ÍNDICE

1. Introducción

En el actual panorama de la ciencia de datos y el análisis de información, la capacidad de manipular y comprender los datos de manera eficiente es esencial para la toma de decisiones informadas. En este contexto, las bibliotecas de Python como NumPy y Pandas han emergido como herramientas fundamentales para llevar a cabo tareas de análisis de datos de manera efectiva.

NumPy es la biblioteca fundamental para la computación científica en Python la cual nos brinda las herramientas necesarias para trabajar con estructuras de datos multidimensionales, en particular, los arrays y matrices. A través de sus funciones universales, es posible realizar operaciones matemáticas y estadísticas en estos arrays, permitiéndonos obtener información valiosa de los datos de manera rápida y eficiente

Pandas se erige como una herramienta indispensable para el análisis de datos más avanzado. Esta biblioteca introduce dos estructuras esenciales: las Series y los DataFrames. Las Series, similares a arrays unidimensionales, pero con etiquetas, nos permiten trabajar con datos de manera más organizada y etiquetada. Por otro lado, los DataFrames, estructuras bidimensionales similares a tablas de bases de datos, nos ofrecen una forma flexible y potente de almacenar, manipular y analizar datos.

1.1. Objetivos

- Desarrollar una base sólida en el uso de Numpy y Pandas.
- Aplicación efectiva de funciones universales y estadísticas en Numpy.
- Dominar las operaciones de filtrado y manipulación en Pandas.
- Optimización y eficiencia en el manejo de datos
- Aplicar funciones estadísticas en pandas.

2. Numpy

NumPy (Numerical Python) es una biblioteca de programación en Python que proporciona soporte para realizar cálculos numéricos y operaciones matemáticas eficientes en matrices y arrays multidimensionales. Esta biblioteca es esencial en el ámbito del análisis de datos, la ciencia de datos y la computación científica debido a su capacidad para manejar grandes conjuntos de datos de manera rápida y eficiente. La característica principal de NumPy es su objeto fundamental, el "array", que es una estructura de datos multidimensional que permite almacenar elementos del mismo tipo. Estos arrays ofrecen ventajas significativas en términos de rendimiento y eficiencia en comparación con las listas de Python tradicionales, ya que están implementados en C y permiten la realización de operaciones vectorizadas y matriciales.

En resumen, NumPy proporciona:

- **Arrays Multidimensionales:** NumPy introduce el concepto de arrays multidimensionales, que son colecciones homogéneas de datos. Estos arrays pueden tener una, dos o más dimensiones, lo que los hace ideales para representar datos como matrices, imágenes y tensores.
- **Funciones Matemáticas Optimizadas:** NumPy proporciona una amplia gama de funciones matemáticas y operaciones que están optimizadas para trabajar con arrays. Estas funciones permiten realizar cálculos complejos de manera más rápida que utilizando bucles de Python estándar.
- **Broadcasting:** NumPy permite realizar operaciones entre arrays de diferentes tamaños y formas mediante el broadcasting, lo que facilita la realización de cálculos en datos que no tienen la misma estructura.
- **Integración con otras Bibliotecas:** NumPy es la base de muchas otras bibliotecas científicas en Python, como SciPy, pandas y scikit-learn. Esto significa que aprender NumPy es esencial para trabajar de manera efectiva con otras herramientas de análisis y visualización de datos.

2.1. Instalación y Configuración

A. Instalación NumPy

Para instalar NumPy en tu entorno de desarrollo, tienes dos opciones principales: utilizar el gestor de paquetes pip o conda

- **Pip**

```
pip install numpy
```

- **Conda,** si estás utilizando el entorno de Anaconda.

```
conda install numpy
```

B. Comprobación de la instalación y versión

Después de la instalación, es importante verificar que NumPy se haya instalado correctamente y conocer la versión que estás utilizando.

- Abre una terminal o consola de comandos.
- Inicia el intérprete de Python o IDE.
- Importa NumPy y verifica su versión:

```
import numpy as np
print("Versión de NumPy:", np.__version__)
```

Esto debería imprimir la versión de NumPy que has instalado.

C. Conceptos básicos de arrays en NumPy

- **Creación de arrays unidimensionales y multidimensionales:** En NumPy, los arrays son la piedra angular. Pueden ser unidimensionales (vectores) o multidimensionales (matrices). Para crear arrays, utilizamos la función numpy.array()

```
import numpy as np

# Crear un array unidimensional (vector)
vector = np.array([1, 2, 3, 4, 5])

# Crear un array bidimensional (matriz)
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
print('matriz \n', matriz)
print('vector \n', vector)
```

- **Acceso a elementos y rebanado (slicing) en arrays:** Los elementos en un array NumPy se indexan desde 0. Para acceder a un elemento específico, se utiliza el índice correspondiente.

```
print(vector[0])  # Acceder al primer elemento del vector (índice
0)
print(matriz[1, 1])  # Acceder al elemento en la fila 1 y columna
1 de la matriz
```

- **Propiedades de los arrays:** forma, tamaño, tipo de datos: Los arrays tienen propiedades que proporcionan información útil sobre ellos.

```
print('Forma del vector: ', vector.shape)
print('Forma de la matriz (filas, columnas): ', matriz.shape)
print('Tamaño total del vector: ', vector.size)
print('Tamaño total de la matriz: ', matriz.size)
print('Tipo de datos en el vector: ', vector.dtype)
print('Tipo de datos en la matriz: ', matriz.dtype)
```

