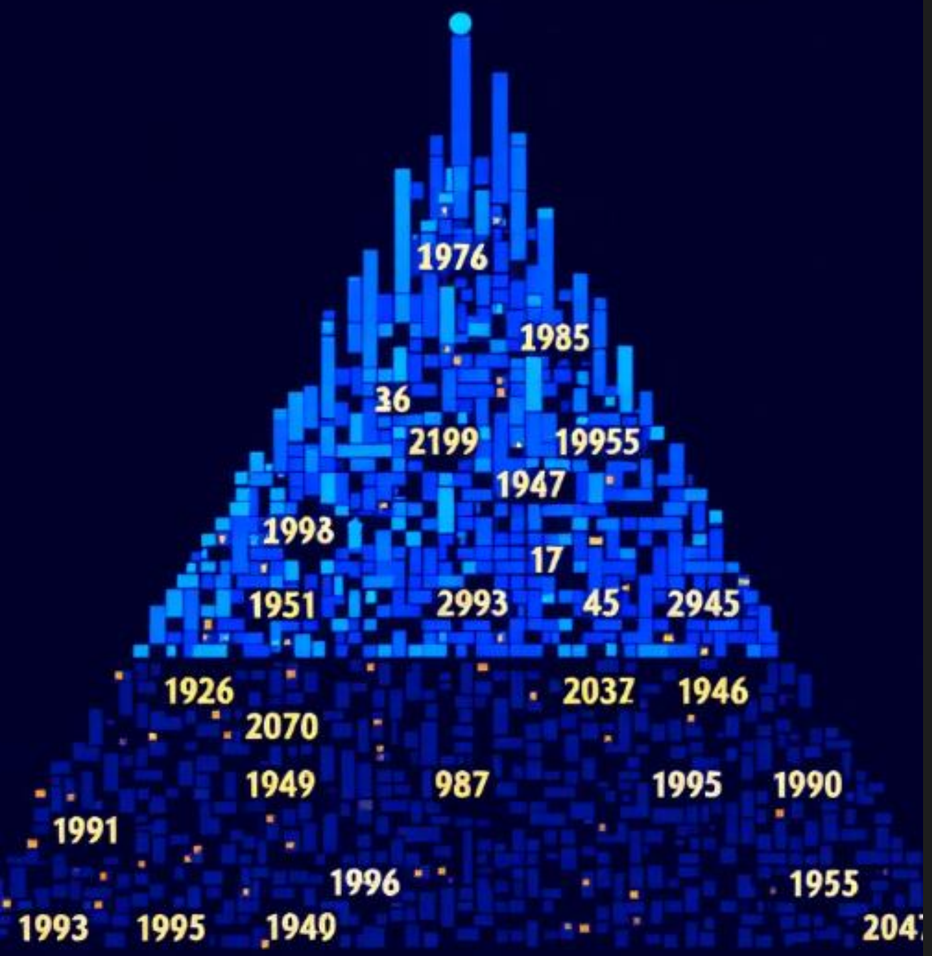
Obtención y Preparación

de Datos

Sesión 5



Indexación Jerárquica: Organizando Datos Multinivel

Concepto Fundamental

La indexación jerárquica (MultiIndex) permite tener múltiples niveles de índices en un DataFrame, facilitando la organización y manipulación de datos con estructura multinivel o multidimensional.

Creación de MultiIndex

Se implementa convirtiendo columnas específicas en niveles de índice mediante el método .set_index(['Columna1', 'Columna2']). Esto transforma las columnas seleccionadas en un índice de múltiples niveles.

Ventajas Prácticas

Facilita el acceso a subconjuntos específicos de datos y permite realizar operaciones complejas de manera más intuitiva, especialmente cuando trabajamos con datos que tienen relaciones jerárquicas naturales.

