In [2]:
```
import numpy as np
import pandas as pd

pd.options.display.max_rows = 20
# Configura pandas para mostrar un
# máximo de 20 filas al imprimir DataFrames.
# Esto ayuda a mantener la salida manejable en la consola.

pd.options.display.max_colwidth = 80
# Configura pandas para que la anchura máxima
# de las columnas sea de 80 caracteres al
# mostrar datos. Esto es útil para asegurarse
# de que las columnas no se trunquen demasiado en la visualización.

pd.options.display.max_columns = 20
# Configura pandas para mostrar un máximo
# de 20 columnas al imprimir DataFrames.
# Similar a max_rows, esto ayuda a
# controlar la cantidad de datos mostrados.

np.random.seed(12345)


import matplotlib.pyplot as plt

plt.rc("figure", figsize=(10, 6)) # Configura matplotlib para que todas
# las figuras tengan un tamaño predeterminado de 10 pulgadas de ancho
# por 6 pulgadas de alto. rc es una función de matplotlib que se
# utiliza para configurar parámetros globales.

np.set_printoptions(precision=4, suppress=True)
# Configura numpy para mostrar los números con una
# precisión de 4 decimales y suprimir el uso de
# notación científica (suppress=True) en la salida impresa.
```

3. Tratamiento de datos (Wrangling): Unir (Join) ,combinar (Combine) y remodelar (Reshape)

En muchas aplicaciones, los datos pueden estar dispersos en varios archivos o bases de datos, o estar organizados de una forma que no es conveniente analizar.

En primer lugar, se introduce el concepto de indexación jerárquica `(Hierarchical Indexing)` en pandas, que se utiliza ampliamente en algunas de estas operaciones.

3.1 Indexación jerárquica

La indexación jerárquica es una característica importante de pandas que le permite tener múltiples (dos o más) niveles de índice en un eje. Otra forma de pensar en ello es que proporciona una manera para que usted pueda trabajar con datos de mayor dimensión

en una forma de menor dimensión. Empecemos con un ejemplo sencillo: crear una Serie con una lista de listas (o arrays) como índice:

In [3]:
```
data = pd.Series(np.random.uniform(size=9),
        index=[["a", "a", "a", "b", "b", "c", "c", "d", "d"],
        [1, 2, 3, 1, 3, 1, 2, 2, 3]])
data
```

Out[3]:
```
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64
```

Lo que está viendo es una vista de una Serie con un Multiíndice (`MultiIndex`) como índice. Los "huecos" en la visualización del índice significan "use la etiqueta directamente arriba":

In [4]:
```
data.index
```

Out[4]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

Con un objeto indexado jerárquicamente, es posible la llamada indexación parcial, que permite seleccionar de forma concisa subconjuntos de los datos:

In [5]:
```
data["b"]
```

Out[5]:
```
1    0.204560
3    0.567725
dtype: float64
```

In [6]:
```
data["b":"c"]
```

Out[6]:
```
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
dtype: float64
```

In [7]:
```
data.loc[["b", "c"]]
```

Out[7]:
```
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
dtype: float64
```

La selección es posible incluso desde un nivel "interior". Aquí seleccionoamos todos los valores que tienen el valor 2 del segundo nivel de índice:

In [8]:
```
data.loc[:, 2]
```

Out[8]:
```
a    0.316376
c    0.964515
d    0.653177
dtype: float64
```

Ejemplo 3.1

In [9]:
```
data_1 = {
    'Store': ['Store1', 'Store1', 'Store1', 'Store2', 'Store2', 'Store2'],
    'Product': ['A', 'A', 'B', 'A', 'B', 'B'],
    'Date': ['2024-07-01', '2024-07-02', '2024-07-01', '2024-07-01', '2024-07-02
    'Sales': [100, 150, 200, 300, 400, 500]
}
df = pd.DataFrame(data_1)

df['Date'] = pd.to_datetime(df['Date'])  # Convertir la columna 'Date' a tipo da
df
```

Out[9]:

	Store	Product	Date	Sales
0	Store1	A	2024-07-01	100
1	Store1	A	2024-07-02	150
2	Store1	B	2024-07-01	200
3	Store2	A	2024-07-01	300
4	Store2	B	2024-07-02	400
5	Store2	B	2024-07-03	500

Configurar a Multiindex

In [10]:
```
df.set_index(['Store', 'Product', 'Date'], inplace=True)
df
```

Out[10]:

			Sales
Store	**Product**	**Date**	
Store1	**A**	**2024-07-01**	100
		2024-07-02	150
	B	**2024-07-01**	200
Store2	**A**	**2024-07-01**	300
	B	**2024-07-02**	400
		2024-07-03	500

La indexación jerárquica desempeña un papel importante en la reorganización de los datos y en las operaciones basadas en grupos, como la formación de una tabla dinámica. Por ejemplo, puede reorganizar estos datos en un DataFrame utilizando su método `unstack` ('desapilar'):

Volvamos al dataframe data

In [11]:
```
data
```

Out[11]:
```
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64
```

In [12]:
```
data.unstack()
```

Out[12]:

	1	2	3
a	0.929616	0.316376	0.183919
b	0.204560	NaN	0.567725
c	0.595545	0.964515	NaN
d	NaN	0.653177	0.748907

La operación inversa de desapilar (`unstack`) es apilar (`stack`):

In [13]:
```
data.unstack().stack()
```

Out[13]:
```
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64
```

Vamos a desapilar el dataframe del ejemplo 3.1

In [14]:
```
df
```

Out[14]:

			Sales
Store	**Product**	**Date**	
Store1	**A**	**2024-07-01**	100
		2024-07-02	150
	B	**2024-07-01**	200
Store2	**A**	**2024-07-01**	300
	B	**2024-07-02**	400
		2024-07-03	500

In [15]:
```
df.unstack()
```

Out[15]:

		Sales		
	Date	**2024-07-01**	**2024-07-02**	**2024-07-03**
Store	**Product**			
Store1	**A**	100.0	150.0	NaN
	B	200.0	NaN	NaN
Store2	**A**	300.0	NaN	NaN
	B	NaN	400.0	500.0

In [16]:
```
df.unstack('Store')
```

Out[16]:

		Sales	
	Store	Store1	Store2
Product	Date		
A	2024-07-01	100.0	300.0
	2024-07-02	150.0	NaN
B	2024-07-01	200.0	NaN
	2024-07-02	NaN	400.0
	2024-07-03	NaN	500.0

In [17]:
```
df.unstack('Product')
```

Out[17]:

		Sales	
	Product	A	B
Store	Date		
Store1	2024-07-01	100.0	200.0
	2024-07-02	150.0	NaN
Store2	2024-07-01	300.0	NaN
	2024-07-02	NaN	400.0
	2024-07-03	NaN	500.0

Con un DataFrame, cualquiera de los ejes puede tener un índice jerárquico:

In [18]:
```
frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
                     index=[["a", "a", "b", "b"], [1, 2, 1, 2]],
                     columns=[["Ohio", "Ohio", "Colorado"],
                     ["Green", "Red", "Green"]])
frame
```

Out[18]:

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Los niveles jerárquicos pueden tener nombres (como cadenas o cualquier objeto Python). Si es así, aparecerán en la salida de la consola:

In [19]:
```
frame.index.names = ["key1", "key2"]
frame.columns.names = ["state", "color"]
frame
```

Out[19]:

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Estos nombres sustituyen al atributo name, que sólo se utiliza con índices de un solo nivel. Puedes ver cuántos niveles tiene un índice accediendo a su atributo `nlevels`:

In [20]:
```
frame.index.nlevels
```

Out[20]: 2

In [21]:
```
frame.columns.nlevels
```

Out[21]: 2

Con la indexación parcial de columnas puede seleccionar grupos de columnas de forma similar:

In [22]:
```
frame["Ohio"]
```

Out[22]:

	color	Green	Red
key1	key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

Un `MultiIndex` puede crearse por sí mismo y luego reutilizarse; las columnas del DataFrame anterior con nombres de nivel también podrían crearse así:

In [23]:
```
pd.MultiIndex.from_arrays([["Ohio", "Ohio", "Colorado"],
                           ["Green", "Red", "Green"]],
                           names=["state", "color"])
```

Out[23]:
```
MultiIndex([(    'Ohio', 'Green'),
            (    'Ohio',   'Red'),
            ('Colorado', 'Green')],
           names=['state', 'color'])
```

Reordenación y clasificación de niveles

A veces puede ser necesario reorganizar el orden de los niveles en un eje u ordenar los datos por los valores de un nivel específico. El método `swaplevel` toma dos números o nombres de nivel y devuelve un nuevo objeto con los niveles intercambiados (pero los datos permanecen inalterados):