2.2. Operaciones básicas con NumPy

A. Operaciones aritméticas y broadcasting

- **Realización de operaciones elementales con arrays:** NumPy permite realizar operaciones aritméticas de manera eficiente en arrays completos sin la necesidad de bucles explícitos.

```
import numpy as np

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

# Suma de arrays
suma = a + b
print('suma: ',suma)

# Resta de arrays
resta = a - b
```

```
print('resta: ',resta)

# Multiplicación de arrays
producto = a * b
print('producto: ',producto)

# División de arrays
division = a / b
print('division: ',division)
```

- **Broadcasting**: cómo NumPy trata operaciones con arrays de diferentes dimensiones, Broadcasting es una característica poderosa que permite realizar operaciones entre arrays de diferentes formas y dimensiones sin necesidad de que tengan las mismas dimensiones exactas.

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])

# Sumar una constante a cada fila de la matriz
resultado = matrix + [10, 20, 30]
print('Sumar: \n',resultado)

# Multiplicar cada columna por un factor
resultado = matrix * [[1], [2]]
print('Multiplicar: \n',resultado)
```

- Algunas de las funciones y características de NumPy

Creación de matrices y vectores:	
numpy.array	Crea una matriz o un vector a partir de una lista o una secuencia.
numpy.zeros	Crea una matriz de ceros con dimensiones especificadas.
numpy.ones	Crea una matriz de unos con dimensiones especificadas.
numpy.eye:	Crea una matriz identidad con dimensiones especificadas.
numpy.random.rand	Generan matrices o vectores con valores aleatorios.
numpy.random.randn	

Operaciones básicas:	
Suma (+) y resta (-) de matrices y vectores. Producto escalar: numpy.dot o el operador @. Producto elemento por elemento: numpy.multiply.	
Álgebra lineal básica:	
numpy.linalg.det	Calcula el determinante de una matriz.
numpy.linalg.inv	Calcula la inversa de una matriz.
numpy.linalg.eig	Calcula los autovalores y autovectores de una matriz.
numpy.linalg.solve	Resuelve sistemas de ecuaciones lineales Ax = b.
numpy.linalg.norm	Calcula la norma de una matriz o vector.
Descomposiciones matriciales:	
numpy.linalg.svd:	Descomposición en valores singulares.
numpy.linalg.qr:	Descomposición QR.
Reshape y Transposición:	
numpy.reshape:	Cambia la forma de una matriz o vector.
numpy.transpose o .T:	Transpone una matriz.
Funciones especiales:	
numpy.trace:	Calcula la traza de una matriz.
numpy.linalg.matrix_power	Calcula la potencia de una matriz.
Resolución de sistemas de ecuaciones lineales:	
numpy.linalg.solve:	Resuelve sistemas de ecuaciones lineales Ax = b.
Operaciones avanzadas:	
numpy.matmul o @:	Realiza multiplicación matricial.
numpy.linalg.pinv:	Calcula la pseudoinversa de una matriz.

B. Funciones Universales

Las funciones universales (ufuncs) en NumPy son funciones que operan elemento por elemento en los arrays NumPy, permitiendo realizar operaciones matemáticas y lógicas de manera eficiente. Estas funciones son esenciales para realizar cálculos numéricos en Python de manera vectorizada, lo que mejora significativamente la velocidad y eficiencia de las operaciones en comparación con los bucles tradicionales.

- **np.add():** Suma dos arrays elemento por elemento.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = np.add(arr1, arr2)  # [5, 7, 9]
result
```

- **np.subtract():** Resta dos arrays elemento por elemento.

```
arr1 = np.array([10, 20, 30])
arr2 = np.array([5, 10, 15])
result = np.subtract(arr1, arr2)  # [5, 10, 15]
result
```

- **np.multiply():** Multiplica dos arrays elemento por elemento.

```
arr1 = np.array([2, 3, 4])
arr2 = np.array([5, 6, 7])
result = np.multiply(arr1, arr2)  # [10, 18, 28]
result
```

- **np.divide():** Divide dos arrays elemento por elemento.

```
arr1 = np.array([10, 20, 30])
arr2 = np.array([2, 4, 6])
result = np.divide(arr1, arr2)  # [5.0, 5.0, 5.0]
result
```

- **np.exp():** Calcula la exponencial de cada elemento en un array.

```
arr = np.array([1, 2, 3])
result = np.exp(arr)  # [2.71828183, 7.3890561, 20.08553692]
result
```

- **np.log():** Calcula el logaritmo natural de cada elemento en un array.

```
arr = np.array([1, 10, 100])
result = np.log(arr)  # [0.0, 2.30258509, 4.60517019]
result
```

- **np.sin():** Calcula el seno de cada elemento en un array.

```
arr = np.array([0, np.pi/2, np.pi])
result = np.sin(arr)  # [0.0, 1.0, 1.2246468e-16]
result
```

- **np.cos():** Calcula el coseno de cada elemento en un array.

```
arr = np.array([0, np.pi/2, np.pi])
result = np.cos(arr)  # [1.0, 6.123233995736766e-17, -1.0]
result
```

- **np.sqrt():** Calcula la raíz cuadrada de cada elemento en un array.

```
arr = np.array([4, 9, 16])
result = np.sqrt(arr)  # [2.0, 3.0, 4.0]
result
```