Ejemplo Práctico de Indexación Jerárquica

✅ DataFrame Original

Comenzamos con un DataFrame que contiene información de ventas por ciudad y año. Inicialmente, "Ciudad", "Año" y "Ventas" son columnas estándar, donde cada fila representa un registro único de ventas.

```python
import pandas as pd

# Crear un DataFrame con indexación jerárquica
data = {
    'Ciudad': ['Madrid', 'Madrid', 'Barcelona', 'Barcelona'],
    'Año': [2020, 2021, 2020, 2021],
    'Ventas': [1000, 1500, 1200, 1300]
}

df = pd.DataFrame(data)
```

Ejemplo Práctico de Indexación Jerárquica

✅ Transformación

Al aplicar .set_index(), convertimos "Ciudad" y "Año" en un índice jerárquico. "Ciudad" se convierte en el nivel superior y "Año" en el segundo nivel del índice.

```python
import pandas as pd

# Crear un DataFrame con indexación jerárquica
data = {
    'Ciudad': ['Madrid', 'Madrid', 'Barcelona', 'Barcelona'],
    'Año': [2020, 2021, 2020, 2021],
    'Ventas': [1000, 1500, 1200, 1300]
}

df = pd.DataFrame(data)

# Establecer el índice jerárquico
df.set_index(['Ciudad', 'Año'], inplace=True)

print(df)
```

Ejemplo Práctico de Indexación Jerárquica

✅ Acceso a Datos

Con la estructura jerárquica (MultiIndex), poemos acceder fácilmente a datos específicos. Por ejemplo, para acceder a las ventas de Barcelona en 2021:

```
# Acceder a los datos de Barcelona en 2021
print(df.loc[('Barcelona', 2021), :])
```

Otros métodos útiles con MultiIndex:

🔄 **swaplevel()**
Permite intercambiar los niveles del índice:

Otros Métodos Útiles MúltiIndex

🔄 **swaplevel()**

Permite intercambiar los niveles del índice:

```
# Intercambiar el orden de los niveles del índice
df_swapped = df.swaplevel()
```

📦 stack()

Convierte columnas en niveles del índice (útil para transformar datos):

```
# Apilar columnas en un nuevo nivel del índice
df_stacked = df.stack()
```

📤 **unstack()**

Convierte un nivel del índice en columnas (inverso de stack):

```
# Desapilar el último nivel del índice y convertirlo en columnas
df_unstacked = df.unstack()
```

Agrupamiento con GroupBy: Dividir-Aplicar-Combinar

1

Dividir

La función .groupby() divide los datos en grupos según los valores de una o más columnas. Cada grupo contiene todas las filas que comparten el mismo valor en la columna de agrupación.

2

Aplicar

Una vez divididos los datos, se aplican funciones de agregación como sum(), mean(), count(), max() o min() a cada grupo por separado, permitiendo análisis específicos por categoría.

3

Combinar

Finalmente, los resultados de cada grupo se combinan en un nuevo DataFrame, proporcionando una vista resumida de los datos originales según los criterios de agrupación establecidos.

Ejemplos de Agrupamiento y Agregación

Agrupamiento Simple

Al usar: df.groupby('Ciudad')['Ventas'].sum()

agrupamos los datos por la columna "Ciudad" y calculamos la suma de ventas para cada ciudad, obteniendo el total de ventas por localidad.

```
# Agrupar por 'Ciudad' y calcular la suma de 'Ventas'
df_grouped = df.groupby('Ciudad')['Ventas'].sum()

print(df_grouped)
```

Múltiples Funciones

Con: df.groupby('Ciudad')['Ventas'].agg(['sum', 'mean', 'count'])

aplicamos varias funciones de agregación simultáneamente, obteniendo suma, promedio y conteo de ventas por ciudad en un solo resultado.

```
# Agrupar por 'Ciudad' y calcular varias agregaciones
df_grouped = df.groupby('Ciudad')['Ventas'].agg([sum, mean, max, min])

print(df_grouped)
```

Pivoteo de DataFrames

1

Concepto de Pivoteo

El pivoteo transforma datos de formato largo (muchas filas) a formato ancho (más columnas), reorganizando los valores para facilitar ciertos tipos de análisis y visualizaciones.

2

Método pivot_table

La función pivot_table() permite especificar qué columnas usar como índices de filas, columnas y valores.

3

Resultado

El resultado es una tabla donde cada combinación única de índice y columna contiene el valor agregado correspondiente, creando una vista matricial de los datos que facilita comparaciones entre categorías.

```python
# Crear un DataFrame de ejemplo
data = {
    'Ciudad': ['Madrid', 'Madrid', 'Barcelona', 'Barcelona'],
    'Año': [2020, 2021, 2020, 2021],
    'Ventas': [1000, 1500, 1200, 1300]
}

df = pd.DataFrame(data)

# Pivoteo con pivot_table
df_pivot = df.pivot_table(values='Ventas', index='Ciudad', columns='Año', aggfunc='sum')

print(df_pivot)
```

Despivoteo con Melt: De Formato Ancho a Largo

1

Concepto de Despivoteo

El despivoteo es la operación inversa al pivoteo, transformando datos de formato ancho (muchas columnas) a formato largo (más filas), lo que es útil para análisis estadísticos y visualizaciones específicas.