In [24]:
```
frame
```

Out[24]:

	state	Ohio	Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [25]:
```
frame.swaplevel("key1", "key2")
```

Out[25]:

	state	Ohio	Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

`sort_index` ordena por defecto los datos lexicográficamente utilizando todos los niveles del índice, pero puede elegir utilizar sólo un nivel o un subconjunto de niveles para ordenar pasando el argumento `level`. Por ejemplo:

In [26]:
```
frame.sort_index()
```

Out[26]:

	state	Ohio	Red	Colorado
	color	Green		Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [27]:
```
frame.sort_index(level=0)
```

Out[27]:

	state	Ohio	Red	Colorado
	color	Green		Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

In [28]:
```
frame.sort_index(level=1)
```

Out[28]:

	state	Ohio	Red	Colorado
	color	Green		Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

In [29]:
```
frame.swaplevel(0, 1)
```

Out[29]:

	state	Ohio	Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

In [30]:
```
frame.swaplevel(0, 1).sort_index(level=0)
#
```

Out[30]:

	state	Ohio	Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

> El rendimiento de la selección de datos es mucho mejor en objetos indexados jerárquicamente si el índice está ordenado lexicográficamente empezando por el nivel más externo, es decir, el resultado de llamar a `sort_index(level=0)` o `sort_index()`.

Resumen estadístico por niveles

Muchas estadísticas descriptivas y de resumen en DataFrame y Series tienen una opción de nivel en la que puede especificar el nivel por el que desea agregar en un eje concreto. Considere el DataFrame anterior; podemos agregar por nivel en las filas o columnas, así:

In [31]:
```
frame.groupby(level="key2").sum()
```

Out[31]:

state	Ohio	Colorado
color	Green	Red	Green
key2			
1	6	8	10
2	12	14	16

In [32]:
```
frame.groupby(level="color", axis="columns").sum()
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_6868\775557097.py:1: FutureWarning: D
ataFrame.groupby with axis=1 is deprecated. Do `frame.T.groupby(...)` without axi
s instead.
  frame.groupby(level="color", axis="columns").sum()
```

Out[32]:

	color	Green	Red
key1	key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

Indexación con las columnas de un DataFrame

No es inusual querer utilizar una o más columnas de un DataFrame como índice de fila; alternativamente, puede desear mover el índice de fila a las columnas del DataFrame. He aquí un ejemplo de DataFrame:

In [33]:
```
frame = pd.DataFrame({"a": range(7), "b": range(7, 0, -1),
                      "c": ["one", "one", "one", "two", "two",
                            "two", "two"],
                      "d": [0, 1, 2, 0, 1, 2, 3]})
frame
```

Out[33]:

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

La función `set_index` de DataFrame creará un nuevo DataFrame utilizando una o más de sus columnas como índice:

In [34]:
```
frame2 = frame.set_index(["c", "d"])
frame2
```

Out[34]:

		a	b
c	d		
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

Por defecto, las columnas se eliminan del DataFrame, aunque puede dejarlas pasando `drop=False` a `set_index`:

In [35]:
```
frame.set_index(["c", "d"], drop=False)
```

Out[35]:

		a	b	c	d
c	d				
one	0	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	0	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

`reset_index`, por otro lado, hace lo contrario que set_index; los niveles de índice jerárquico se mueven a las columnas:

In [36]:
```
frame2.reset_index()
```

Out[36]:

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

3.2 Combinar y fusionar conjuntos de datos

Los datos contenidos en los objetos pandas pueden combinarse de varias maneras:

`pandas.merge` : Conectar filas en DataFrames basándose en una o más claves. Esto resultará familiar a los usuarios de SQL u otras bases de datos relacionales, ya que implementa operaciones de unión de bases de datos.

`pandas.concat` : Concatena o "apila" objetos a lo largo de un eje.

`combine_first` : Empalma (Splice) datos superpuestos para rellenar los valores que faltan en un objeto con valores de otro.

