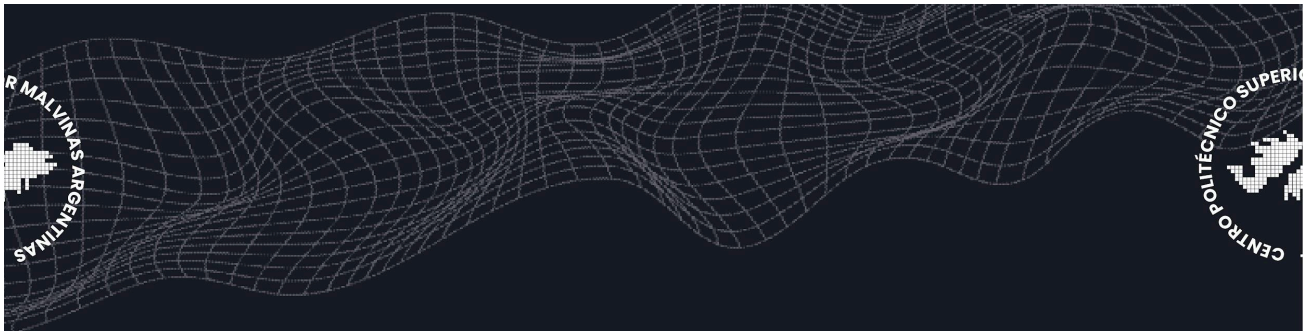


Programación 1

Año de cursada 1º Año

Clase N.º 9: Ordenación en arreglos

Contenido:

En la clase de hoy trabajaremos los siguientes temas:

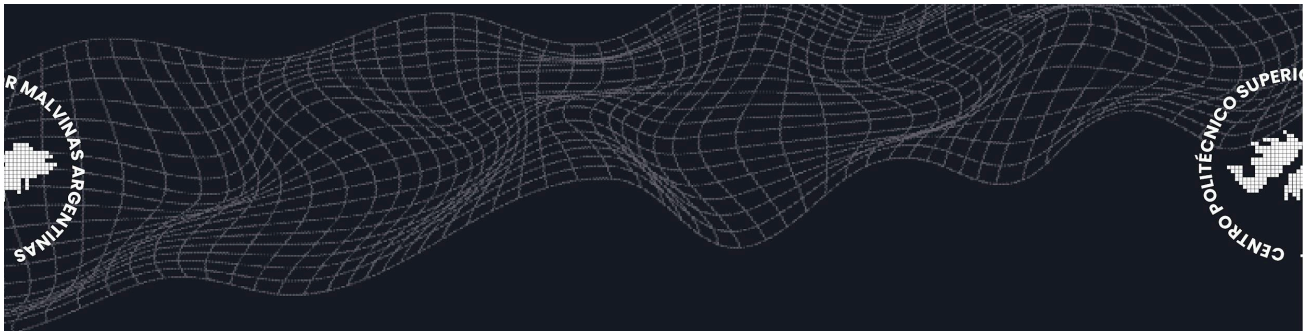
- Ordenación ascendente
- Ordenación descendente
- Ordenación personalizada
- Ordenación basada en atributos de objetos

1. Presentación

¡Hola, bienvenidos/as a la novena clase de Programación. Esta clase ha sido pensada para los/as recién iniciados/as en el mundo de la Programación 1.

Esta presentación trata sobre un concepto fundamental en programación: la ordenación de arreglos con Python!

A lo largo de esta sesión, exploraremos cómo organizar datos en un orden específico para facilitar su manipulación y búsqueda.

