda fila con 2 elementos ([4, 5]),
- Y una tercera fila con 4 elementos ([6, 7, 8, 9]).

Python no restringe los tamaños de las sublistas, lo que permite mucha flexibilidad en la estructura de datos. Sin embargo, debes tener cuidado al realizar operaciones que asumen que todas las filas tienen el mismo tamaño, como acceder a un índice específico de cada sublista.

Acceso a Elementos en Listas Irregulares

Cuando accedes a los elementos de una lista bidimensional irregular, debes tener en cuenta que las sublistas pueden tener diferentes longitudes. Intentar acceder a un índice que no existe en una sublista más corta provocará un error IndexError.

```
print(lista_irregular[0][2])  # Accede al elemento 3 de la primera fila, funciona.
```

```
print(lista_irregular[1][2])  # Error: la segunda fila solo tiene 2 elementos.
```

Comprobación del Tamaño de las Sublistas

Para evitar errores, puedes comprobar el tamaño de cada sublista antes de acceder a un elemento específico.

```
for fila in lista_irregular:
    if len(fila) > 2:
        print(fila[2])
    else:
        print("La fila es muy corta para acceder al índice 2")
```

Aplicaciones Prácticas

Este tipo de listas irregulares son útiles cuando trabajas con datos que no están uniformemente estructurados. Por ejemplo, podrías usar una lista irregular para almacenar:

- Notas de estudiantes donde algunos tienen más calificaciones que otros.
- Resultados de encuestas donde no todos los participantes respondieron a todas las preguntas.
- Información jerárquica donde algunas categorías contienen más subcategorías que otras.

En resumen, las listas bidimensionales irregulares son perfectamente válidas en Python y ofrecen flexibilidad adicional para trabajar con datos de estructura variable.

Funciones y Operaciones Comunes

1. Suma de todos los elementos:

Para sumar todos los elementos de una lista bidimensional, puedes usar un bucle anidado o una comprensión de listas.

```
suma_total = sum(sum(fila) for fila in matriz)
print(suma_total)
```

Aquí, sum(fila) suma todos los elementos de cada fila, y luego sum() suma los resultados para obtener la suma total de la matriz.

2. Transponer una Matriz:

Transponer una matriz implica intercambiar las filas por columnas. Puedes lograrlo con una comprensión de listas.

```
transpuesta = [[matriz[j][i] for j in range(len(matriz))] for i in range(len(matriz[0]))]

print(transpuesta)

# Salida: [[1, 4, 7], [2, 10, 8], [3, 6, 9]]
```

Esto toma cada elemento en la posición [i][j] de la matriz original y lo coloca en la posición [j][i] de la matriz transpuesta.

Uso de NumPy para Matrices

Aunque las listas bidimensionales son útiles, **NumPy** ofrece una manera más eficiente de trabajar con matrices y arrays multidimensionales, especialmente en operaciones matemáticas.

Para crear una matriz con NumPy:

```
import numpy as np

matriz_numpy = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

print(matriz_numpy)
```

NumPy proporciona una amplia gama de funciones optimizadas para realizar operaciones matemáticas y manipular matrices, como la transposición, suma de filas o columnas, productos matriciales, etc.

Ejemplos Prácticos

Ejemplo 1: Matriz Identidad

Crea una matriz identidad de tamaño 4x4, donde los elementos de la diagonal principal son 1 y el resto son 0.

```
identidad = [[1 if i == j else 0 for j in range(4)] for i in range(4)]

for fila in identidad:
```

```
    print(fila)
```

Ejemplo 2: Suma de Filas

Escribe un programa que calcule la suma de cada fila en una lista bidimensional.

```
for i, fila in enumerate(matriz):
    print(f"Suma de la fila {i+1}: {sum(fila)}")
```

Conclusión:

Las listas bidimensionales en Python son una herramienta flexible para almacenar y manipular datos tabulares o en forma de matriz. Aunque no son tan eficientes como los arrays nativos de otros lenguajes (como en C o Java), son fáciles de usar y permiten manipular datos de forma intuitiva. Para operaciones más avanzadas y rendimiento optimizado, la biblioteca **NumPy** es la opción recomendada para trabajar con matrices y arrays multidimensionales en Python.

TP Listas Bidimensionales

Ejercicio 1: Crear una Matriz de Números

Crea una función que reciba dos parámetros: el número de filas y columnas. La función debe generar una matriz de ese tamaño, donde los valores son números enteros consecutivos empezando desde 1.

Ejercicio 2: Suma de Todos los Elementos

Escribe un programa que calcule la suma de todos los elementos en una lista bidimensional.

Pista: Aplique la función sum

Ejercicio 3: Suma de Cada Fila

Modifica el programa anterior para que imprima la suma de cada fila de la lista bidimensional.

Ejercicio 4: Matriz Transpuesta

Escribe un programa que calcule la transpuesta de una matriz. La transpuesta de una matriz intercambia sus filas por columnas.

Ejercicio 5: Encontrar el Elemento Mayor

Escribe un programa que encuentre el valor más grande en una lista bidimensional.

Ejercicio 6: Multiplicar una Matriz por un Escalar

Escribe un programa que multiplique cada elemento de una lista bidimensional por un valor escalar dado por el usuario.

Ejercicio 7: Diagonal de una Matriz Cuadrada

Escribe un programa que extraiga los elementos de la diagonal principal de una matriz cuadrada.

Ejercicio 8: Matriz Identidad

Crea un programa que genere una matriz identidad de tamaño n. Una matriz identidad es una matriz cuadrada donde los elementos de la diagonal principal son 1 y el resto son 0.

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Ejercicio 9: Matriz Identidad Inversa

Crea un programa que genere una matriz identidad inversa de tamaño n. Una matriz identidad inversa es una matriz cuadrada donde los elementos de la diagonal inversa principal son 1 y el resto son 0.

0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

Ejercicio 10: Verificar Matriz Simétrica

Una matriz es simétrica si es igual a su transpuesta. Escribe un programa que verifique si una matriz es simétrica.

Ejercicio 11: Rotar una Matriz 90 Grados

Escribe un programa que gire una lista bidimensional (matriz) 90 grados en el sentido de las agujas del reloj.