Python - Pandas

Pandas es una poderosa biblioteca de Python utilizada para el análisis de datos. Proporciona estructuras de datos flexibles y eficientes, así como herramientas para manipular y analizar datos de manera rápida y sencilla. Para comprender plenamente pandas, es importante familiarizarse con algunos conceptos teóricos clave que subyacen en su funcionamiento. A continuación, se presenta una introducción detallada sobre estos conceptos:

1. Estructuras de datos:
   Pandas se basa en dos estructuras de datos principales: Series y DataFrame. Una Serie es una estructura unidimensional que contiene datos homogéneos o heterogéneos, mientras que un DataFrame es una estructura bidimensional que almacena datos en filas y columnas. Estas estructuras proporcionan una forma intuitiva y flexible de representar y manipular datos.
2. Indexación y etiquetado:
   El indexado en pandas es esencial para acceder y manipular los datos. Cada Serie o columna en un DataFrame tiene un índice, que puede ser numérico o basado en etiquetas. El índice permite la identificación única de elementos en la estructura de datos y facilita la manipulación y recuperación eficiente de datos.
3. Operaciones vectorizadas:
   Pandas aprovecha el concepto de operaciones vectorizadas, lo que significa que las operaciones se aplican a todo el conjunto de datos en lugar de iterar elemento por elemento. Esto permite un procesamiento más rápido y eficiente de grandes conjuntos de datos, ya que evita bucles explícitos en Python.
4. Tratamiento de datos faltantes:
   Los datos faltantes son comunes en conjuntos de datos reales. Pandas proporciona métodos y herramientas para detectar, manejar y limpiar datos faltantes de manera efectiva. Puedes eliminar filas o columnas con datos faltantes, rellenarlos con valores específicos o utilizar técnicas más avanzadas, como la interpolación, para completar los valores faltantes.
5. Agrupación y agregación:
   Pandas permite realizar operaciones de agrupación y agregación en conjuntos de

datos basados en una o varias columnas. Esto implica dividir los datos en grupos según criterios específicos, aplicar funciones de agregación (como suma, promedio o conteo) a cada grupo y combinar los resultados en una nueva estructura de datos. Esta funcionalidad es especialmente útil para el análisis estadístico y el resumen de datos.

6. Operaciones de fusión y unión:
   Pandas facilita la combinación de múltiples conjuntos de datos utilizando operaciones de fusión y unión. La fusión permite combinar conjuntos de datos en función de columnas comunes, mientras que la unión permite combinar conjuntos de datos en función de índices comunes. Estas operaciones son útiles cuando se trabaja con datos dispersos en múltiples archivos o fuentes de datos diferentes.
7. Transformación de datos:
   Pandas proporciona una amplia gama de funciones y métodos para transformar datos. Puedes realizar operaciones de filtrado para seleccionar filas o columnas específicas, aplicar funciones a columnas o filas completas utilizando expresiones lambda o funciones definidas por el usuario, y realizar operaciones de mapeo para asignar valores basados en ciertos criterios.
8. Visualización de datos:
   Pandas se integra de manera fluida con bibliotecas de visualización de datos populares, como Matplotlib y Seaborn, lo que facilita la creación de gráficos y visualizaciones atractivas. Estas herramientas permiten explorar y comunicar de manera efectiva los patrones y tendencias presentes en los datos, lo que resulta fundamental en el análisis de datos.
9. Manejo de fechas y tiempo:
   Pandas tiene capacidades incorporadas para trabajar con datos de series de tiempo. Puedes realizar operaciones de indexación y filtrado basadas en fechas y horas, así como realizar cálculos y agregaciones temporales. Pandas también ofrece funciones para la resampleo de datos en diferentes frecuencias de tiempo y el manejo de zonas horarias.
10. Optimización y rendimiento:
    Cuando se trabaja con grandes conjuntos de datos, el rendimiento y la optimización son consideraciones importantes. Pandas ofrece técnicas y herramientas para mejorar el rendimiento, como la selección y filtrado eficiente de datos, el uso

adecuado de tipos de datos y la iteración optimizada sobre las estructuras de datos.

11. Interoperabilidad con otras bibliotecas:
    Pandas se integra bien con otras bibliotecas y herramientas de análisis de datos en el ecosistema de Python. Por ejemplo, puedes combinar pandas con NumPy para realizar operaciones numéricas eficientes, scikit-learn para realizar análisis de machine learning o SQLalchemy para interactuar con bases de datos SQL.