- **np.maximum():** Devuelve el elemento máximo entre dos arrays elemento por elemento.

```
arr1 = np.array([3, 8, 5])
arr2 = np.array([6, 2, 9])
result = np.maximum(arr1, arr2)  # [6, 8, 9]
result
```

- **np.minimum():** Devuelve el elemento mínimo entre dos arrays elemento por elemento.

```
arr1 = np.array([3, 8, 5])
arr2 = np.array([6, 2, 9])
result = np.minimum(arr1, arr2)  # [3, 2, 5]
result
```

- **np.power():** Calcula la potencia de cada elemento en un array.

```
arr = np.array([2, 3, 4])
result = np.power(arr, 3)  # [8, 27, 64]
result
```

2.3. Indexado y selección avanzada

Es posible realizar indexado y selección avanzada para acceder y manipular elementos de matrices de formas diversas. Esto te permite realizar operaciones complejas y eficientes en conjuntos de datos multidimensionales.

- **Indexado básico:** El indexado en NumPy es similar al indexado en las listas de Python, pero con la posibilidad de trabajar con múltiples dimensiones. Puedes acceder a elementos individuales de una matriz utilizando índices.

```
import numpy as np

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

# Accediendo a elementos individuales
print(arr[0, 1])  # Output: 2
print(arr[2, 0])  # Output: 7
```

- **Slicing (rebanado):** Puedes utilizar el slicing para extraer subconjuntos de una matriz. El slicing en NumPy funciona similar al de las listas de Python.

```
# Extrayendo una submatriz
sub_matrix = arr[1:3, 1:3]
print(sub_matrix)
# Output: [[5 6] [8 9]]
```

- **Indexado booleano:** El indexado booleano te permite seleccionar elementos de una matriz basados en una condición booleana.

```
bool_mask = arr > 5
print(bool_mask)
# Output: [[False False False]
#          [False False  True]
#          [ True  True  True]]

selected_values = arr[bool_mask]
print(selected_values)  # Output: [6 7 8 9]
```

- **Selección avanzada con índices:** Puedes usar matrices de índices para seleccionar elementos específicos de una matriz.

```
row_indices = np.array([0, 1, 2])
col_indices = np.array([1, 2, 0])
selected_elements = arr[row_indices, col_indices]
print(selected_elements)  # Output: [2 6 7]
```

- **Selección por condición lógica:** La función np.where() te permite seleccionar elementos basados en una condición lógica.

```
selected_indices = np.where(arr > 5)
print(selected_indices)
# Output: (array([1, 2, 2, 2]), array([2, 0, 1, 2]))
```

- **Selección con el método np.take():** El método np.take() te permite seleccionar elementos de una matriz utilizando índices en forma de matriz.

```
indices = np.array([[0, 2],
                    [1, 2]])
selected_values = np.take(arr, indices)
print(selected_values)# Output: [[1 3]  [5 6]]
```

- **Indexación Fancy (Indexación Elegante):** La indexación elegante (fancy indexing) te permite seleccionar elementos de una matriz utilizando listas o matrices de índices. Esto te proporciona una forma poderosa de seleccionar elementos no contiguos o específicos de una matriz.

```
indices = [0, 2]
selected_rows = arr[indices]
print(selected_rows)  # Output: [[1 2 3]
                      #          [7 8 9]]
```

- **Indexación avanzada con máscaras booleanas:** Además de utilizar máscaras booleanas para seleccionar elementos, también puedes modificar elementos basados en condiciones lógicas.

```
arr[arr > 5] = 0
print(arr)
# Output: [[1 2 3]
#          [4 5 0]
#          [0 0 0]]
```

- **Indexación con el método np.ix_:** El método np.ix_ te permite seleccionar subconjuntos de una matriz utilizando índices para diferentes dimensiones en forma más conveniente.

```
row_indices = np.array([0, 2])
col_indices = np.array([1, 2])
selected_submatrix = arr[np.ix_(row_indices, col_indices)]
print(selected_submatrix)
# Output: [[2 3]
#          [0 0]]
```

- **Indexación con np.meshgrid():** La función np.meshgrid() se utiliza para generar matrices de coordenadas a partir de vectores unidimensionales. Esto es especialmente útil cuando necesitas evaluar una función en una malla de puntos. Puedes combinar las matrices generadas con np.meshgrid() para indexar elementos en una matriz multidimensional.

```
x = np.array([0, 1, 2])
y = np.array([10, 20, 30])
X, Y = np.meshgrid(x, y)

print(X)
# Output: [[0 1 2]
#          [0 1 2]
#          [0 1 2]]

print(Y)
# Output: [[10 10 10]
#          [20 20 20]
#          [30 30 30]]

# Indexando elementos utilizando las matrices X e Y
selected_elements = X + Y
print(selected_elements)
# Output: [[10 11 12]
#          [20 21 22]
#          [30 31 32]]
```

3. Pandas

Pandas es una biblioteca de Python ampliamente utilizada para el análisis y manipulación de datos. Proporciona estructuras de datos flexibles y eficientes, como Series y DataFrames, que permiten trabajar con datos de manera más conveniente. Pandas es esencial en el análisis de datos, ya que simplifica tareas como la carga de datos, la limpieza, la transformación y el análisis, lo que acelera el proceso de toma de decisiones informadas basadas en datos.