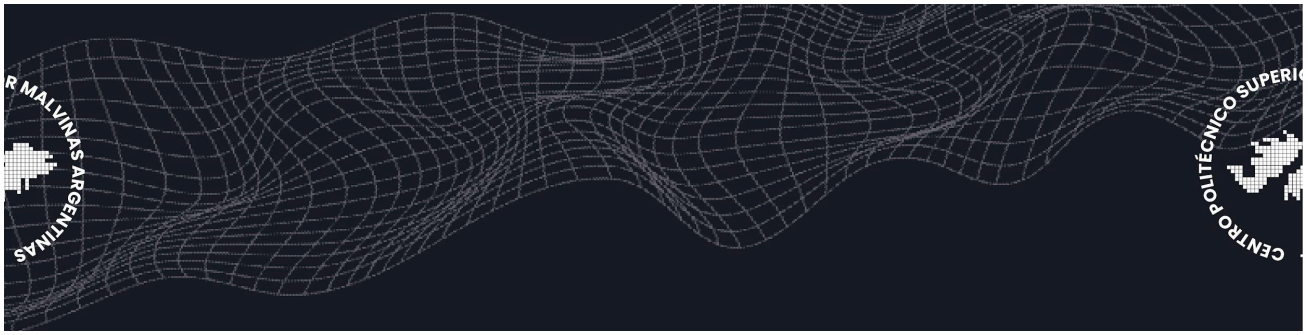


Desarrollo y Actividades

Ordenación ascendente con sorted() y sort()

sorted() es una función incorporada que toma un iterable (como una lista) y devuelve una nueva lista ordenada. sort() es un método que se aplica directamente a una lista para ordenarla en su lugar.

```
# Usando sorted()
original_list = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_list = sorted(original_list)
print(sorted_list)  # Resultado: [1, 1, 2, 3, 4, 5, 6, 9]
print(original_list)  # Resultado: [3, 1, 4, 1, 5, 9, 2, 6] (sin cambios)

# Usando sort()
original_list.sort()
print(original_list)  # Resultado: [1, 1, 2, 3, 4, 5, 6, 9] (modificado)
```



Explicación:

1- Creamos una lista llamada original_list que contiene números desordenados: [3, 1, 4, 1, 5, 9, 2, 6].

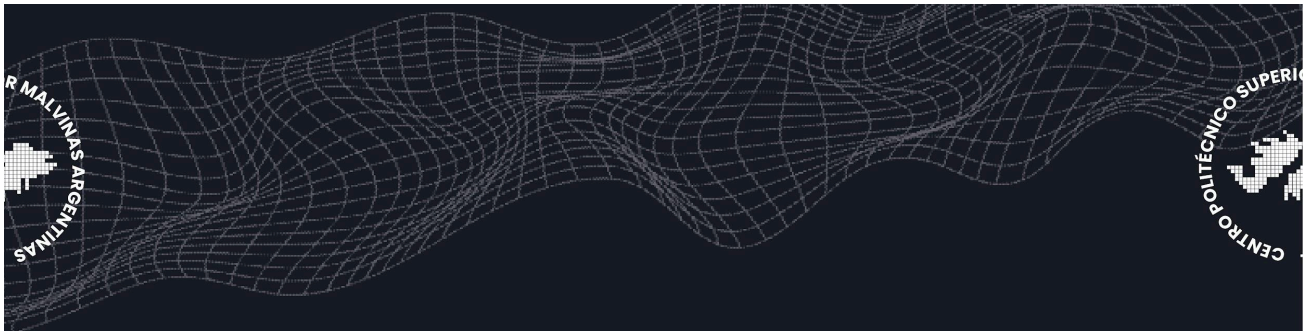
2- Usamos la función sorted(original_list) para ordenar los elementos de original_list y crear una nueva lista ordenada llamada sorted_list. Esta función devuelve una nueva lista sin modificar la lista original.

3- Imprimimos la lista sorted_list, que ahora contiene los números ordenados en forma ascendente: [1, 1, 2, 3, 4, 5, 6, 9]. Luego, imprimimos la lista original_list para demostrar que no ha sido modificada: [3, 1, 4, 1, 5, 9, 2, 6].

4- A continuación, utilizamos el método sort() directamente en la lista original_list. Esto ordenará los elementos de la lista original en su lugar, es decir, modificará la lista original directamente.

5- Finalmente, imprimimos la lista original_list una vez más para mostrar que ahora está ordenada: [1, 1, 2, 3, 4, 5, 6, 9].

En resumen, tanto la función sorted() como el método sort() se utilizan para ordenar listas, pero sorted() crea una nueva lista ordenada sin modificar la original, mientras que sort() modifica la lista original directamente.