En resumen, pandas es una biblioteca esencial en el análisis de datos en Python. Su capacidad para gestionar eficientemente estructuras de datos, manipular datos faltantes, realizar operaciones de agregación y transformación, y visualizar datos, lo convierten en una herramienta poderosa para el análisis y la manipulación de datos. Con su amplia gama de funciones y su integración con otras bibliotecas, pandas se ha convertido en una opción popular y versátil para los científicos de datos y los profesionales del análisis de datos.

Introducción a pandas:

1. ¿Qué es pandas y por qué es importante en el análisis de datos?
   Pandas es una biblioteca de Python ampliamente utilizada para el análisis de datos. Proporciona estructuras de datos de alto rendimiento y herramientas de manipulación de datos que permiten trabajar con conjuntos de datos estructurados de manera eficiente. Algunas de las razones por las cuales pandas es importante en el análisis de datos son:

```
import pandas as pd

# Crear un DataFrame con datos imaginarios
data = {
    'Nombre': ['Juan', 'María', 'Carlos', 'Laura', 'Pedro'],
    'Edad': [28, 34, 29, 42, 36],
    'Ciudad': ['Buenos Aires', 'Madrid', 'Lima', 'Sao Paulo', 'México DF'],
    'Puntuación': [75, 82, 88, 95, 67]
}

df = pd.DataFrame(data)

# Mostrar el DataFrame completo
print(df)
```

En este ejemplo, se crea un DataFrame utilizando un diccionario `data` que contiene información ficticia sobre personas. Pandas permite manejar y estructurar estos datos en forma tabular, lo que facilita su análisis y manipulación. Además, pandas proporciona numerosas funciones y métodos para realizar operaciones comunes en el análisis de datos, como filtrar, agrupar, unir y transformar datos.

2. Instalación de pandas y sus dependencias:
   Para instalar pandas y sus dependencias, puedes utilizar pip, el gestor de paquetes de Python. Abre una terminal y ejecuta el siguiente comando:

   ```
   pip install pandas
   ```

   Este comando instalará pandas en tu entorno de Python, junto con sus dependencias necesarias, como NumPy. Asegúrate de tener una conexión a Internet activa para descargar los paquetes necesarios. Una vez instalado, puedes importar pandas en tus scripts de Python para utilizarlo en el análisis de datos.

3. Importación de la biblioteca y la convención de alias:

   ```
   import pandas as pd
   ```

   En este ejemplo, importamos la biblioteca pandas utilizando la declaración `import pandas`. Además, utilizamos la convención de alias `as pd`, lo que nos permite referirnos a pandas usando el alias `pd`. Esta convención es ampliamente adoptada en la comunidad de pandas y simplifica el acceso a las funciones y métodos de la biblioteca.

4. Estructuras de datos en pandas:

   a. Series: una estructura unidimensional que puede contener diferentes tipos de datos:

   ```
   import pandas as pd
   ```

```
# Crear una serie con datos imaginarios
data = [10, 20, 30, 40, 50]
serie = pd.Series(data)

# Mostrar la serie
print(serie)
```

En este ejemplo, creamos una serie llamada `serie` utilizando la función `pd.Series()`. La serie contiene una secuencia de números del 10 al 50. Las series pueden contener diferentes tipos de datos, como números, cadenas, booleanos, etc. La serie se muestra en la consola, donde se muestra el índice de la serie (por defecto, un rango numérico) y los valores correspondientes.

b. DataFrame: una estructura tabular bidimensional con columnas etiquetadas y filas indexadas:

```
import pandas as pd

# Crear un DataFrame con datos imaginarios
data = {
    'Nombre': ['Juan', 'María', 'Carlos', 'Laura', 'Pedro'],
    'Edad': [28, 34, 29, 42, 36],
    'Ciudad': ['Buenos Aires', 'Madrid', 'Lima', 'Sao Paulo', 'México DF']
}

df = pd.DataFrame(data)

# Mostrar el DataFrame
print(df)
```

En este ejemplo, creamos un DataFrame llamado `df` utilizando un diccionario `data` que contiene información ficticia sobre personas. El DataFrame tiene tres columnas etiquetadas ('Nombre', 'Edad' y 'Ciudad') y cada fila está indexada. El DataFrame se muestra en la consola, mostrando los nombres de las columnas y los datos correspondientes en forma tabular.