Importancia:

- **Facilita la manipulación de datos:** Pandas proporciona estructuras de datos que permiten almacenar y manipular datos de manera más intuitiva que las listas o arrays de NumPy. Esto simplifica tareas como filtrado, agrupación, transformación y agregación de datos.
- **Limpieza de datos:** Pandas ofrece herramientas para detectar y manejar valores faltantes, duplicados y errores en los datos, lo que es crucial para obtener resultados confiables en el análisis.
- **Análisis exploratorio:** Las funciones y métodos de Pandas permiten realizar análisis exploratorios de manera rápida, lo que ayuda a comprender la estructura y las características de los datos.
- **Integración con otras bibliotecas:** Pandas se integra fácilmente con otras bibliotecas populares de Python para análisis de datos, como Matplotlib, Seaborn y Scikit-learn, lo que proporciona una suite completa de herramientas para análisis y visualización.
- **Manipulación de series de tiempo:** Pandas está diseñado para trabajar de manera eficiente con series de tiempo, lo que lo convierte en una elección natural para análisis financieros y económicos.

3.1. Instalación y Configuración

A. Instalación Pandas

Para instalar Pandas en tu entorno de desarrollo, tienes dos opciones principales: utilizar el gestor de paquetes pip o conda

- **Pip**

```
pip install pandas
```

- **Conda,** si estás utilizando el entorno de Anaconda.

```
conda install pandas
```

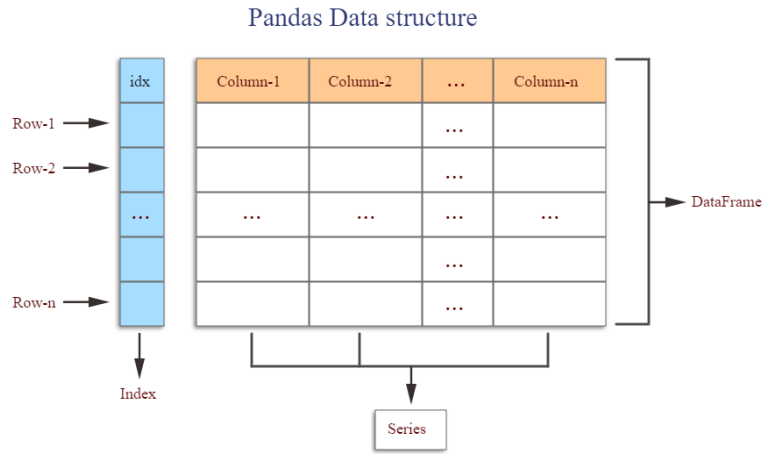
B. Comprobación de la instalación y versión

Después de la instalación, es importante verificar que Pandas se haya instalado correctamente y conocer la versión que estás utilizando.

- Abre una terminal o consola de comandos.
- Inicia el intérprete de Python o IDE.
- Importa Pandas y verifica su versión:

```
import pandas as pd
print("Versión de Pandas:", pd.__version__)
```

C. Estructuras de datos principales: Series y DataFrames



- **Series:** Una Serie es una estructura de datos unidimensional que puede contener datos de diversos tipos, como números, cadenas y fechas. Se asemeja a una columna en una hoja de cálculo y tiene etiquetas de índice para cada elemento. Para crear una Serie:

```
serie = pd.Series([10, 20, 30, 40, 50])
type(serie)
```

- **DataFrames:** Un DataFrame es una estructura de datos bidimensional, similar a una tabla en una base de datos o una hoja de cálculo. Está compuesto por una colección de Series y comparte un índice común. Para crear un DataFrame:

```
data = {'Nombre': ['Juan', 'María', 'Carlos'],
        'Edad': [25, 30, 28]}
df = pd.DataFrame(data)
type(df)
```

- **Index:** El objeto Index se utiliza para etiquetar filas o columnas en un DataFrame o Serie. Puede contener etiquetas únicas y es inmutable. En un DataFrame, hay dos objetos Index: el índice de filas y el índice de columnas.

```
import pandas as pd

data = {'Nombre': ['Alice', 'Bob', 'Charlie'],
        'Edad': [25, 30, 22]}
df = pd.DataFrame(data)
index = df.index
index
```

D. Selección de filas y columnas en DataFrames.

- **Selección de Columnas:** Puedes acceder a una columna específica utilizando la notación de corchetes [] y el nombre de la columna como una clave. Esto devolverá una Serie que contiene los valores de la columna seleccionada.

```
import pandas as pd

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)

column_A = df['A']  # Acceso a la columna 'A'
print(column_A)
# Output:
# 0    1
# 1    2
```

```
# 2    3
# Name: A, dtype: int64
```

- **Selección de Filas:** Puedes acceder a filas específicas utilizando la notación de corchetes [] y un rango de índices o una condición booleana. Esto devolverá un DataFrame que contiene las filas seleccionadas.

```
import pandas as pd
data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
index_labels = ['row1', 'row2', 'row3']
df = pd.DataFrame(data, index=index_labels)

# Acceso a filas utilizando índices
subset_rows = df[1:3]  # Filas con índices 1 y 2 (excluye 3)
print(subset_rows)
# Output:
#        A  B
# row2  2  5
# row3  3  6
```

E. índices

Un índice en Pandas es una estructura que etiqueta y proporciona acceso eficiente a las filas de un DataFrame o a los elementos de una Serie. Los índices son una parte esencial de las estructuras de datos en Pandas, ya que permiten un acceso rápido a los datos basado en estas etiquetas en lugar de depender de las posiciones numéricas.

