Pandas

DataFrames

CertiDevs

Índice de contenidos

1. DataFrames . . . 1
2. Importar Pandas . . . 2
3. Creación de DataFrames . . . 3
4. Leer datos de CSV . . . 3
5. Leer de SQL . . . 5
    5.1. Ejemplo 1: Leer todas las filas y columnas de una tabla: . . . 6
    5.2. Ejemplo 2: Leer filas específicas utilizando condiciones: . . . 6
    5.3. Ejemplo 3: Leer columnas específicas de una tabla: . . . 6
    5.4. Ejemplo 4: Ordenar los resultados de la consulta: . . . 6
    5.5. Ejemplo 5: Leer datos utilizando una consulta SQL que involucre varias tablas: . . . 6
6. Acceder a columnas, filas y elementos específicos en un DataFrame . . . 7
7. Seleccionar y filtrar datos en un DataFrame . . . 7
8. Manipulación de datos en DataFrames . . . 8
9. Funciones estadísticas y de agregación en DataFrames . . . 9
10. Fusionar, unir y concatenar DataFrames . . . 9
11. Agrupar datos en DataFrames . . . 10
12. Aplicar funciones personalizadas a DataFrames . . . 11
13. Pivotar y remodelar DataFrames . . . 11
14. Guardar DataFrames en archivos . . . 12

1. DataFrames

Un **DataFrame** en Pandas es una **estructura de datos bidimensional** etiquetada similar a una **tabla** de una hoja de cálculo o una base de datos.

Puede contener **múltiples columnas**, cada una de las cuales puede tener un tipo de datos diferente.

Al igual que con las Series, un DataFrame **tiene índices** tanto para filas como para columnas.

Las operaciones en DataFrames incluyen una amplia gama de acciones y transformaciones que se pueden realizar en los datos.

Pandas proporciona una gran cantidad de funciones y métodos para realizar operaciones, como fusionar, unir y concatenar DataFrames, agrupar datos y aplicar funciones personalizadas.

- **Creación de DataFrames**:
  - Desde diccionarios
  - Desde listas
  - Desde arreglos de Numpy
  - Desde archivos CSV, Excel, JSON, SQL, etc.
- **Selección y filtrado**:
  - Seleccionar columnas
  - Seleccionar filas por índice
  - Filtrar filas por condiciones
  - Filtrar usando query()
  - Seleccionar filas y columnas usando iloc[] y loc[]
- **Manipulación de columnas**:
  - Renombrar columnas
  - Cambiar el orden de las columnas
  - Añadir nuevas columnas
  - Eliminar columnas
  - Cambiar el tipo de datos de las columnas
- **Manipulación de filas**:
  - Añadir nuevas filas
  - Eliminar filas
  - Cambiar el orden de las filas
  - Ordenar filas por columnas
  - Reindexar filas
- **Manejo de datos faltantes (NaN)**:

    - Detectar datos faltantes
    - Rellenar datos faltantes
    - Eliminar filas/columnas con datos faltantes
- **Agrupación y agregación**:
    - Agrupar datos usando groupby()
    - Aplicar funciones de agregación (suma, media, conteo, etc.)
    - Aplicar funciones personalizadas a grupos
- **Transformación y aplicación de funciones**:
    - Aplicar funciones a filas/columnas usando apply()
    - Aplicar funciones a elementos individuales usando applymap() y map()
    - Transformar datos usando transform()
- **Combinar DataFrames**:
    - Unir DataFrames usando merge()
    - Concatenar DataFrames usando concat()
    - Añadir filas de un DataFrame a otro usando append()
- **Pivotar y remodelar DataFrames**:
    - Crear tablas pivote
    - Desapilar y apilar DataFrames
    - Cambiar entre formatos largo y ancho
- **Estadísticas y análisis**:
    - Calcular estadísticas descriptivas (media, mediana, desviación estándar, etc.)
    - Calcular correlaciones
    - Calcular covarianzas
- **Visualización**:
    - Crear gráficos básicos usando la integración de Pandas con Matplotlib
    - Crear gráficos más avanzados utilizando bibliotecas como Seaborn, Plotly, etc.