5. Lectura y escritura de datos:
    a. Carga de datos desde diferentes fuentes (CSV, Excel, bases de datos, etc.):
    - Carga desde un archivo CSV:

```
import pandas as pd

# Cargar datos desde un archivo CSV
df = pd.read_csv('archivo.csv')

# Mostrar el DataFrame cargado desde el archivo CSV
print(df)
```

- Carga desde un archivo Excel:

```
import pandas as pd

# Cargar datos desde un archivo Excel
df = pd.read_excel('archivo.xlsx', sheet_name='Hoja1')

# Mostrar el DataFrame cargado desde el archivo Excel
print(df)
```

- Carga desde una base de datos (utilizando SQLAlchemy):

```
import pandas as pd
from sqlalchemy import create_engine

# Conexión a la base de datos
engine = create_engine('postgresql://usuario:contraseña@localhost:5432/nombre_bd')

# Consulta SQL para obtener los datos
query = 'SELECT * FROM tabla'

# Cargar datos desde la base de datos
df = pd.read_sql_query(query, engine)

# Mostrar el DataFrame cargado desde la base de datos
print(df)
```

6. Exploración de los métodos de lectura y escritura en pandas:

- Lectura desde una fuente de datos:

```
import pandas as pd

# Lectura desde un archivo CSV
df_csv = pd.read_csv('archivo.csv')

# Lectura desde un archivo Excel
df_excel = pd.read_excel('archivo.xlsx', sheet_name='Hoja1')

# Lectura desde una base de datos
df_db = pd.read_sql_query('SELECT * FROM tabla', engine)
```

- Escritura a una fuente de datos:

```
import pandas as pd

# Escritura a un archivo CSV
df.to_csv('nuevo_archivo.csv', index=False)

# Escritura a un archivo Excel
df.to_excel('nuevo_archivo.xlsx', sheet_name='Hoja1', index=False)

# Escritura a una base de datos
df.to_sql('nueva_tabla', engine, if_exists='replace', index=False)
```

En estos ejemplos, utilizamos los métodos `read_csv()`, `read_excel()`, `read_sql_query()`, `to_csv()`, `to_excel()` y `to_sql()` de pandas para leer y escribir datos desde y hacia diferentes fuentes. Puedes adaptar estos ejemplos según tus necesidades y la fuente de datos específica con la que estés trabajando.

1. Manipulación y limpieza de datos:
    a. Selección de columnas y filas relevantes:

```
import pandas as pd

# Selección de columnas
columnas_relevantes = ['Nombre', 'Edad']
df_columnas = df[columnas_relevantes]

# Selección de filas basada en una condición
```

```
df_filtrado = df[df['Edad'] > 30]
```

En este ejemplo, `df_columnas` contendrá solo las columnas 'Nombre' y 'Edad' del DataFrame original `df`, mientras que `df_filtrado` contendrá solo las filas donde la columna 'Edad' es mayor a 30.

b. Manipulación de datos faltantes o nulos:

```
# Verificación de valores nulos en el DataFrame
print(df.isnull())

# Eliminación de filas con valores nulos
df_sin_nulos = df.dropna()

# Relleno de valores nulos con un valor específico
df_rellenado = df.fillna(0)
```

El método `isnull()` verifica si hay valores nulos en el DataFrame. `dropna()` elimina las filas que contienen valores nulos, mientras que `fillna()` rellena los valores nulos con el valor especificado (en este caso, 0).

1. Eliminación de duplicados:

```
# Eliminación de filas duplicadas
df_sin_duplicados = df.drop_duplicates()
```

El método `drop_duplicates()` elimina las filas duplicadas del DataFrame, manteniendo solo la primera aparición de cada fila única.

1. Cambio de tipos de datos:

```
# Cambio de tipo de dato de una columna
df['Edad'] = df['Edad'].astype(float)
```

El método `astype()` se utiliza para cambiar el tipo de dato de una columna específica. En este ejemplo, convertimos la columna 'Edad' a tipo de dato float.

1. Renombrar columnas:

```
# Renombrar una columna
df = df.rename(columns={'Edad': 'Años'})
```

El método `rename()` se utiliza para cambiar el nombre de una o varias columnas. En este ejemplo, renombramos la columna 'Edad' a 'Años'.

