Programación Básica-Intermedia en Python
Clase 2

Fabián Palacios Pereira, MSc. Ing.[1]

[1] *fpalacios@fiuna.edu.py* Laboratorio de Sistemas Distribuidos,
Dpto. de Ing. Electrónica y Mecatrónica (DIEM),
Centro de Innovación Tecnológica (CITEC),
Facultad de Ingeniería de la UNA (FIUNA), Isla Bogado, Luque, Paraguay.

17 de Mayo del 2025

Resumen

En esta clase se introduce el manejo de estructuras de datos fundamentales en Python como listas, tuplas y diccionarios, detallando sus diferencias, formas de creación, modificación y acceso. Posteriormente, se presenta la biblioteca NumPy, enfocándose en la creación y manipulación de vectores y matrices, así como en operaciones aritméticas básicas entre arreglos. Finalmente, se abordan conceptos esenciales de álgebra lineal aplicados con NumPy, incluyendo el cálculo del producto punto, producto matricial, determinante, inversa de matrices y la resolución de sistemas de ecuaciones lineales, con énfasis en su aplicación en problemas de ingeniería.

1. Concepto de listas, tuplas y diccionarios en Python

1.1. Listas

Las listas son estructuras de datos **mutables** que permiten almacenar una colección ordenada de elementos, que pueden ser de diferentes tipos (núme-

ros, textos, booleanos, incluso otras listas).

```
# Lista vacia
mi_lista = []

# Lista con elementos
mi_lista = [1, 2, 3, "Hola", True, [4, 5]]
```

Características clave de las listas en Python:

- Ordenadas: Los elementos de una lista mantienen el orden en el que se agregaron.
- Mutable: Las listas son dinámicas, lo que significa que se pueden agregar, eliminar o modificar elementos después de su creación.
- Heterogéneas: Una lista puede contener elementos de diferentes tipos (enteros, strings, booleanos, otras listas, etc.).
- Acceso por índice: Los elementos de una lista se pueden acceder a través de su índice, que comienza en 0.
- Permiten duplicados: Una lista puede contener el mismo elemento varias veces.

Operaciones comunes en las listas:

- Agregar elementos:
  - append(): Agrega un elemento al final de la lista.
  - insert(): Agrega un elemento en una posición específica.
  - extend(): Agrega los elementos de otra lista al final de la lista actual.
- Eliminar elementos:
  - remove(): Elimina la primera ocurrencia de un elemento.
  - pop(): Elimina el elemento en una posición específica (o el último si no se especifica).
  - del: Elimina un elemento o una sección de la lista por índice.

- Modificar elementos: Asignar un nuevo valor a un elemento usando su índice.
- Acceder a elementos: Usar el índice para acceder a un elemento específico.
- Segmentar (slicing): Crear una nueva lista con una sección de la lista original.
- Iterar sobre una lista: Usar bucles for para procesar cada elemento de la lista.
- Sumar los elementos con sum().
- Obtener el tamaño de la lista con len().
- Obtener el valor máximo y mínimo con max() y min(), respectivamente.

```
mi_lista = [1, 2, 3]

# Agregar elementos
mi_lista.append(4)  # Resultado: [1, 2, 3, 4]
mi_lista.insert(1, 5)  # Resultado: [1, 5, 2, 3, 4]
mi_lista.extend([6, 7, 8])  # Resultado: [1, 5, 2, 3, 4, 6, 7, 8]

# Eliminar elementos
mi_lista.remove(3)  # Resultado: [1, 5, 2, 4, 6, 7, 8]
elemento_eliminado = mi_lista.pop(2)  # Resultado: [1, 5, 4, 6, 7, 8], elemento_eliminado = 2
del mi_lista[4]  # Resultado: [1, 5, 4, 6, 7]

# Modificar elementos
mi_lista[0] = 9  # Resultado: [9, 5, 4, 6, 7]

# Acceder a elementos
primer_elemento = mi_lista[0]  # Resultado: 9
ultimo_elemento = mi_lista[-1]  # Resultado: 7

```

```
    # Segmentar
    sub_lista = mi_lista[1:3]  # Resultado: [5, 4]

    # Iterar
    for elemento in mi_lista:
        print(elemento)
```

Ejercicios:

- Un sistema de adquisición de datos registra voltajes de una señal analógica a lo largo del tiempo. Se tienen 10 muestras de voltaje almacenadas en una lista:

  ```
      voltajes = [3.2, 3.5, 3.1, 3.6, 3.3, 3.7,
         3.0, 3.4, 3.2, 3.5]
  ```

  Entonces:

  - Calcular el promedio del voltaje.
  - Determinar el valor máximo y el mínimo.
  - Mostrar las posiciones (índices) donde el voltaje es mayor al promedio.