2. Importar Pandas

Para importar Pandas, utilice el siguiente comando:

```
import pandas as pd
```

3. Creación de DataFrames

Puede **crear un DataFrame** en Pandas utilizando el constructor `pd.DataFrame()`.

A continuación, se muestran diferentes formas de crear DataFrames a partir de diccionarios, listas, arrays de NumPy y archivos CSV:

```
# Crear un DataFrame a partir de un diccionario
my_dict = {'A': [10, 20, 30], 'B': [40, 50, 60], 'C': [70, 80, 90]}
df_from_dict = pd.DataFrame(my_dict)
print("DataFrame a partir de un diccionario:")
print(df_from_dict)

# Crear un DataFrame a partir de una lista de listas
my_list = [[10, 40, 70], [20, 50, 80], [30, 60, 90]]
df_from_list = pd.DataFrame(my_list, columns=['A', 'B', 'C'])
print("\nDataFrame a partir de una lista de listas:")
print(df_from_list)

# Crear un DataFrame a partir de un array de NumPy
my_array = np.array([[10, 40, 70], [20, 50, 80], [30, 60, 90]])
df_from_array = pd.DataFrame(my_array, columns=['A', 'B', 'C'])
print("\nDataFrame a partir de un array de NumPy:")
print(df_from_array)

# Crear un DataFrame a partir de un archivo CSV
# Asegúrese de tener un archivo CSV válido en la ruta especificada
# df_from_csv = pd.read_csv('my_data.csv')
# print("\nDataFrame a partir de un archivo CSV:")
# print(df_from_csv)
```

4. Leer datos de CSV

Leer un archivo CSV básico:

```
import pandas as pd

# Leer el archivo CSV
data = pd.read_csv('archivo.csv')

# Mostrar los primeros registros
print(data.head())
```

Cambiar el delimitador de campos:

Por defecto, Pandas utiliza la coma (`,`) como delimitador. Para cambiarlo, usa el parámetro `sep` o `delimiter`.

```
data = pd.read_csv('archivo.tsv', sep='\t')
```

Especificar la codificación del archivo:

Utiliza el **parámetro** `encoding` para indicar la codificación del archivo, por ejemplo, `utf-8` o `ISO-8859-1`.

```
data = pd.read_csv('archivo.csv', encoding='ISO-8859-1')
```

Leer un archivo CSV sin cabecera:

Si el archivo CSV no tiene una fila de encabezado, usa el parámetro header=None.

```
data = pd.read_csv('archivo_sin_cabecera.csv', header=None)
```

Asignar nombres de columnas personalizados:

Utiliza el parámetro names para proporcionar una lista de nombres de columnas.

```
columnas = ['nombre', 'edad', 'ciudad']
data = pd.read_csv('archivo_sin_cabecera.csv', header=None, names=columnas)
```

Saltar filas al inicio del archivo:

Usa el parámetro `skiprows` para saltar un número específico de filas al principio del archivo.

```
data = pd.read_csv('archivo.csv', skiprows=2)
```

Leer solo un número específico de filas:

Utiliza el parámetro `nrows` para leer solo un número específico de filas.

```
data = pd.read_csv('archivo.csv', nrows=10)
```

Seleccionar columnas específicas a cargar:

Usa el parámetro `usecols` para cargar solo columnas específicas.

```
data = pd.read_csv('archivo.csv', usecols=['nombre', 'edad'])
```

Convertir **valores nulos** a `NaN`:

Pandas convierte automáticamente los valores vacíos en `NaN`. Si deseas convertir otros valores

específicos en `NaN`, utiliza el parámetro `na_values`.