Uniones de DataFrames al estilo de las bases de datos

Las operaciones de `Merge` o `join` combinan conjuntos de datos enlazando filas mediante una o varias claves. Estas operaciones son particularmente importantes en bases de datos relacionales (ejm SQL). La función `pandas.merge` en pandas es el principal punto de entrada para utilizar estos algoritmos en sus datos. Empecemos con un ejemplo sencillo:

In [37]:
```
df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"],
                    "data1": pd.Series(range(7), dtype="Int64")})
df1
```

Out[37]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [38]:
```
df2 = pd.DataFrame({"key": ["a", "b", "d"],
                    "data2": pd.Series(range(3), dtype="Int64")})

df2
```

Out[38]:

	key	data2
0	a	0
1	b	1
2	d	2

Este es un ejemplo de un join muchos-a-uno; los datos en df1 tienen múltiples filas etiquetadas como a y b, mientras que df2 tiene sólo una fila para cada valor en la columna clave. Llamando a `pandas.merge` con estos objetos, obtenemos:

In [39]:
```
pd.merge(df1, df2)
```

Out[39]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

Nótese que no se ha especificado en qué columna unir. Si no se especifica esa información, `pandas.merge` utiliza los nombres de las columnas solapadas como claves. Sin embargo, es una buena práctica especificarlo explícitamente:

In [40]:
```
pd.merge(df1, df2, on="key")
```

Out[40]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

En general, el orden de salida de las columnas en las operaciones `pandas.merge` no está especificado.

Si los nombres de las columnas son diferentes en cada objeto, puede especificarlos por separado:

In [41]:
```
df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c", "a", "a", "b"],
                    "data1": pd.Series(range(7), dtype="Int64")})

df3
```

Out[41]:

	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [42]:
```
df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
                    "data2": pd.Series(range(3), dtype="Int64")})
df4
```

Out[42]:

	rkey	data2
0	a	0
1	b	1
2	d	2

In [43]:
```
pd.merge(df3, df4, left_on="lkey", right_on="rkey")
```

Out[43]:

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	a	4	a	0
4	a	5	a	0
5	b	6	b	1

Puede notar que los valores "c" y "d" y los datos asociados faltan en el resultado. Por defecto, `pandas.merge` hace una unión "inner" (interna); las claves en el resultado son la intersección, o el conjunto común encontrado en ambas tablas. Otras opciones posibles son "left", "right" y "outer". La unión externa toma la unión de las claves, combinando el efecto de aplicar las uniones izquierda y derecha:

In [44]:
```
pd.merge(df1, df2, how="outer")
```

Out[44]:

	key	data1	data2
0	a	2	0
1	a	4	0
2	a	5	0
3	b	0	1
4	b	1	1
5	b	6	1
6	c	3	<NA>
7	d	<NA>	2

In [45]:
```
pd.merge(df3, df4, left_on="lkey", right_on="rkey", how="outer")
```

Out[45]:

	lkey	data1	rkey	data2
0	a	2	a	0
1	a	4	a	0
2	a	5	a	0
3	b	0	b	1
4	b	1	b	1
5	b	6	b	1
6	c	3	NaN	<NA>
7	NaN	<NA>	d	2

En una unión externa (outer join), las filas de los objetos DataFrame izquierdo o derecho que no coincidan en las claves del otro DataFrame aparecerán con valores NA en las columnas del otro DataFrame para las filas que no coincidan.

`how=inner` : Utiliza sólo las combinaciones de claves observadas en ambas tablas

`how="left"` : Utiliza todas las combinaciones de claves de la tabla de la izquierda

`how="right"` : Utiliza todas las combinaciones de claves de la tabla de la derecha

`how="outer"` : Utilizar conjuntamente todas las combinaciones de claves observadas en ambas tablas

Las fusiones (merges) de muchos a muchos forman el producto cartesiano de las claves coincidentes. He aquí un ejemplo:

In [46]:
```
df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
                    "data1": pd.Series(range(6), dtype="Int64")})

df1
```

Out[46]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [47]:
```
df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
                    "data2": pd.Series(range(5), dtype="Int64")})
df2
```

Out[47]:

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

In [48]:
```
pd.merge(df1, df2, on="key", how="left")
```

Out[48]:

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	a	2	0
5	a	2	2
6	c	3	<NA>
7	a	4	0
8	a	4	2
9	b	5	1
10	b	5	3

Como había tres filas "b" en el DataFrame izquierdo y dos en el derecho, hay seis filas "b" en el resultado. El método join pasado al argumento de la palabra clave `how` afecta sólo a los valores clave distintos que aparecen en el resultado:

In [49]:
```
pd.merge(df1, df2, how="inner")
```

Out[49]:

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	a	2	0
5	a	2	2
6	a	4	0
7	a	4	2
8	b	5	1
9	b	5	3

Para combinar (merge) con varias claves, pase una lista de nombres de columnas:

In [50]:
```
left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
                     "key2": ["one", "two", "one"],
                     "lval": pd.Series([1, 2, 3], dtype='Int64')})
left
```

Out[50]:

	key1	key2	lval
0	foo	one	1
1	foo	two	2
2	bar	one	3

In [51]:
```
right = pd.DataFrame({"key1": ["foo", "foo", "bar", "bar"],
                      "key2": ["one", "one", "one", "two"],
                      "rval": pd.Series([4, 5, 6, 7], dtype='Int64')})
right
```

Out[51]:

	key1	key2	rval
0	foo	one	4
1	foo	one	5
2	bar	one	6
3	bar	two	7

In [52]:
```
pd.merge(left, right, on=["key1", "key2"], how="outer")
```

Out[52]:

	key1	key2	lval	rval
0	bar	one	3	6
1	bar	two	<NA>	7
2	foo	one	1	4
3	foo	one	1	5
4	foo	two	2	<NA>

Una última cuestión a tener en cuenta en las operaciones de fusión (merge) es el tratamiento de los nombres de columnas que se solapan. Por ejemplo:

In [53]:
```
pd.merge(left, right, on="key1")
```

Out[53]:

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

Mientras que puede tratar el solapamiento manualmente, `pandas.merge` tiene una opción de `suffixes` para especificar cadenas a añadir a los nombres solapados en los

objetos DataFrame izquierdo y derecho:

In [54]:
```
pd.merge(left, right, on="key1", suffixes=("_left", "_right"))
```

Out[54]:

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

Repaso de los tipos básicos de Join

- `Inner Join`

> Devuelve un DataFrame que contiene solo las filas donde hay una coincidencia en ambas tablas. Las filas no coincidentes se descartan.

En pandas sería:

```
result = pd.merge(df1, df2, how='inner', on='column_name')
```

`how='inner'` especifica el tipo de unión.
`on='column_name'` indica la columna en la que se basa la unión.

- `Right join`

> Devuelve todas las filas del DataFrame de la derecha y las filas coincidentes del DataFrame de la izquierda. Si no hay coincidencia, las filas del DataFrame de la izquierda tendrán valores NaN.

En pandas sería

```
result = pd.merge(df1, df2, how='right', on='column_name')
```

- `Left Join`

> Devuelve todas las filas del DataFrame de la izquierda y las filas coincidentes del DataFrame de la derecha. Si no hay coincidencia, las filas del DataFrame de la derecha tendrán valores NaN.

En pandas sería

```
result = pd.merge(df1, df2, how='left', on='column_name')
```

- `Outer Join` = `Left Join` + `Right Join`

> Devuelve todas las filas cuando hay una coincidencia en una de las tablas. Si no hay coincidencia, se utilizarán valores NaN para las filas faltantes.

En pandas sería

```
result = pd.merge(df1, df2, how='outer', on='column_name')
```

Ejemplo 3.2:

Partiendo de los siguientes DataFrames dados abajo, repsonder las 'preguntas de negocio'.

Dataframe 1: Información de clientes

In [55]:
```
clientes = pd.DataFrame({
    'ClienteID': [1, 2, 3, 4, 5, 6],
    'Nombre': ['Ana', 'Luis', 'María', 'Carlos', 'Laura', 'Jorge'],
    'Edad': [34, 45, 23, 54, 31, 40],
    'Ciudad': ['Madrid', 'Barcelona', 'Madrid', 'Sevilla', 'Valencia', 'Bilbao'],
    'Email': ['ana@example.com', 'luis@example.com', 'maria@example.com',
              'carlos@example.com', 'laura@example.com', 'jorge@example.com'],
    'FechaRegistro': ['2020-01-01', '2020-06-15', '2021-03-20',
                      '2021-07-30', '2022-02-17', '2022-05-25']
})

print("DataFrame de Clientes:")
clientes
```

```
DataFrame de Clientes:
```

Out[55]:

	ClienteID	Nombre	Edad	Ciudad	Email	FechaRegistro
0	1	Ana	34	Madrid	ana@example.com	2020-01-01
1	2	Luis	45	Barcelona	luis@example.com	2020-06-15
2	3	María	23	Madrid	maria@example.com	2021-03-20
3	4	Carlos	54	Sevilla	carlos@example.com	2021-07-30
4	5	Laura	31	Valencia	laura@example.com	2022-02-17
5	6	Jorge	40	Bilbao	jorge@example.com	2022-05-25