2

Método melt

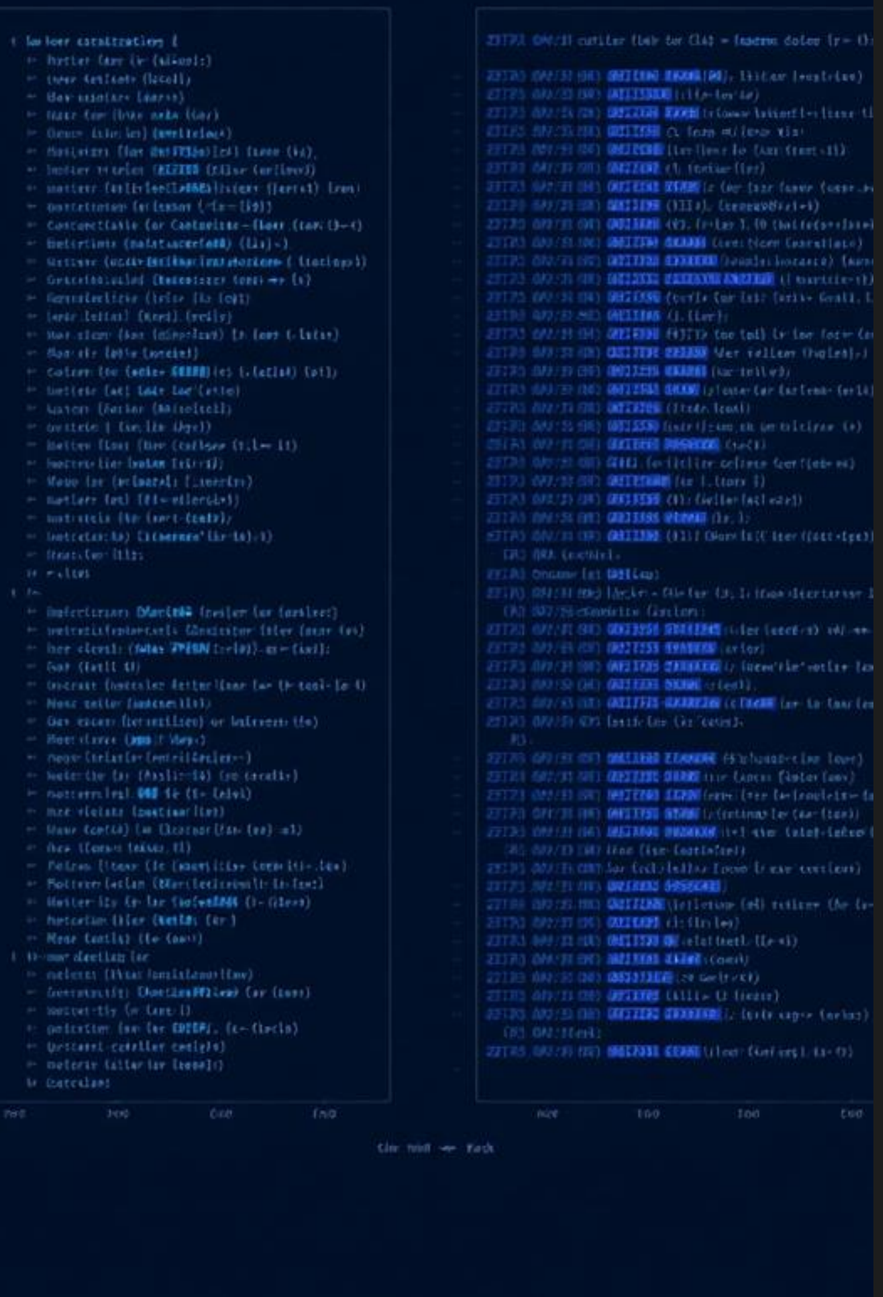
La función melt() convierte columnas en filas, especificando qué columnas mantener como identificadores y cuáles "derretir" en pares de variable-valor.

```python
# Crear un DataFrame de ejemplo
data = {
    'Ciudad': ['Madrid', 'Barcelona'],
    '2020': [1000, 1200],
    '2021': [1500, 1300]
}

df = pd.DataFrame(data)

# Despivoteo con melt
df_melted = pd.melt(df, id_vars=['Ciudad'], value_vars=['2020', '2021'], var_name='Año', value_name='Ventas')

print(df_melted)
```