Ordenación descendente con el argumento reverse

Tanto sorted() como sort() pueden aceptar el argumento reverse=True para realizar una ordenación descendente.

```
original_list = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_descending = sorted(original_list, reverse=True)
print(sorted_descending)  # Resultado: [9, 6, 5, 4, 3, 2, 1, 1]

original_list.sort(reverse=True)
print(original_list)  # Resultado: [9, 6, 5, 4, 3, 2, 1, 1]
```

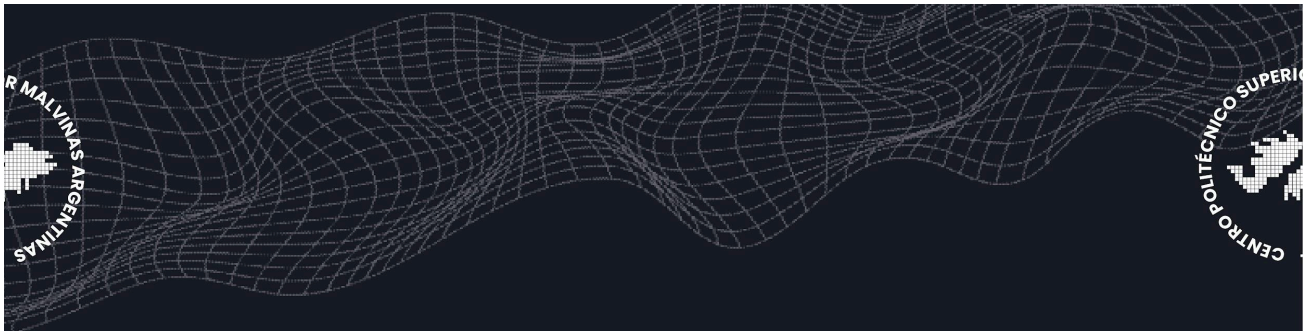
Explicación:

1- Creamos una lista llamada original_list que contiene números desordenados: [3, 1, 4, 1, 5, 9, 2, 6].

2- Usamos la función sorted(original_list, reverse=True) para ordenar los elementos de original_list en orden descendente. El argumento reverse=True indica que queremos ordenar en sentido descendente.

3- Imprimimos la lista sorted_descending, que ahora contiene los números ordenados en orden descendente: [9, 6, 5, 4, 3, 2, 1, 1].

4- A continuación, utilizamos el método sort(reverse=True) directamente en la lista original_list. Esto ordenará los elementos de la lista original en orden descendente.



5- Finalmente, imprimimos la lista original_list una vez más para mostrar que ahora está ordenada en orden descendente: [9, 6, 5, 4, 3, 2, 1, 1].

En resumen, al usar el argumento reverse=True con las funciones sorted() y sort(), podemos ordenar una lista en sentido descendente en lugar del orden ascendente predeterminado.

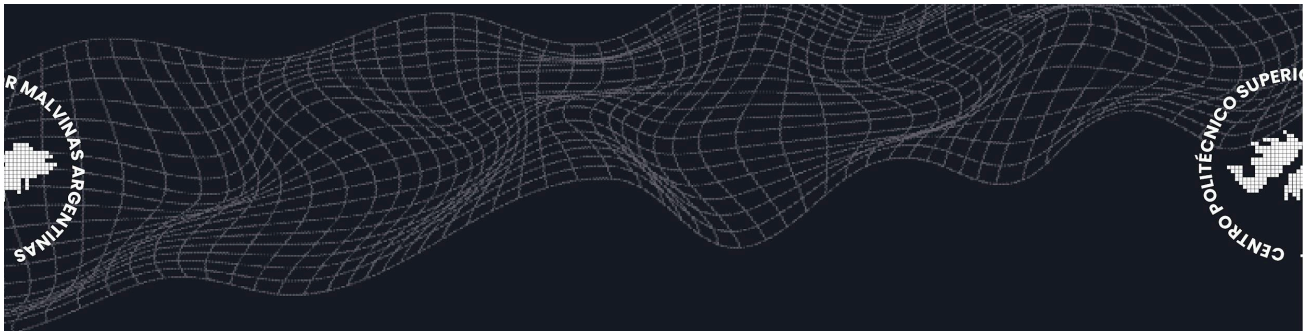
Ordenación personalizada con key

Tanto sorted() como sort() pueden usar el argumento key para definir una función que determine cómo se realizará la comparación entre elementos.

```
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # Resultado: ['date', 'apple', 'banana', 'cherry']
```

Explicación:

1- Creamos una lista llamada words que contiene algunas palabras: ["apple", "banana", "cherry", "date"].