1. Filtrado y reordenamiento de datos:

```
# Filtrar datos basado en múltiples condiciones
df_filtrado = df[(df['Edad'] > 30) & (df['Puntuación'] > 80)]

# Reordenar el DataFrame por una columna específica
df_ordenado = df.sort_values('Edad')
```

En el ejemplo de filtrado, utilizamos múltiples condiciones para seleccionar las filas que cumplen con dichas condiciones. En el ejemplo de reordenamiento, utilizamos `sort_values()` para reordenar el DataFrame según los valores de la columna 'Edad'.

2. Operaciones básicas con datos:
    a. Indexación y slicing de datos:

```
import pandas as pd

# Acceder a una columna específica
columna = df['Nombre']

# Acceder a una fila específica por índice
fila = df.loc['Nombre']

# Acceder a una porción del DataFrame usando slicing
porcion = df.iloc[1:4, 2:4]
```

En este ejemplo, `columna` contendrá la columna 'Nombre' del DataFrame `df`, `fila` contendrá la segunda fila del DataFrame y `porcion` contendrá una porción del DataFrame delimitada por las filas 1 a 3 (excluyendo la cuarta fila) y las columnas 2 y 3.

1. Aplicación de funciones y operaciones a columnas o filas:

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

# Aplicar una operación a filas
df['Suma'] = df.sum(axis=1)
```

En este ejemplo, utilizamos el método `apply()` para aplicar una función lambda a cada valor de la columna 'Edad' y crear una nueva columna llamada 'Edad_Duplicada' que contiene el resultado. También utilizamos la función `sum()` junto con el parámetro `axis=1` para calcular la suma de cada fila y agregarla como una nueva columna llamada 'Suma'.

1. Agrupación de datos y operaciones de agregación:

```
# Agrupar datos por una columna y realizar una operación de agregación
grupo_ciudad = df.groupby('Ciudad')['Puntuación'].mean()

# Aplicar múltiples operaciones de agregación a diferentes columnas
resumen_grupo = df.groupby('Ciudad').agg({'Edad': 'mean', 'Puntuación': 'max'})
```

En el primer ejemplo, agrupamos los datos por la columna 'Ciudad' y calculamos el promedio de la columna 'Puntuación' para cada grupo. En el segundo ejemplo, agrupamos los datos por la columna 'Ciudad' y aplicamos múltiples operaciones de agregación, como calcular la media de 'Edad' y el máximo de 'Puntuación' para cada grupo.

1. Combinación de DataFrames mediante concatenación, unión y fusión:

```
# Concatenación de DataFrames verticalmente
df_concat = pd.concat([df1, df2])
```

```
# Unión de DataFrames basada en una columna común
df_union = pd.merge(df1, df2, on='ID')

# Fusión de DataFrames basada en una columna común
df_fusion = pd.merge(df1, df2, on='ID', how='left')
```

En el ejemplo de concatenación, utilizamos `concat()` para unir verticalmente los DataFrames `df1` y `df2`. En el ejemplo de unión, utilizamos `merge()` para unir los DataFrames `df1` y `df2` basándonos en la columna 'ID'. En el ejemplo de fusión, también utilizamos `merge()`, pero especificamos el parámetro `how='left'` para realizar una fusión izquierda, conservando todas las filas del DataFrame izquierdo `df1` y agregando las columnas correspondientes del DataFrame derecho `df2`.

3. Análisis exploratorio de datos:
    a. Estadísticas descriptivas básicas:

```
import pandas as pd

# Calcular estadísticas descriptivas básicas
promedio = df['Edad'].mean()
mediana = df['Edad'].median()
maximo = df['Puntuación'].max()
minimo = df['Puntuación'].min()
```

En este ejemplo, utilizamos algunas funciones de estadísticas descriptivas básicas. `mean()` calcula el promedio de la columna 'Edad', `median()` calcula la mediana, `max()` obtiene el valor máximo y `min()` obtiene el valor mínimo de la columna 'Puntuación'.

1. Manejo de fechas y tiempo:

```
import pandas as pd

# Convertir una columna en tipo de dato de fecha
df['Fecha'] = pd.to_datetime(df['Fecha'])

# Extraer componentes de fecha y tiempo
```

```
df['Año'] = df['Fecha'].dt.year
df['Mes'] = df['Fecha'].dt.month
```

En este ejemplo, utilizamos `pd.to_datetime()` para convertir una columna en tipo de dato de fecha. Luego, utilizamos `dt.year` y `dt.month` para extraer el año y el mes de la columna 'Fecha' y agregarlos como nuevas columnas 'Año' y 'Mes'.

1. Creación de nuevas columnas derivadas:

```
# Crear una nueva columna basada en cálculos de otras columnas
df['Puntuación_Doble'] = df['Puntuación'] * 2

# Crear una nueva columna basada en condiciones
df['Grupo_Edad'] = df['Edad'].apply(lambda x: 'Joven' if x < 30 else 'Adulto')
```

En este ejemplo, creamos una nueva columna 'Puntuación_Doble' que contiene el doble de los valores de la columna 'Puntuación'. También creamos una nueva columna 'Grupo_Edad' que clasifica a las personas como 'Joven' si su edad es menor a 30, y 'Adulto' en caso contrario.

1. Visualización de datos con pandas y otras bibliotecas complementarias:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Visualización de datos con pandas
df.plot(x='Fecha', y='Puntuación', kind='line')

# Visualización de datos con Matplotlib
plt.scatter(df['Edad'], df['Puntuación'])
plt.xlabel('Edad')
plt.ylabel('Puntuación')
plt.title('Relación entre Edad y Puntuación')

# Visualización de datos con Seaborn
sns.boxplot(x='Grupo_Edad', y='Puntuación', data=df)
```

En este ejemplo, utilizamos diferentes bibliotecas para visualizar los datos. Con pandas, utilizamos el método `plot()` para generar un gráfico de línea de la

columna 'Puntuación' en función de la columna 'Fecha'. Con Matplotlib, utilizamos `scatter()` para crear un gráfico de dispersión entre las columnas 'Edad' y 'Puntuación'. Con Seaborn, utilizamos `boxplot()` para generar un gráfico de caja y bigotes que muestra la distribución de la 'Puntuación' según el 'Grupo_Edad'.

4. Manipulación avanzada de datos:

   a. Uso de índices jerárquicos:

```
import pandas as pd

# Creación de un DataFrame con índices jerárquicos
data = {
    'Grupo': ['A', 'A', 'B', 'B'],
    'Subgrupo': ['X', 'Y', 'X', 'Y'],
    'Valor': [10, 20, 30, 40]
}
df = pd.DataFrame(data)
df.set_index(['Grupo', 'Subgrupo'], inplace=True)

# Acceso a datos con índices jerárquicos
valor = df.loc[('A', 'X'), 'Valor']
```

En este ejemplo, creamos un DataFrame con índices jerárquicos utilizando las columnas 'Grupo' y 'Subgrupo'. Utilizamos `set_index()` para establecer estas columnas como índices. Luego, podemos acceder a los datos utilizando los valores de los índices jerárquicos, como en el caso de `df.loc[('A', 'X'), 'Valor']`, que devuelve el valor correspondiente a la fila con índices 'A' y 'X' en la columna 'Valor'.

1. Tratamiento de datos categóricos:

```
import pandas as pd

# Convertir una columna en tipo de dato categórico
df['Categoría'] = df['Categoría'].astype('category')

# Obtener las categorías únicas y sus frecuencias
categorias = df['Categoría'].unique()
frecuencias = df['Categoría'].value_counts()
```

En este ejemplo, convertimos la columna 'Categoría' en tipo de dato categórico utilizando `astype('category')`. Esto puede ser útil para ahorrar memoria y aplicar operaciones específicas a datos categóricos. Luego, podemos obtener las categorías únicas utilizando el atributo `unique()` y contar la frecuencia de cada categoría utilizando `value_counts()`.

1. Manejo de datos temporales y de series de tiempo:

```
import pandas as pd

# Convertir una columna en tipo de dato de fecha y hora
df['Fecha'] = pd.to_datetime(df['Fecha'])

# Establecer la columna de fecha como índice
df.set_index('Fecha', inplace=True)

# Realizar operaciones de agregación y resampleo por período de tiempo
promedio_diario = df['Valor'].resample('D').mean()
suma_mensual = df['Valor'].resample('M').sum()
```

En este ejemplo, convertimos la columna 'Fecha' en tipo de dato de fecha y hora utilizando `pd.to_datetime()`. Luego, establecemos esta columna como el índice del DataFrame utilizando `set_index()`. Esto facilita el manejo de datos temporales y el cálculo de estadísticas por períodos de tiempo utilizando el método `resample()`, como en los ejemplos de `mean()` para calcular el promedio diario y `sum()` para sumar los valores mensuales.