3

Resultado

El resultado es un DataFrame en formato largo, con tres columnas: identificadores, *variable* y *valor*. Cada fila representa una observación única



Concatenación de DataFrames

Vertical

La función pd.concat([df1, df2], axis=0) apila DataFrames verticalmente, agregando las filas del segundo DataFrame después del primero. Es útil cuando tenemos datos de la misma estructura que queremos combinar secuencialmente.

Concatenación Horizontal

Con pd.concat([df1, df2], axis=1) unimos DataFrames horizontalmente, añadiendo las columnas del segundo DataFrame junto a las del primero. No requiere que ambos DataFrames tengan el mismo número de filas, ya que se alinean por índice. Si un índice no existe, rellenará con NaN.

Gestión de Índices

Los índices originales se conservan por defecto. Para restablecerlos y crear una secuencia continua, podemos usar reset_index(drop=True) después de la concatenación.

Ejemplos Concatenación de DataFrames

✅ Concatenación vertical (agregar filas):

```python
# Crear dos DataFrames
df1 = pd.DataFrame({'ID': [1, 2], 'Ventas': [100, 200]})
df2 = pd.DataFrame({'ID': [3, 4], 'Ventas': [300, 400]})

# Concatenar los DataFrames
df_concat = pd.concat([df1, df2], axis=0)

print(df_concat)
```

✅ Concatenación horizontal (unir columnas):

```python
df3 = pd.DataFrame({'ID': [1, 2, 3], 'Descuento': [10, 15, 20]})
df_hconcat = pd.concat([df1, df3], axis=1)
print(df_hconcat)
```

Merge de DataFrames: Uniones Tipo SQL

La función pd.merge() permite combinar DataFrames basándose en una clave común, similar a una unión (JOIN) en SQL.

Tipos de Merge disponibles:

1. inner (intersección) → Solo valores comunes.
2. left (izquierda) → Conserva todos los valores del primer DataFrame y agrega los del segundo si hay coincidencia.
3. right (derecha) → Conserva todos los valores del segundo DataFrame.
4. outer (unión) → Conserva todos los valores de ambos DataFrames, rellenando con NaN si no hay coincidencia.

- *Nota: Si no se declara el parámetro how en pd.merge(), Pandas por defecto usa how='inner'.*

Merge con múltiples claves y sufijos

✅ **¿Por qué usar múltiples claves?**

Cuando queremos combinar dos DataFrames donde la relación depende de más de una columna (por ejemplo, id y fecha), usamos on=['col1', 'col2'].

💡 **Evitar conflicto de nombres**

Si ambos DataFrames tienen columnas con el mismo nombre, podemos usar el parámetro suffixes para diferenciarlas.

📌 **Ejemplo:**

```python
import pandas as pd

# DataFrame 1
df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'fecha': ['2021-01', '2021-02', '2021-03'],
    'ventas': [100, 150, 200]
})

# DataFrame 2
df2 = pd.DataFrame({
    'id': [1, 2, 3],
    'fecha': ['2021-01', '2021-02', '2021-03'],
    'costos': [50, 70, 90]
})

# Merge con múltiples claves y sufijos
df_merged = pd.merge(df1, df2, on=['id', 'fecha'], suffixes=('_ventas', '_costos'))
print(df_merged)
```

Criterios para Elegir Entre Concat y Merge

Método	Cuándo usarlo	Descripción
pd.concat()	Para unir filas o columnas sin depender de claves comunes.	Une DataFrames por filas (axis=0) o columnas (axis=1).
pd.merge()	Para combinar datos con una clave común, como en SQL.	Combina DataFrames según una clave (inner, left, right, outer).

- La elección entre concat() y merge() depende fundamentalmente de la estructura de tus datos y el tipo de combinación que necesites realizar:

- Si los DataFrames tienen la misma estructura y solo necesitas unirlos secuencialmente, concat() es la opción más directa. Si necesitas combinar información relacionada basada en valores comunes entre diferentes conjuntos de datos, merge() ofrece la flexibilidad de las uniones tipo SQL.

Preguntas

Sección de preguntas

Obtención y Preparación

de Datos

Continúe con las actividades