- Crear una lista con los nombres de 3 sensores utilizados en un sistema industrial (por ejemplo: temperatura, presión y caudal). Luego:

  - Mostrar la lista completa.
  - Mostrar el primer y el último sensor.
  - Cambiar el segundo sensor por otro (por ejemplo, reemplazar "presión" por "humedad").
  - Mostrar la lista actualizada.

1.2. Tuplas

En Python, una tupla es una secuencia ordenada e inmutable de elementos. Es similar a una lista, pero una vez creada, no se puede modificar. Las tuplas se definen entre paréntesis y sus elementos están separados por comas.

Características:

- Inmutabilidad: Una vez que se crea una tupla, no se pueden agregar, eliminar o modificar sus elementos.
- Orden: Los elementos de una tupla se mantienen en el orden en que se definen, lo que permite acceder a ellos mediante índices.
- Diversidad de tipos: Los elementos de una tupla pueden ser de diferentes tipos de datos (números, cadenas, booleanos, etc.).

Creación de tuplas:

- Se definen entre paréntesis () y los elementos se separan con comas:

```
mi_tupla = (1, 2, 3, "Hola", True)
```

- Una tupla con un solo elemento debe tener una coma al final:

```
tupla_un_elemento = (1,)  # La coma es
   crucial
```

- También se puede crear una tupla sin paréntesis, mediante un proceso llamado embalado de tupla:

```
a, b, c = 1, 2, 3  # Esto crea una tupla (1,
   2, 3) y la desembala en las variables a, b
   , c
```

Acceso a elementos en las tuplas:

- Se accede a los elementos de una tupla utilizando índices (empezando en 0):

```
print(mi_tupla[0])  # Imprimira 1
print(mi_tupla[3])  # Imprimira "Hola"
```

Operaciones con tuplas:

- Concatenación: Se pueden concatenar tuplas usando el operador +:

```
otra_tupla = (4, 5)
nueva_tupla = mi_tupla + otra_tupla
```

- Repetición: Se puede repetir una tupla usando el operador *:

```
tupla_repetida = mi_tupla * 2  # tupla_repetida sera (1, 2, 3, "Hola", True, 1, 2, 3, "Hola", True)
```

Ejercicios:

- Se desea registrar las coordenadas (x, y) de dos puntos en un plano cartesiano.
  - Crear una tupla con las coordenadas de los puntos, por ejemplo P1(3, 4) y P1(5, 6).
  - Mostrar el valor de x e y por separado de los puntos.
  - Calcular el punto medio entre ambos puntos, utilizando la fórmula:

$$x_p = \frac{x_1 + x_2}{2}; y_p = \frac{y_1 + y_2}{2} \tag{1}$$

  - Guardar las coordenadas del punto medio en una nueva tupla y mostrarla en la salida.

1.3. Diccionarios

Un diccionario (dict) es una estructura de datos que asocia claves con valores. Es parecido a un "diccionario real", donde buscas una palabra (clave) y encontrás su definición (valor).Cada elemento se almacena como un par **clave: valor**.

Sintaxis básica:

```
diccionario = {
    "nombre": "Juan",
    "edad": 30,
    "activo": True
}
```

- Las claves pueden ser cadenas, números u otros tipos inmutables.
- Los valores pueden ser cualquier tipo de dato.

- Las claves deben ser únicas.

Acceso a elementos:

- Para acceder a un valor, se utiliza su clave:

```
print(diccionario["nombre"])  # Imprime: Juan
```

- Si se accede con una clave que no existe, retorna un error:

```
print(diccionario["altura"])  # Error:
    KeyError
```

- Para evitar el error, se puede utilizar get():

```
print(diccionario.get("altura", "No definido
    "))  # Imprime: No definido
```

Operaciones comunes:

- Agregar un nuevo par clave-valor:

```
diccionario["ciudad"] = "Encarnacion"
```

- Modificar un valor existente:

```
diccionario["edad"] = 31
```

- Eliminar un elemento:

```
del diccionario["activo"]
```

- Obtener todas las claves o valores:

```
print(diccionario.keys())
print(diccionario.values())
print(diccionario.items())
```

2. Introducción a paquetes: Numpy

Un paquete o librería es un conjunto de módulos (archivos con código Python) que alguien ya escribió para realizar tareas específicas. Usar paquetes nos permite no "reinventar la rueda" y aprovechar código ya optimizado y probado. Por ejemplo, hay paquetes para matemáticas, manejo de datos, gráficos, redes, etc.