Dataframe 2: Información de contratos

In [56]:
```
contratos = pd.DataFrame({
    'ContratoID': [101, 102, 103, 104, 105, 106, 107],
```

```
    'ClienteID': [1, 2, 2, 3, 5, 6, 7],
    'Tipo': ['Móvil', 'Internet', 'Televisión', 'Móvil', 'Internet', 'Televisión
    'FechaInicio': ['2021-01-15', '2021-06-23', '2021-03-10',
                    '2021-08-05', '2021-12-12', '2021-07-19', '2022-01-10'],
    'DuraciónMeses': [24, 12, 12, 24, 12, 24, 12],
    'CostoMensual': [20, 40, 50, 25, 45, 55, 30]
})

print("DataFrame de Contratos:")
contratos
```

```
DataFrame de Contratos:
```

Out[56]:

	ContratoID	ClienteID	Tipo	FechaInicio	DuraciónMeses	CostoMensual
0	101	1	Móvil	2021-01-15	24	20
1	102	2	Internet	2021-06-23	12	40
2	103	2	Televisión	2021-03-10	12	50
3	104	3	Móvil	2021-08-05	24	25
4	105	5	Internet	2021-12-12	12	45
5	106	6	Televisión	2021-07-19	24	55
6	107	7	Móvil	2022-01-10	12	30

Preguntas

1- (Inner Join)¿Cuáles son los detalles de contacto de los clientes que tienen contratos activos?

In [57]:
```
clientes_con_contratos = pd.merge(clientes, contratos, on='ClienteID', how='inne
print("Detalles de contacto de clientes con contratos activos:")
print(clientes_con_contratos[['Nombre', 'Email', 'Tipo', 'FechaInicio']])
```

```
Detalles de contacto de clientes con contratos activos:
  Nombre              Email        Tipo FechaInicio
0    Ana    ana@example.com       Móvil  2021-01-15
1   Luis   luis@example.com    Internet  2021-06-23
2   Luis   luis@example.com  Televisión  2021-03-10
3  María  maria@example.com       Móvil  2021-08-05
4  Laura  laura@example.com    Internet  2021-12-12
5  Jorge  jorge@example.com  Televisión  2021-07-19
```

2- ¿Qué clientes se registraron antes de obtener un contrato?

In [58]:
```
clientes_con_contratos = pd.merge(clientes, contratos, on='ClienteID', how='inne
clientes_antes_contrato = clientes_con_contratos[clientes_con_contratos['FechaRe
print("Clientes que se registraron antes de obtener un contrato:")
print("#########################################################################")
print(clientes_antes_contrato[['Nombre', 'FechaRegistro', 'FechaInicio']])
```

```
Clientes que se registraron antes de obtener un contrato:
########################################################################
  Nombre FechaRegistro FechaInicio
0    Ana    2020-01-01  2021-01-15
1   Luis    2020-06-15  2021-06-23
2   Luis    2020-06-15  2021-03-10
3  María    2021-03-20  2021-08-05
```

3- ¿Qué clientes no tienen ningún contrato asociado y cuáles son sus detalles de contacto?

In [59]:
```
clientes_sin_contratos = pd.merge(clientes, contratos, on='ClienteID', how='left
clientes_sin_contratos = clientes_sin_contratos[clientes_sin_contratos['_merge']
# filtra el resultado de la unión para conservar
# solo las filas donde el valor en la columna _merge
# es left_only.

print("Clientes sin contratos asociados:")
print('############################################################')
print(clientes_sin_contratos[['Nombre', 'Email', 'Ciudad']])
```

```
Clientes sin contratos asociados:
############################################################
   Nombre              Email   Ciudad
4  Carlos  carlos@example.com  Sevilla
```

La línea:

- `clientes_sin_contratos = clientes_sin_contratos[clientes_sin_contratos['_merge'] == 'left_only']`

Significa que se seleccionan únicamente las filas que están presentes en el DataFrame de clientes pero no tienen correspondencia en el DataFrame de contratos.

El parámetro `indicator=True` agrega una columna extra llamada `_merge` al resultado de la unión. Esta columna indica el origen de cada fila en el resultado de la unión:

- `left_only` si la fila solo está presente en el DataFrame de la izquierda.
- `right_only` si la fila solo está presente en el DataFrame de la derecha.
- `both` si la fila está presente en ambos DataFrames.