Tipos de índices en Pandas

- **Index por defecto:** Cuando creas un DataFrame sin especificar un índice, Pandas asigna un índice por defecto que es simplemente una secuencia numérica que comienza desde 0.

```
import pandas as pd
data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
df = pd.DataFrame(data)
print(df)
# Output:
```

```
#    A  B
# 0  1  4
# 1  2  5
# 2  3  6
```

- **Index personalizado:** Puedes establecer tus propias etiquetas de índice al crear un DataFrame. Esto es útil cuando deseas etiquetar tus filas de manera más significativa, como usar fechas o nombres únicos.

```
import pandas as pd

data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}
index_labels = ['row1', 'row2', 'row3']
df = pd.DataFrame(data, index=index_labels)
print(df)
# Output:
#        A  B
# row1  1  4
# row2  2  5
# row3  3  6
```

- **MultiIndex:** El MultiIndex es una extensión del Index que permite tener múltiples niveles de etiquetas en un índice. Se utiliza para manejar datos con múltiples dimensiones y es especialmente útil para indexar y organizar datos jerárquicos.

```
import pandas as pd

data = {'Ventas': [100, 150, 200, 120, 180],
        'Gastos': [70, 100, 90, 80, 120]}
index = pd.MultiIndex.from_tuples([('Q1', 'Enero'), ('Q1',
'Febrero'), ('Q2', 'Marzo'), ('Q2', 'Abril'), ('Q2', 'Mayo')],
                                  names=['Trimestre', 'Mes'])
df = pd.DataFrame(data, index=index)
df
```

Características clave de los índices:

- **Etiquetas únicas:** Cada etiqueta de índice debe ser única, lo que garantiza una identificación única para cada fila o elemento.
- **Inmutabilidad:** Una vez que se crea un índice, generalmente no se puede modificar directamente. Esto garantiza la coherencia y la integridad de los datos.
- **Facilita la alineación de datos:** Los índices permiten la alineación automática de datos cuando se realizan operaciones entre DataFrames o Series diferentes. Esto es especialmente útil cuando se combinan o se realizan operaciones aritméticas.
- **Acceso eficiente:** Los índices proporcionan un acceso rápido a los datos utilizando las etiquetas en lugar de las posiciones numéricas.

F. Carga de datos desde diferentes fuentes

Puede cargar datos desde diferentes fuentes utilizando diversas funciones proporcionadas por Pandas. Aquí hay ejemplos de cómo cargar datos desde diferentes fuentes:

- CSV

```
import pandas as pd

# Cargar datos desde un archivo CSV
data = pd.read_csv('sueldo_funcionarios_2019.csv')
```

- EXCEL

```
# Asegúrate de tener la biblioteca `openpyxl` instalada: pip
install openpyxl
import pandas as pd

# Cargar datos desde una hoja específica de un archivo Excel
data = pd.read_excel('sueldo_funcionarios_2019.xlsx',
sheet_name='sueldo_funcionarios_2019')
```

- URL

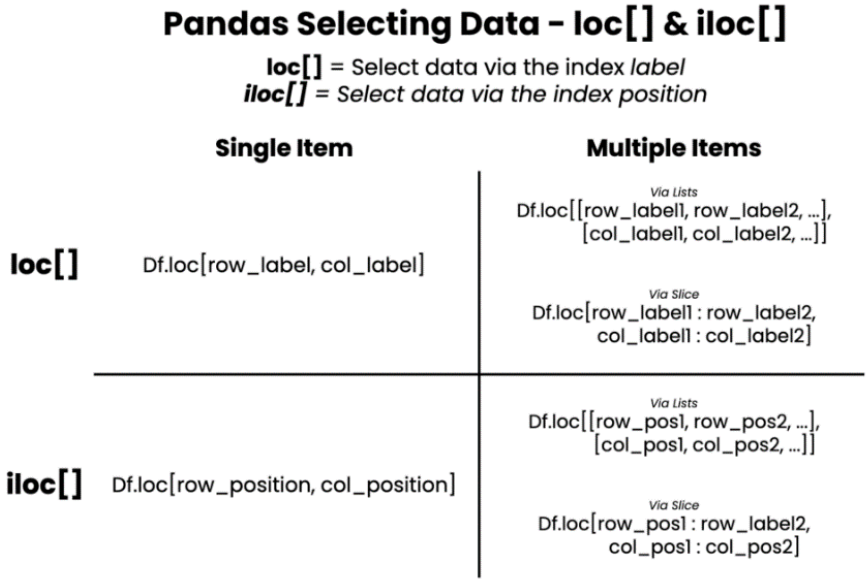
```
import pandas as pd

# Cargar datos desde una URL
url =
'http://cdn.buenosaires.gob.ar/datosabiertos/datasets/sueldo-
funcionarios/sueldo_funcionarios_2019.csv'
data = pd.read_csv(url)
```

3.2. Manipulación de Datos con Pandas

La manipulación de datos es una parte esencial del análisis de datos y la ciencia de datos en general. Pandas es una biblioteca de Python ampliamente utilizada para la manipulación y análisis de datos. Proporciona estructuras de datos flexibles y eficientes, como DataFrames y Series, que permiten trabajar con datos tabulares y de series temporales de manera eficiente.