2- Usamos la función sorted() en words, y utilizamos el argumento key para especificar una función que determine cómo se realizará la comparación entre elementos para la ordenación.

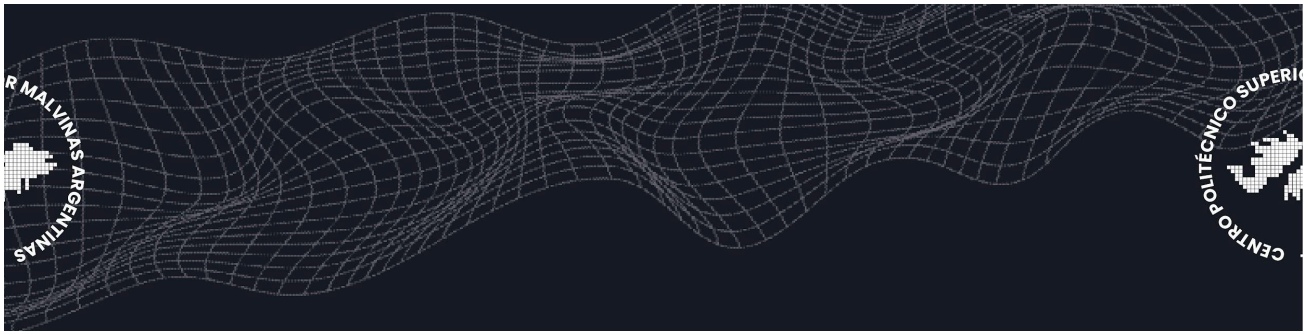
3- En este caso, estamos usando una función lambda: lambda x: len(x). Esta función toma una palabra x como entrada y devuelve su longitud (len(x)). Lo que estamos haciendo aquí es decirle a sorted() que compare las palabras según su longitud en lugar de sus valores.

4- La función sorted() utiliza la función key para ordenar las palabras en función de su longitud. Como resultado, obtendremos la lista sorted_words con las palabras ordenadas por su longitud: ['date', 'apple', 'banana', 'cherry'].

En resumen, el argumento key nos permite definir una función de comparación personalizada para la ordenación, lo que nos brinda flexibilidad para ordenar datos de acuerdo con criterios específicos.

Ordenación basada en atributos de objetos

Si tienes una lista de objetos y quieres ordenarla en función de un atributo específico de esos objetos, puedes usar el argumento key con una función que extraiga ese atributo.