4-¿Qué contratos no tienen un cliente asociado en la base de datos de clientes y cuáles son los detalles del contrato?

In [60]:
```
contratos_sin_clientes = pd.merge(clientes, contratos, on='ClienteID', how='righ

contratos_sin_clientes = contratos_sin_clientes[contratos_sin_clientes['_merge']
# filtra el resultado de la unión para conservar
# solo las filas donde el valor en la columna _merge
# es right_only.

print("Contratos sin clientes asociados:")
```

```
print('############################################################')
print(contratos_sin_clientes[['ContratoID', 'Tipo', 'FechaInicio', 'CostoMensual
```

```
Contratos sin clientes asociados:
############################################################
   ContratoID   Tipo FechaInicio  CostoMensual
6         107  Móvil  2022-01-10            30
```

La linea:

- `contratos_sin_clientes = contratos_sin_clientes[contratos_sin_clientes['_merge'] == 'right_only']`

Esto significa que se seleccionan únicamente las filas que están presentes en el DataFrame de contratos pero no tienen correspondencia en el DataFrame de clientes.

5-Obtener una lista completa de todos los clientes y todos los contratos, independientemente de si tienen coincidencias en las otras tablas.

In [61]:
```
todos_clientes_contratos = pd.merge(clientes, contratos, on='ClienteID', how='ou
print("Todos los clientes y contratos:")
print('############################################################################')
print(todos_clientes_contratos)
```

```
Todos los clientes y contratos:
############################################################################
   ClienteID  Nombre  Edad     Ciudad              Email FechaRegistro  \
0          1     Ana  34.0     Madrid     ana@example.com    2020-01-01
1          2    Luis  45.0  Barcelona    luis@example.com    2020-06-15
2          2    Luis  45.0  Barcelona    luis@example.com    2020-06-15
3          3   María  23.0     Madrid   maria@example.com    2021-03-20
4          4  Carlos  54.0    Sevilla  carlos@example.com    2021-07-30
5          5   Laura  31.0   Valencia   laura@example.com    2022-02-17
6          6   Jorge  40.0     Bilbao   jorge@example.com    2022-05-25
7          7     NaN   NaN        NaN                 NaN           NaN

   ContratoID        Tipo FechaInicio  DuraciónMeses  CostoMensual      _merge
0       101.0       Móvil  2021-01-15           24.0          20.0        both
1       102.0    Internet  2021-06-23           12.0          40.0        both
2       103.0  Televisión  2021-03-10           12.0          50.0        both
3       104.0       Móvil  2021-08-05           24.0          25.0        both
4         NaN         NaN         NaN            NaN           NaN   left_only
5       105.0    Internet  2021-12-12           12.0          45.0        both
6       106.0  Televisión  2021-07-19           24.0          55.0        both
7       107.0       Móvil  2022-01-10           12.0          30.0  right_only
```

6- Cuáles son los clientes con contratos de más de 12 meses y cuáles son los detalles del contrato?

In [62]:
```
clientes_con_contratos_largos = pd.merge(clientes, contratos, on='ClienteID', ho
clientes_con_contratos_largos = clientes_con_contratos_largos[clientes_con_contr
print("Clientes con contratos de más de 12 meses:")
print(clientes_con_contratos_largos[['Nombre', 'Tipo', 'DuraciónMeses', 'CostoMe
```

```
Clientes con contratos de más de 12 meses:
  Nombre        Tipo  DuraciónMeses  CostoMensual
0    Ana       Móvil             24            20
3  María       Móvil             24            25
5  Jorge  Televisión             24            55
```

7- ¿Qué clientes tienen contratos de un costo mensual superior a 40?

In [63]:
```
clientes_con_contratos_caros = pd.merge(clientes, contratos, on='ClienteID', how
clientes_con_contratos_caros = clientes_con_contratos_caros[clientes_con_contrat
print("Clientes con contratos de un costo mensual superior a 40:")
print(clientes_con_contratos_caros[['Nombre', 'Tipo', 'CostoMensual']])
```

```
Clientes con contratos de un costo mensual superior a 40:
  Nombre        Tipo  CostoMensual
2   Luis  Televisión            50
4  Laura    Internet            45
5  Jorge  Televisión            55
```

Observar la siguiente tabla para una referencia de argumentos en `pandas.merge`. La siguiente sección cubre la unión (join) usando el índice de filas del DataFrame.

`left` : DataFrame a fusionar en el lado izquierdo.