A. Indexing

el "indexing" se refiere a la forma en que accedes y manipulas los datos almacenados en un DataFrame o una Serie. El "indexing" detallado en Pandas se puede dividir en dos categorías principales: indexación por etiquetas (label-based indexing), e indexación por posición (integer-based indexing).



- **Indexación por etiquetas (Label-Based Indexing):** En la indexación por etiquetas, utilizas las etiquetas de las filas y las columnas para acceder a los datos. El índice de fila y el índice de columna se pueden

establecer de manera personalizada. Algunos métodos y atributos utilizados para la indexación por etiquetas son:

```
import pandas as pd
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
}

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])
```

- **.loc[row_label, column_label]:** Accede a un elemento o a un subconjunto de datos utilizando las etiquetas de fila y columna.

```
element = df.loc['row2', 'B']
print(element)  # Output: 5
```

- **.at[row_label, column_label]:** Accede a un único elemento utilizando las etiquetas de fila y columna, similar a .loc pero más rápido para acceder a un solo valor.

```
element = df.at['row2', 'B']
print(element)  # Output: 5
```

- **.loc[row_label]:** Accede a una fila completa utilizando la etiqueta de fila.

```
row_data = df.loc['row1']
print(row_data)
```

- **.loc[:, column_label]:** Accede a una columna completa utilizando la etiqueta de columna.

```
column_data = df.loc[:, 'B']
print(column_data)
```

- **Indexación por posición (Integer-Based Indexing):** En la indexación por posición, utilizas las posiciones numéricas de las filas y columnas para acceder a los datos. Las filas y columnas se numeran desde 0. Algunos métodos y atributos utilizados para la indexación por posición son:
    - **.iloc[row_index, column_index]:** Accede a un elemento o a un subconjunto de datos utilizando las posiciones numéricas de fila y columna.

```
element = df.iloc[1, 2]
print(element)
```

    - **.iat[row_index, column_index]:** Accede a un único elemento utilizando las posiciones numéricas de fila y columna, similar a .iloc pero más rápido para acceder a un solo valor.

```
element = df.iat[1, 2]
print(element)  # Output: 8
```

    - **.iloc[row_index]:** Accede a una fila completa utilizando la posición numérica de fila.

```
row_data = df.iloc[0]
print(row_data)
```

    - **.iloc[:, column_index]:** Accede a una columna completa utilizando la posición numérica de columna.

```
column_data = df.iloc[:, 1]
print(column_data)
```

B. Filtros con Pandas

Filtrar datos es una tarea común en el análisis de datos utilizando la biblioteca Pandas en Python. Pandas ofrece métodos muy útiles para filtrar filas o columnas de un DataFrame según ciertas condiciones.

```
import pandas as pd
df = pd.read_csv('sueldo_funcionarios_2019.csv')
```

- **Simple:** Filtrar filas basadas en una única condición

```
condicion_salario = df['total_salario_bruto_i_+_ii'] > 200000
df[condicion_salario]
```

- **Filtrado por una condición de cadena:** Filtrar filas basadas en una condición de cadena en una columna.

```
condicion_nombre = df['funcionario_apellido'].str.startswith('C')
df[condicion_nombre]
```

- **Filtrado por múltiples condiciones:** Filtrar filas basadas en múltiples condiciones utilizando operadores lógicos.

```
condicion_multiple = (df['total_salario_bruto_i_+_ii'] > 200000)
& (df['mes'] == 12)
df[condicion_multiple]
```

- **Filtrado por valores en una lista:** Filtrar filas basadas en si los valores de una columna están en una lista dada.

```
valores_deseados = ['Ministerio de Salud', 'SECR de Medios']
condicion_lista = df['reparticion'].isin(valores_deseados)
df[condicion_lista]
```

- **Filtrado por valores nulos:** Filtrar filas basadas en si una columna tiene valores nulos.

```
condicion_nulos = df['aguinaldo_ii'].isna()
df[condicion_nulos]
```

- **Filtrado inverso:** Filtrar filas que no cumplen una cierta condición utilizando el operador ~.

```
condicion_inversa = ~df['funcionario_apellido'].str.contains('MACCHIAVELLI')
df[condicion_inversa]
```

- **Filtrado por categoría de texto:** Filtrar filas basadas en una categoría específica en una columna categórica.

```
categoria_deseada = 'Ministerio de Salud'
condicion_categoria = df['reparticion'] == categoria_deseada
df[condicion_categoria]
```

- **Filtrado por condición compleja:** Filtrar filas basadas en condiciones complejas usando paréntesis para agrupar condiciones.

```
condicion_compleja = ((df['total_salario_bruto_i_+_ii'] > 200000)
| (df['aguinaldo_ii'] > 15000)) & (df['mes'] == 12)
df[condicion_compleja]
```

- **query():** Filtrar usando query() con condiciones de cadena y operador OR

```
df.query("(funcionario_nombre == 'FERNAN') | (funcionario_nombre
== 'CHRISTIAN')")
```

- **Usar métodos .query() con variables:** Puedes definir variables en tu entorno y utilizarlas en tus consultas .query() para facilitar el filtrado.

```
monto_minimo = 200000
df.query("`total_salario_bruto_i_+_ii` > @monto_minimo")
```

Si estás tratando de usar la función query en un DataFrame de pandas y el nombre de la columna contiene caracteres reservados o espacios, puedes utilizar comillas invertidas (`) para rodear el nombre de la columna.