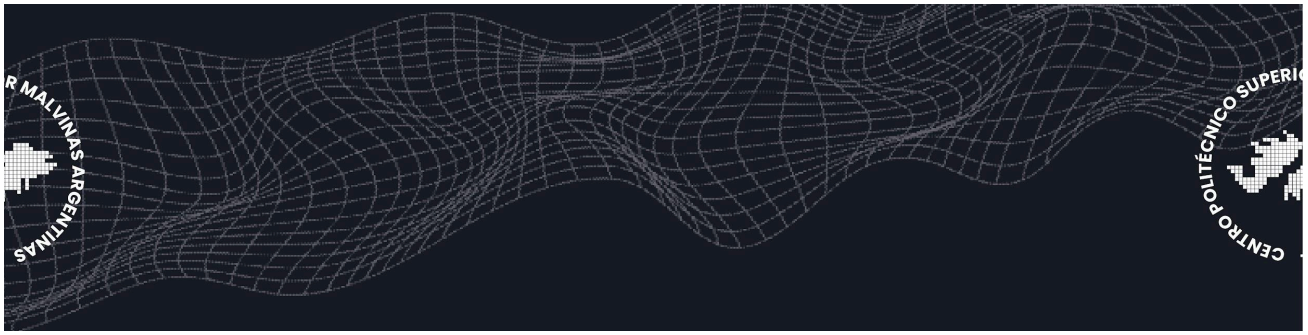

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

people = [Person("Alice", 25), Person("Bob", 30), Person("Eve", 22)]
sorted_people = sorted(people, key=lambda person: person.age)
for person in sorted_people:
    print(person.name, person.age)
# Resultado:
# Eve 22
# Alice 25
# Bob 30
```

Explicación:

1- Definimos una clase llamada Person que tiene un constructor __init__() que toma dos parámetros: name (nombre) y age (edad). Esto nos permite crear objetos Person con atributos name y age.

2- Creamos una lista llamada people que contiene tres objetos de la clase Person, cada uno con un nombre y una edad: Person("Alice", 25), Person("Bob", 30), y Person("Eve", 22).

3- Usamos la función sorted() en la lista people, y utilizamos el argumento key para especificar una función lambda que extraerá el atributo age de cada objeto Person.

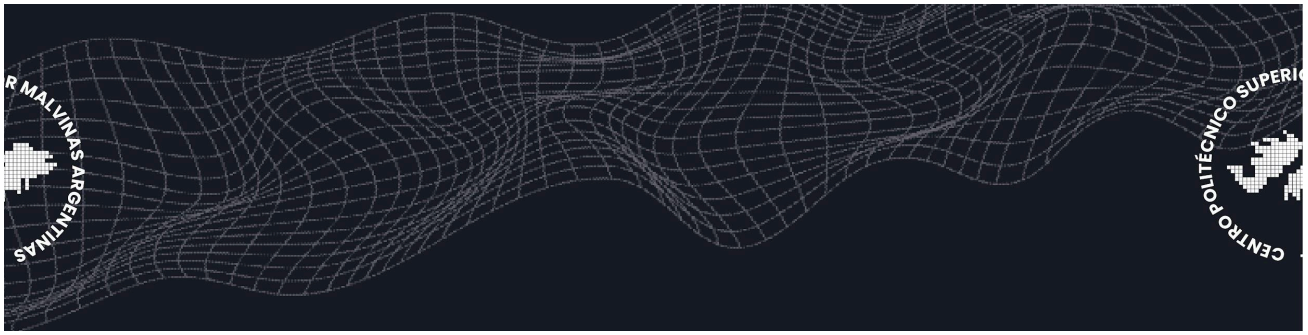


4- La función lambda lambda person: person.age toma un objeto Person como entrada y devuelve su atributo age. Esto le dice a sorted() que compare los objetos Person según sus edades en lugar de sus posiciones en la lista.

5- La función sorted() utiliza la función key para ordenar los objetos Person en función de sus edades. El resultado es la lista sorted_people con los objetos Person ordenados por edad.

6- Finalmente, recorremos la lista sorted_people usando un bucle for y mostramos los nombres y edades de las personas en orden ascendente de edad.

En resumen, usar el argumento key nos permite ordenar una lista de objetos basándonos en un atributo específico de esos objetos, lo cual es especialmente útil cuando trabajamos con conjuntos de datos más complejos.



Actividad 1

Ordenar una lista de números

Dada la siguiente lista de números desordenados, ordénala en orden ascendente utilizando tanto sorted() como el método sort().

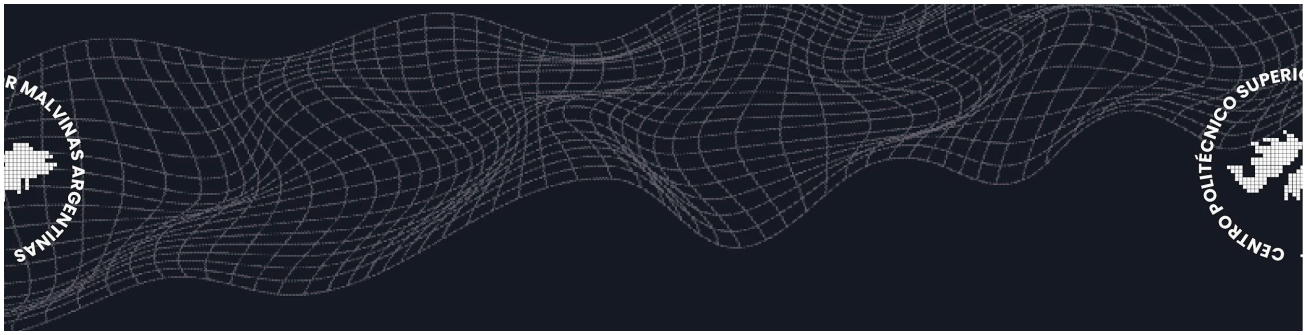
```
numbers = [8, 3, 1, 7, 4, 2, 6, 5]
```

Actividad 2

Ordenar una lista de cadenas por longitud

Dada la siguiente lista de palabras, ordénalas en orden ascendente según la longitud de las cadenas.