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

Leer un archivo CSV con una columna como índice:

Utiliza el parámetro `index_col` para especificar una columna como índice.

```
data = pd.read_csv('archivo.csv', index_col='nombre')
```

5. Leer de SQL

Para leer datos desde una base de datos relaciones como por ejemplo MySQL en Pandas, primero necesitas instalar un driver de MySQL para Python como mysql-connector-python. Puedes instalarlo usando pip:

```
pip install mysql-connector-python
```

Una vez que hayas instalado el controlador, sigue estos pasos para leer datos desde MySQL en un DataFrame de Pandas:

Importa las bibliotecas necesarias:

```
import mysql.connector
import pandas as pd
```

Establece una conexión con la base de datos MySQL:

```
cnx = mysql.connector.connect(
    host="localhost",
    user="your_username",
    password="your_password",
    database="your_database"
)
```

Utiliza el método `read_sql()` para ejecutar una consulta SQL y almacenar los resultados en un DataFrame de Pandas:

```
query = "SELECT * FROM your_table;"
df = pd.read_sql(query, cnx)
```

Cierra la conexión a la base de datos:

```
cnx.close()
```

Aquí tienes algunos ejemplos de consultas SQL y cómo leer los datos en DataFrames de Pandas:

5.1. Ejemplo 1: Leer todas las filas y columnas de una tabla:

```
query = "SELECT * FROM your_table;"
df = pd.read_sql(query, cnx)
```

5.2. Ejemplo 2: Leer filas específicas utilizando condiciones:

```
query = "SELECT * FROM your_table WHERE column_name = 'some_value';"
df = pd.read_sql(query, cnx)
```

5.3. Ejemplo 3: Leer columnas específicas de una tabla:

```
query = "SELECT column1, column2, column3 FROM your_table;"
df = pd.read_sql(query, cnx)
```

5.4. Ejemplo 4: Ordenar los resultados de la consulta:

```
query = "SELECT * FROM your_table ORDER BY column_name ASC;"
df = pd.read_sql(query, cnx)
```

5.5. Ejemplo 5: Leer datos utilizando una consulta SQL que involucre varias tablas:

```
query = """
    SELECT t1.column1, t2.column2
    FROM table1 AS t1
    JOIN table2 AS t2 ON t1.id = t2.id
    WHERE t1.column1 > 100;
"""
df = pd.read_sql(query, cnx)
```

Estos ejemplos muestran cómo ejecutar diferentes consultas SQL y leer los datos en DataFrames de

Pandas.

6. Acceder a columnas, filas y elementos específicos en un DataFrame

Puede acceder a columnas, filas y elementos específicos en un DataFrame utilizando diferentes métodos y funciones, como la notación de corchetes, `loc`, `iloc`, `at` y `iat`:

```
# Acceder a una columna por su nombre
print("\nColumna 'A':")
print(df_from_dict['A'])

# Acceder a una fila por su índice
print("\nFila 0:")
print(df_from_dict.loc[0])

# Acceder a un elemento específico por su fila y columna (etiquetas)
print("\nElemento en la fila 0 y columna 'A':")
print(df_from_dict.at[0, 'A'])

# Acceder a un elemento específico por su posición (índices numéricos)
print("\nElemento en la posición (0, 0):")
print(df_from_dict.iat[0, 0])
```

7. Seleccionar y filtrar datos en un DataFrame

Puede seleccionar y filtrar datos en un DataFrame utilizando operadores de comparación, máscaras booleanas y funciones como `query()`:

```
# Filtrar filas según una condición
print("\nFilas donde la columna 'A' es mayor que 10:")
print(df_from_dict[df_from_dict['A'] > 10])
```

Utilizar **máscaras booleanas** para filtrar datos

```
mask = (df_from_dict['A'] > 10) & (df_from_dict['B'] < 60)
print("\nFilas donde la columna 'A' es mayor que 10 y la columna 'B' es menor que
60:")
print(df_from_dict[mask])
```

Utilizar la función `query()` para filtrar datos

```
print("\nFilas donde la columna 'A' es mayor que 10 y la columna 'C' es menor que
90:")
print(df_from_dict.query("A > 10 and C < 90"))
```