- **Usar el método .query() con condiciones más complejas:** Puedes aplicar condiciones más complejas utilizando la función .query() y aprovechar las variables definidas en tu entorno.

```
min_salario = 180000
max_mes = 6
df.query("asignacion_por_cargo_i > @min_salario & mes <=
@max_mes")
```

- **Usar el método .eval() para evaluar expresiones: Puedes utilizar este método para evaluar expresiones en el contexto del DataFrame y filtrar según el resultado.**

```
df[df.eval("aguinaldo_ii > `total_salario_bruto_i_+_ii` * 0.2")]
```

- **Usar métodos .apply() o .map() para aplicar funciones personalizadas:** Puedes utilizar estos métodos para aplicar funciones a columnas y filtrar filas basadas en el resultado de la función.

```
def funcion_filtrado(row):
    return row['total_salario_bruto_i_+_ii'] > 200000

df[df.apply(funcion_filtrado, axis=1)]
```

C. Vectorización con Pandas

La "vectorización" en el contexto de pandas se refiere a aplicar operaciones en forma de vectores (arrays) en lugar de utilizar bucles explícitos para manipular datos en un DataFrame. Esto es importante porque pandas está optimizado para operar en columnas completas de datos de manera eficiente, en lugar de operar en elementos individuales.

En lugar de iterar a través de filas o columnas de un DataFrame usando bucles, puedes utilizar operaciones vectorizadas que son mucho más eficientes y rápidas. Las operaciones vectorizadas están implementadas en las funciones de pandas y NumPy, que es una biblioteca de álgebra lineal en Python.

- **Operaciones aritméticas:** En lugar de iterar a través de las filas de un DataFrame para realizar operaciones aritméticas, puedes simplemente aplicar las operaciones directamente a las columnas:

```
import pandas as pd
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
# Operación aritmética vectorizada
df['C'] = df['A'] + df['B']
df['C']
```

- **Funciones de transformación:** En lugar de aplicar una función a cada elemento en un bucle, puedes usar funciones vectorizadas de pandas:

```
# Aplicar una función a una columna
df['D'] = df['C'].apply(lambda x: x * 2)
df['D']
```

- **Funciones de agregación:** En lugar de iterar a través de filas para calcular sumas, promedios u otras estadísticas, puedes usar funciones de agregación de pandas:

```
# Calcular la suma y el promedio de una columna
sum_c = df['C'].sum()
avg_c = df['C'].mean()
sum_c, avg_c
```

- **Uso de operaciones booleanas:** Las operaciones booleanas también se pueden vectorizar para filtrar datos:

```
# Filtrar filas basadas en una condición
filtered_df = df[df['C'] > 8]
filtered_df
```

¿Cuál es la relevancia de utilizar Pandas en lugar de crear nuestras propias funciones en el lenguaje Python y procesar datos mediante bucles 'for' de manera nativa?

Veamos una demostración de que vectorizar es más eficiente. Vamos a crear dos listas de 1.000.000 de números aleatorios cada una y vamos a tratar de multiplicar elemento por elemento con pandas y sin pandas:

```python
import pandas as pd
from numpy.random import random
import time

# Crear dos listas de 1,000,000 de números aleatorios
num_elements = 1000000
data1, data2 = random(num_elements), random(num_elements)

# Utilizando Pandas para la multiplicación
df1, df2 = pd.Series(data1), pd.Series(data2)
start_time = time.time()
result_pandas = df1 * df2
pandas_time = time.time() - start_time

# Utilizando bucle 'for' para la multiplicación sin Pandas
result_native = []
start_time = time.time()
for i in range(num_elements):
    result_native.append(data1[i] * data2[i])
native_time = time.time() - start_time

print(f"Tiempo con Pandas: {pandas_time:.6f} segundos")
print(f"Tiempo sin Pandas: {native_time:.6f} segundos")
```

Otra forma de medir el tiempo de ejecución de código en Jupyter Notebook:

```
%timeit result_pandas = df1 * df2
```

```
%%time
for i in range(num_elements):
    result_native.append(data1[i] * data2[i])
```

Al utilizar %timeit antes de una línea de código, Jupyter Notebook ejecutará esa línea varias veces y calculará un tiempo promedio de ejecución. Por otro lado, %%time al principio de una celda calcula el tiempo de ejecución de la celda completa.

3.3. Análisis Exploratorio de Datos

El análisis exploratorio de datos (EDA, por sus siglas en inglés) es una fase crucial en cualquier proyecto de análisis de datos, ya que te permite comprender mejor tus datos, identificar patrones, detectar valores atípicos y formular hipótesis iniciales antes de aplicar técnicas más avanzadas de modelado. La librería de Python llamada pandas es ampliamente utilizada para llevar a cabo el análisis exploratorio de datos debido a su facilidad de uso y capacidades de manipulación de datos.

```
import pandas as pd
df = pd.read_csv('sueldo_funcionarios_2019.csv')
```

A. Funciones básicas

- Mostrar las primeras filas del DataFrame

```
df.head(5)
```

- Mostrar las ultimas filas del DataFrame

```
df.tail(5)
```

- Mostrar las filas aleatorias del DataFrame

```
df.sample(5)
```

- Obtener información general sobre el DataFrame

```
df.info()
```

- Mostrar las propiedades del DataFrame

```
# La propiedad shape nos devuelve una tupla (filas,columnas)
df.shape
```