Python tiene un gestor de paquetes llamado pip (Python Package Installer, Pip Install Packages). Con pip podemos instalar paquetes con un comando sencillo, por ejemplo:

```
pip install numpy
```

Luego, en nuestro código, importamos la librería para usar sus funciones:

```
import numpy as np
```

Para saber qué librerías o paquetes están instalados se puede utilizar el comando:

```
pip list
```

Para saber si se tiene una librería instalada:

```
pip show numpy
```

Existen otros gestores de paquetes, como por ejemplo Conda, sin embargo Pip es el más utilizado y viene incluido con las versiones recientes de Python.

2.1. Numpy

- NumPy (Numerical Python) es la librería más popular para computación numérica en Python.
- Permite trabajar con arreglos unidimensionales, bidimensionales y multidimensionales (vectores, matrices, tensores) de forma eficiente.
- Incluye muchas funciones para álgebra lineal, estadística, transformadas, y más.
- Es base fundamental para muchos otros paquetes científicos, como pandas, scipy, matplotlib, y frameworks de machine learning.

2.2. ¿Por qué usar NumPy en ingeniería?

- Los ingenieros suelen trabajar con señales, datos de sensores, simulaciones y modelos matemáticos que involucran vectores y matrices.
- NumPy permite manipular todo tipo de cantidad de datos de forma rápida y con sintaxis clara.
- Por ejemplo, para realizar operaciones entre vectores, cálculos de transformadas, resolver sistemas lineales, etc.

2.3. Creación de vectores y matrices con numpy

2.3.1. Vectores (arrays unidimensionales)

Un vector es una colección ordenada de números, por ejemplo, una lista de mediciones o señales.

```
import numpy as np
# Crear un vector de forma manual
vector = np.array([10, 20, 30, 40])
print("Vector:", vector)
```

2.3.2. Matrices (arrays bidimensionales)

Una matriz es una colección de números organizada en filas y columnas, muy usada en álgebra lineal.

```
# Crear una matriz 2x3 (2 filas, 3 columnas)
matriz = np.array([[1, 2, 3],
                   [4, 5, 6]])
print("Matriz:\n", matriz)
```

2.4. Operaciones con arreglos

NumPy permite trabajar de manera eficiente con vectores y matrices gracias a su estructura optimizada y sintaxis sencilla. Algunas de las operaciones básicas incluyen suma, resta, multiplicación y división entre arreglos del mismo tamaño (shape).

2.4.1. Operaciones básicas

Las operaciones elemento a elemento se realizan directamente con los operadores aritméticos. Estas operaciones funcionan tanto con vectores (1D) como con matrices (2D), siempre que tengan las mismas dimensiones.

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(a + b)  # [5 7 9]
print(a - b)  # [-3 -3 -3]
print(a * b)  # [4 10 18]
print(a / b)  # [0.25 0.4  0.5 ]
```

2.4.2. Producto punto vs multiplicación de matrices

NumPy permite realizar productos entre arreglos usando la función np.dot(). Su comportamiento varía según la dimensión de los arreglos:

- Si se usan vectores 1D, np.dot(a, b) calcula el producto escalar:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(np.dot(a, b))  # 32
```

- Si se usan matrices 2D, entonces np.dot(A, B) realiza la multiplicación de matrices clásica, siguiendo las reglas del álgebra lineal: El número de columnas de A debe coincidir con el número de filas de B.

```
A = np.array([[1, 2, 3]])      # 1x3
B = np.array([[4],
              [5],
              [6]])            # 3x1

print(np.dot(A, B))  # [[32]]
```

2.4.3. Determinante de una matriz

```
    det_A = np.linalg.det(A)
    print("Determinante de A:", det_A)
```

Si det_A == 0, la matriz no tiene inversa (es singular). La razón por la que se habla de la inversa justo después de calcular el determinante es porque el determinante es una condición previa para saber si una matriz es invertible.

2.4.4. Inversa de una matriz

```
    if det_A != 0:
        inv_A = np.linalg.inv(A)
        print("Inversa de A:\n", inv_A)
    else:
        print("La matriz no es invertible.")
```

2.4.5. Traspuesta de una matriz

```
    import numpy as np

    # Matriz A de 2x3
    A = np.array([[1, 2, 3],
                  [4, 5, 6]])

    # Transpuesta de A
    A_transpuesta = A.T
    print("Transpuesta de A:")
    print(A_transpuesta)
```

Ejercicio

Dadas tres matrices:

- A de dimensión 2x3,
- B de dimensión 3x2,
- C de dimensión 2x2,

Entonces:

- Crea las matrices A, B y C con valores enteros a elección.
- Calcula el producto matricial de A por B (AB).
- Suma el resultado de AB con la matriz C. ¿Es esto posible?
- Calcula la transpuesta de la matriz A.
- Crea una nueva matriz D como el producto de B por C.
- Calcula el determinante de la matriz C.
- Si el determinante es distinto de cero, halla la inversa de C.

Tarea: investigar sobre métodos de la librería numpy para resolver sistemas de ecuaciones lineales.