8. Manipulación de datos en DataFrames

Pandas proporciona varias funciones y métodos para **manipular datos** en DataFrames, como **agregar**, **eliminar** y **modificar** columnas, **cambiar** el índice y **ordenar** datos:

Agregar una nueva columna al DataFrame:

```
df_from_dict['D'] = [100, 110, 120]
print("\nDataFrame con nueva columna 'D':")
print(df_from_dict)
```

Eliminar una columna del DataFrame

```
df_without_column = df_from_dict.drop('D', axis=1)
print("\nDataFrame sin la columna 'D':")
print(df_without_column)
```

Modificar los valores de una columna

```
df_from_dict['A'] = df_from_dict['A'] * 2
print("\nDataFrame con la columna 'A' modificada:")
print(df_from_dict)
```

Cambiar el índice del DataFrame

```
df_with_new_index = df_from_dict.set_index('A')
print("\nDataFrame con un nuevo índice:")
print(df_with_new_index)
```

Ordenar el DataFrame por una columna

```
sorted_df = df_from_dict.sort_values(by='B')
print("\nDataFrame ordenado por la columna 'B':")
print(sorted_df)
```

9. Funciones estadísticas y de agregación en DataFrames

Puede aplicar funciones estadísticas y de agregación a un DataFrame utilizando métodos como `mean()`, `median()`, `min()`, `max()`, `sum()` y `describe()`:

Calcular la **media** de cada columna

```
print("\nMedia de cada columna:")
print(df_from_dict.mean())
```

Calcular la **suma** de cada fila

```
print("\nSuma de cada fila:")
print(df_from_dict.sum(axis=1))
```

Describir **estadísticas básicas** del DataFrame

```
print("\nEstadísticas básicas del DataFrame:")
print(df_from_dict.describe())
```

10. Fusionar, unir y concatenar DataFrames

Pandas ofrece varias funciones para **combinar DataFrames**, como `merge()`, `join()` y `concat()`:

Crear DataFrames de ejemplo:

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
'A': ['A0', 'A1', 'A2', 'A3'],
'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
'C': ['C0', 'C1', 'C2', 'C3'],
'D': ['D0', 'D1', 'D2', 'D3']})
```

Fusionar DataFrames utilizando una columna clave:

```
merged = pd.merge(left, right, on='key')
print("\nDataFrames fusionados:")
print(merged)
```

Unir DataFrames utilizando índices:

```
joined = left.join(right, lsuffix='_left', rsuffix='_right')
print("\nDataFrames unidos:")
print(joined)
```

Concatenar DataFrames

```
concatenated = pd.concat([left, right], axis=1)
print("\nDataFrames concatenados:")
print(concatenated)
```

11. Agrupar datos en DataFrames

Puede **agrupar datos** en un DataFrame utilizando el método `groupby()`, que permite realizar cálculos y transformaciones en grupos de datos:

Crear un DataFrame de ejemplo:

```
data = {'Company': ['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],
'Person': ['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],
'Sales': [200, 120, 340, 124, 243, 350]}

df = pd.DataFrame(data)
```

Agrupar datos por la columna 'Company'

```
grouped = df.groupby('Company')
```

Calcular la suma de ventas por compañía

```
print("\nSuma de ventas por compañía:")
print(grouped['Sales'].sum())
```

Calcular la media de ventas por compañía

```
print("\nMedia de ventas por compañía:")
print(grouped['Sales'].mean())
```

Obtener estadísticas básicas por compañía

```
print("\nEstadísticas básicas por compañía:")
print(grouped.describe())
```

12. Aplicar funciones personalizadas a DataFrames

Puede aplicar funciones personalizadas a un DataFrame utilizando el método `apply()`:

Crear una función personalizada para calcular el cuadrado de un número