- Mostrar las columnas del DataFrame

```
df.columns
```

- Estadísticas descriptivas de las columnas numéricas

```
df.describe()
```

- Contar los valores únicos en una columna categórica

```
df['reparticion'].value_counts()
```

- Matriz de correlación para entender las relaciones entre columnas numéricas

```
df.select_dtypes(include=['number']).corr()
```

B. Funciones de Agregación básicas

- mean(): Calcula el promedio de los valores.
- sum(): Calcula la suma de los valores.
- min(): Encuentra el valor mínimo.
- max(): Encuentra el valor máximo.
- count(): Cuenta la cantidad de valores no nulos.

```
# Ejemplos de funciones de agregación básicas
promedio = df['asignacion_por_cargo_i'].mean()
suma = df['asignacion_por_cargo_i'].sum()
minimo = df['aguinaldo_ii'].min()
maximo = df['aguinaldo_ii'].max()
conteo = df['funcionario_nombre'].count()

print(f"Promedio: {promedio}")
print(f"Suma: {suma}")
print(f"Mínimo: {minimo}")
print(f"Máximo: {maximo}")
print(f"Conteo: {conteo}")
```

Otras funciones de agregación:

- median(): Calcula la mediana de los valores.
- std(): Calcula la desviación estándar.
- var(): Calcula la varianza.
- quantile(q): Calcula el percentil q de los valores.
- agg(func): Permite aplicar múltiples funciones de agregación a la vez.

```
# Ejemplos de otras funciones de agregación
mediana = df['asignacion_por_cargo_i'].median()
desviacion_estandar = df['asignacion_por_cargo_i'].std()
varianza = df['aguinaldo_ii'].var()
percentil_75 = df['aguinaldo_ii'].quantile(0.75)

print(f"Mediana: {mediana}")
print(f"Desviación Estándar: {desviacion_estandar}")
print(f"Varianza: {varianza}")
print(f"Percentil 75: {percentil_75}")
```

- **Funciones de agregación con groupby:** Puedes usar la función groupby() junto con las funciones de agregación para calcular estadísticas agrupadas por categorías.

```
# Ejemplo de agregación con groupby
agregaciones_por_categoria =
df.groupby('mes')['asignacion_por_cargo_i'].mean()
print(agregaciones_por_categoria)
```

4. Manejo de ficheros

Pandas es una biblioteca de Python ampliamente utilizada para el análisis y manipulación de datos. Proporciona herramientas eficientes y flexibles para trabajar con datos tabulares, incluido el manejo de ficheros. Aquí te proporcionaré una introducción al manejo de ficheros con Pandas, incluyendo la lectura, escritura y algunos parámetros útiles.

4.1. Lectura:

Pandas permite leer una variedad de formatos de ficheros, como CSV, Excel, JSON, SQL, entre otros. Aquí tienes ejemplos de cómo leer diferentes tipos de ficheros:

Tipo	Codigo
CSV	`df_csv = pd.read_csv('archivo.csv')`
Excel	`df_excel = pd.read_excel('archivo.xlsx', sheet_name='Hoja1')`
JSON	`df_json = pd.read_json('archivo.json')`

información adicional basada en ficheros CSV

- **Personalización de columnas:** Si solo necesitas cargar un subconjunto de columnas de un archivo, puedes especificar las columnas a leer utilizando el parámetro usecols.

```
import pandas as pd
cols_to_read = ['columna1', 'columna2']
data = pd.read_csv('archivo.csv', usecols=cols_to_read)
```

- **Manipulación de tipos de datos:** Pandas intenta inferir los tipos de datos al leer archivos, pero puedes especificarlos explícitamente usando el parámetro dtype.

```
dtypes = {'columna1': 'int32', 'columna2': 'float64'}
data = pd.read_csv('archivo.csv', dtype=dtypes)
```

- **Tratamiento de valores faltantes:** Usa el parámetro na_values para especificar cómo manejar valores faltantes durante la lectura.

```
data = pd.read_csv('archivo.csv', na_values=['N/A', 'NULL'])
```

- **Lectura en lotes (chunking):** Si estás trabajando con archivos muy grandes, puedes leerlos en fragmentos más pequeños usando el parámetro chunksize.

```
chunk_size = 1000
chunks = pd.read_csv('archivo.csv', chunksize=chunk_size)
for chunk in chunks:
    # Procesar cada fragmento (chunk) aquí
```

4.2. Escritura

Tipo	Codigo
CSV	`df.to_csv('nuevo_archivo.csv', index=False)`
Excel	`df.to_excel('nuevo_archivo.xlsx', index=False)`
JSON	`df.to_json('nuevo_archivo.json', orient='records')`

Guardado en otros formatos: Pandas puede guardar datos en varios formatos como Excel, JSON, HDF5, Parquet, etc.

```
data.to_excel('datos.xlsx', sheet_name='hoja_datos', index=False)
data.to_json('datos.json', orient='records')
data.to_hdf('datos.h5', key='data', mode='w')
```

Escritura en varias hojas de Excel: Cuando guardas en un archivo Excel, puedes escribir en diferentes hojas utilizando un objeto ExcelWriter.

```
with pd.ExcelWriter('datos.xlsx') as writer:
    data1.to_excel(writer, sheet_name='hoja1', index=False)
    data2.to_excel(writer, sheet_name='hoja2', index=False)
```