```
words = ["apple", "grape", "banana", "pineapple", "kiwi"]
```



Actividad 3

Ordenar una lista de tuplas por segundo elemento

Dada la siguiente lista de tuplas, donde cada tupla contiene un nombre y una edad, ordénala en orden ascendente según las edades.

```
people = [("Alice", 25), ("Bob", 30), ("Eve", 22), ("David", 28)]
```

2. Conclusión

1- Ordenación Ascendente:

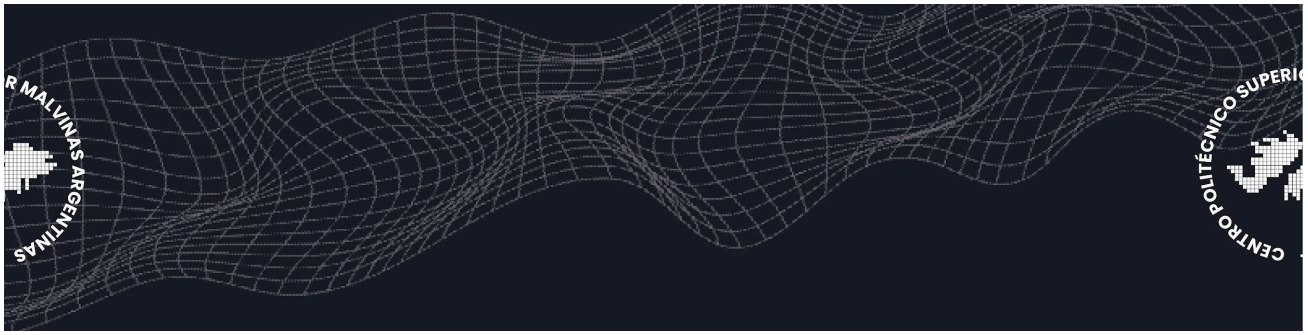
- Identifica Tendencias de Crecimiento: Utiliza la ordenación ascendente para identificar patrones de crecimiento en tus datos, lo que puede ser útil para planificar estrategias a largo plazo.

- Resalta Valores Pequeños: Si estás buscando elementos con valores bajos, la ordenación ascendente te ayudará a encontrarlos rápidamente en la parte superior de la lista.

2- Ordenación Descendente:

- Destaca Valores Altos: Utiliza la ordenación descendente para resaltar los elementos con valores más altos, lo que puede ser crucial para identificar outliers o tendencias a la baja.

- Prioriza Elementos Grandes: Si necesitas seleccionar los elementos más grandes de un conjunto, la ordenación descendente colocará estos elementos al principio de la lista.



3- Ordenación Personalizada:

- Ajusta a tus Necesidades: Define tus propios criterios de orden para adaptar la organización de datos a tus objetivos específicos y requisitos únicos.

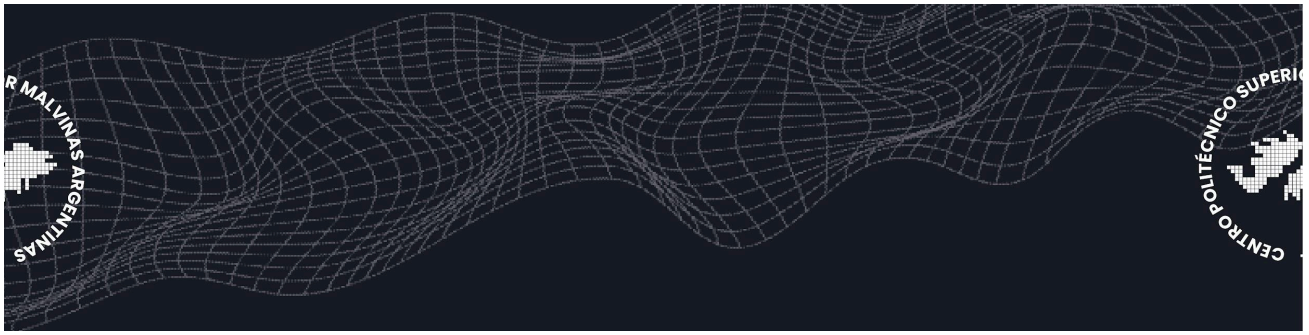
- Resalta Aspectos Relevantes: Utiliza la ordenación personalizada para destacar elementos que son particularmente importantes para tu análisis, incluso si no siguen un patrón de ordenación convencional.

4- Ordenación Basada en Atributos de Objetos:

- Analiza Propiedades Específicas: Organiza los objetos en función de atributos clave, lo que te permitirá obtener una comprensión más profunda de cómo se relacionan y difieren en términos de características específicas.

- Facilita la Selección: La ordenación basada en atributos te ayuda a encontrar rápidamente los objetos que cumplen ciertos criterios, lo que es útil en tareas de selección y filtrado.