1. Uso de expresiones lambda y funciones definidas por el usuario:

```
import pandas as pd

# Aplicar una expresión lambda a una columna
df['Duplicado'] = df['Valor'].apply(lambda x: x * 2)

# Definir una función y aplicarla a una columna
def calcular_cuadrado(x):
    return x ** 2

df['Cuadrado'] = df['
```

5. Optimización de rendimiento:
    a. Selección de columnas relevantes: Si estás trabajando con un DataFrame que tiene muchas columnas, puedes seleccionar solo las columnas necesarias para tu análisis. Esto reducirá la cantidad de datos cargados en memoria y mejorará el rendimiento de las operaciones.

```
df = df[['columna1', 'columna2', 'columna3']]  # Selecciona solo las columnas nec
esarias
```

    b. Uso de tipos de datos adecuados: Pandas ofrece una variedad de tipos de datos para representar diferentes tipos de valores. Al elegir el tipo de datos adecuado para cada columna, puedes ahorrar memoria y mejorar el rendimiento de las operaciones. Por ejemplo, si tienes una columna numérica que no requiere decimales, puedes usar el tipo de dato `int` en lugar de `float`.

```
df['columna_numerica'] = df['columna_numerica'].astype(int)  # Cambia el tipo de
 dato a entero
```

    c. Iteración eficiente sobre el DataFrame: Evita iterar sobre filas o columnas de un DataFrame utilizando bucles `for` ya que puede ser muy lento. En su lugar, utiliza las operaciones vectorizadas de pandas para realizar operaciones en todo el DataFrame de manera eficiente.
    d. Uso de funciones de agregación y operaciones vectorizadas: En lugar de aplicar operaciones a cada fila o columna individualmente, aprovecha las funciones de agregación y las operaciones vectorizadas de pandas. Estas operaciones se ejecutan de manera eficiente en el conjunto de datos completo y pueden mejorar significativamente el rendimiento.
    e. Carga de datos en fragmentos (chunking): Si estás trabajando con conjuntos de datos extremadamente grandes que no caben en memoria, puedes considerar la carga de datos en fragmentos más pequeños utilizando el

parámetro `chunksize` en las funciones de lectura de pandas, como `read_csv()`. Esto te permite procesar los datos por partes y reducir la carga en la memoria.

```
chunk_size = 100000  # Tamaño del fragmento
for chunk in pd.read_csv('archivo.csv', chunksize=chunk_size):
    # Procesa cada fragmento de datos
```

f. Uso de métodos y atributos específicos para la eficiencia: Pandas ofrece métodos y atributos específicos que son más eficientes en términos de memoria y rendimiento en comparación con otras operaciones equivalentes. Por ejemplo, utilizar métodos como `fillna()` en lugar de `apply()` para llenar valores faltantes o utilizar atributos como `shape` en lugar de `len()` para obtener la forma del DataFrame.

6. Casos de uso y ejemplos prácticos:
   - Aplicación de pandas en problemas reales de análisis de datos.
   - Ejemplos prácticos de manipulación y análisis de datos utilizando pandas.

```
Crear un DataFrame con datos imaginarios
data = {
'Nombre': ['Juan', 'María', 'Carlos', 'Laura', 'Pedro'],
'Edad': [28, 34, 29, 42, 36],
'Ciudad': ['Buenos Aires', 'Madrid', 'Lima', 'Sao Paulo', 'México DF'],
'Puntuación': [75, 82, 88, 95, 67]
}

df = pd.DataFrame(data)

Mostrar el DataFrame completo
print("DataFrame completo:")
print(df)
print()

Obtener información básica del DataFrame
print("Información básica del DataFrame:")
print( df.info ())
print()

Obtener estadísticas descriptivas del DataFrame
print("Estadísticas descriptivas:")
print(df.describe())
print()

Filtrar los datos para obtener personas mayores de 30 años
mayores_de_30 = df[df['Edad'] > 30]
print("Personas mayores de 30 años:")
```

```
print(mayores_de_30)
print()

Calcular el promedio de la puntuación
promedio_puntuacion = df['Puntuación'].mean()
print("Promedio de la puntuación:", promedio_puntuacion)
```