```
def square(x):
    return x ** 2

# Aplicar la función 'square' a la columna 'Sales' del DataFrame

df['Squared Sales'] = df['Sales'].apply(square)
print("\nDataFrame con la columna 'Squared Sales':")
print(df)
```

También puede utilizar funciones lambda

```
df['Cubed Sales'] = df['Sales'].apply(lambda x: x ** 3)
print("\nDataFrame con la columna 'Cubed Sales':")
print(df)
```

13. Pivotar y remodelar DataFrames

Puede pivotar y remodelar un DataFrame utilizando funciones como `pivot()` y `melt()`:

Crear un DataFrame de ejemplo

```
data = {'Date': ['2021-01-01', '2021-01-01', '2021-01-01', '2021-01-02', '2021-01-02',
'2021-01-02'],
'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Los Angeles', 'Chicago'],
'Temperature': [32, 75, 30, 31, 77, 28],
'Humidity': [80, 10, 85, 82, 12, 90]}

df = pd.DataFrame(data)
```

Pivotar el DataFrame

```
pivoted = df.pivot(index='Date', columns='City')
print("\nDataFrame pivotado:")
print(pivoted)
```

Remodelar el DataFrame utilizando melt()

```
melted = pd.melt(df, id_vars=['Date', 'City'], var_name='Variable', value_name=
'Value')
print("\nDataFrame remodelado:")
print(melted)
```

14. Guardar DataFrames en archivos

Puede guardar un DataFrame en un archivo utilizando métodos como `to_csv()`, `to_excel()` y `to_json()`.

Pandas ofrece una función llamada to_csv para guardar fácilmente un DataFrame en un archivo CSV. Aquí hay algunas opciones comunes y sus descripciones:

Guardar un DataFrame básico en un archivo CSV:

```
import pandas as pd

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

# Guardar el DataFrame en un archivo CSV
df.to_csv('archivo.csv', index=False)
```

Cambiar el delimitador de campos:

Por defecto, Pandas utiliza la coma (`,`) como delimitador. Para cambiarlo, usa el parámetro `sep`.

```
df.to_csv('archivo.tsv', sep='\t', index=False)
```

Especificar la codificación del archivo:

Utiliza el parámetro encoding para indicar la codificación del archivo, por ejemplo, 'utf-8' o 'ISO-8859-1'.

```
df.to_csv('archivo.csv', encoding='ISO-8859-1', index=False)
```

No escribir los nombres de las columnas en el archivo:

Si no deseas que los nombres de las columnas se escriban en el archivo, utiliza el parámetro `header=False`.

```
df.to_csv('archivo_sin_cabecera.csv', header=False, index=False)
```

No escribir el índice en el archivo:

Por defecto, Pandas escribe el índice en el archivo CSV. Si no deseas que se escriba el índice, utiliza el parámetro `index=False`.

```
df.to_csv('archivo_sin_indice.csv', index=False)
```

Especificar el formato de los números de coma flotante:

Usa el parámetro float_format para especificar el formato de los números de coma flotante.

```
df.to_csv('archivo.csv', float_format='%.2f', index=False)
```

Especificar las columnas a escribir en el archivo:

Utiliza el parámetro columns para seleccionar las columnas que deseas escribir en el archivo.

```
df.to_csv('archivo_columnas_seleccionadas.csv', columns=['A'], index=False)
```

Comprimir el archivo de salida:

Usa el parámetro compression para especificar el tipo de compresión a utilizar al guardar el archivo. Pandas admite 'gzip', 'bz2', 'zip' y 'xz'.

```
df.to_csv('archivo.csv.gz', compression='gzip', index=False)
```

Escribir en un archivo con un modo específico:

Por defecto, Pandas sobrescribe el archivo de salida. Utiliza el parámetro mode para especificar el modo en el que deseas escribir en el archivo (por ejemplo, 'a' para agregar datos al final del archivo).

```
df.to_csv('archivo.csv', mode='a', header=False, index=False)
```

Estas son algunas opciones que puedes utilizar con la función `to_csv` de Pandas para guardar un DataFrame en un archivo CSV.