En última instancia, la elección entre estos enfoques dependerá de tus objetivos y los tipos de datos con los que estés trabajando. Combinar estos métodos de manera estratégica puede ofrecer una visión más completa y valiosa de tus conjuntos de datos.

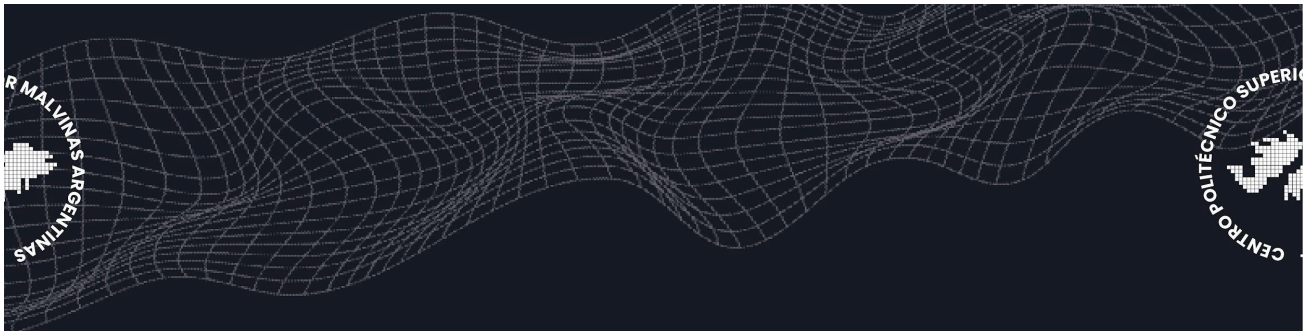


Bibliografía Obligatoria

- FUNDAMENTOS DE PROGRAMACIÓN. Algoritmos, estructura de datos y objetos-Cuarta edición - Luis Joyanes Aguilar- McGrawHill - 2008 - ISBN 978-84-481-6111-8

Bibliografía sugerida de la Unidad

- "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein - Este libro es ampliamente considerado como una referencia fundamental en el campo de los algoritmos. Cubre varios algoritmos de búsqueda, incluyendo búsqueda lineal y búsqueda binaria, y proporciona una base sólida en el diseño y análisis de algoritmos.
- "Algorithms, Part I" de Robert Sedgewick y Kevin Wayne - Este curso en línea y su libro asociado ofrecen una introducción detallada a los algoritmos y



estructuras de datos, incluyendo algoritmos de búsqueda. Cubre la búsqueda lineal, búsqueda binaria y otros algoritmos fundamentales.

- "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser - Este libro se centra en la implementación de estructuras de datos y algoritmos en Python. Incluye explicaciones y ejemplos de búsqueda lineal, búsqueda binaria y otras técnicas de búsqueda.
- "Algorithms, Part II" de Robert Sedgewick y Kevin Wayne - Este curso en línea y su libro complementario se enfocan en algoritmos más avanzados, incluyendo técnicas de búsqueda como la búsqueda de salto. Proporciona una cobertura detallada y práctica de algoritmos y estructuras de datos.

- "Introduction to the Design and Analysis of Algorithms" by Anany Levitin - Este libro ofrece una introducción accesible a los algoritmos y su análisis. Incluye una sección dedicada a los algoritmos de búsqueda y sus variantes.