`right` : DataFrame que se fusionará en el lado derecho.

`how` : Tipo de join a aplicar: "inner", "outer", "left", o "right"; por defecto es "inner".

`on` : Nombres de las columnas a unir (join). Deben encontrarse en ambos objetos DataFrame. Si no se especifica y no se dan otras claves de unión (join), se utilizará la intersección de los nombres de columna de la izquierda y la derecha como claves de join.

`left_on` : Columnas del DataFrame izquierdo (left) que se utilizarán como claves de unión (join). Puede ser un único nombre de columna o una lista de nombres de columna.

`right_on` : Análogo a left_on para DataFrame derecho(right).

`left_index` : Utiliza el índice de la fila de la izquierda como clave de unión(join) (o claves, si es un MultiIndex).

`right_index` : Análogo a `left_index`

`sort` : Ordena los datos fusionados lexicográficamente por las claves de unión (join); Falso por defecto.

`suffixes` : Tuple de valores de cadena para añadir a los nombres de columna en caso de solapamiento; por defecto ("_x", "_y") (por ejemplo, si "data" está en ambos objetos DataFrame, aparecerá como "data_x" y "data_y" en el resultado).

`copy` : Si es False, evita copiar datos en la estructura de datos resultante en algunos casos excepcionales; por defecto siempre copia.

`validate` : Verifica si la fusión (merge) es del tipo especificado, ya sea uno a uno, uno a muchos o muchos a muchos. Consulte el docstring para obtener más información sobre las opciones.

`indicator`: Añade una columna especial `_merge` que indica el origen de cada fila; los valores serán "left_only", "right_only" o "both" en función del origen de los datos unidos (joined) en cada fila.

Merging en índice (Index)

En algunos casos, la(s) clave(s) de fusión(merging) de un DataFrame se encuentra(n) en su índice (etiquetas de fila). En este caso, puede pasar `left_index=True` o `right_index=True` (o ambos) para indicar que el índice debe utilizarse como clave de fusión:

In [64]:
```
left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
                      "value": pd.Series(range(6), dtype="Int64")})
left1
```

Out[64]:

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

In [65]:
```
right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])
right1
```

Out[65]:

	group_val
a	3.5
b	7.0

In [66]:
```
pd.merge(left1, right1, left_on="key", right_index=True)
```

Out[66]:

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0

- `left_on="key"` significa que vamos a usar la columna `key` del DataFrame `left1` como la clave para la unión.

- `right_index=True` significa que vamos a usar el índice del DataFrame `right1` como la clave para la unión.

Ejemplo 3.3

Supongamos que tenemos dos DataFrames: uno con información de ventas de dispositivos y otro con detalles de los dispositivos.

In [67]:
```
# Dataframe ventas
ventas = pd.DataFrame({
    'VentaID': [1, 2, 3, 4, 5],
    'DispositivoID': [101, 102, 101, 103, 102],
    'Cantidad': [10, 5, 8, 7, 6]
})

# Dataframe dispositivos
dispositivos = pd.DataFrame({
    'NombreDispositivo': ['iPhone 13', 'Samsung Galaxy S21', 'Google Pixel 6'],
    'Precio': [799, 999, 599]
}, index=[101, 102, 103])

print("DataFrame de Ventas:")
ventas
```

```
DataFrame de Ventas:
```

Out[67]:

	VentaID	DispositivoID	Cantidad
0	1	101	10
1	2	102	5
2	3	101	8
3	4	103	7
4	5	102	6

In [68]:
```
print("\nDataFrame de Dispositivos:")
print(dispositivos)
```

```
DataFrame de Dispositivos:
      NombreDispositivo  Precio
101           iPhone 13     799
102  Samsung Galaxy S21     999
103       Google Pixel 6     599
```

Ahora realizamos el merge "utilizando la columna `DispositivoID` del DataFrame izquierdo (ventas) como clave para la unión". Además se usará el índice del DataFrame derecho (`dispositivos`) como clave para la unión.

In [69]:
```
# Realizar el merge
resultado = pd.merge(ventas, dispositivos, left_on='DispositivoID', right_index=

print("\nResultado del Merge:")
resultado
```

```
Resultado del Merge:
```

Out[69]:

	VentaID	DispositivoID	Cantidad	NombreDispositivo	Precio
0	1	101	10	iPhone 13	799
1	2	102	5	Samsung Galaxy S21	999
2	3	101	8	iPhone 13	799
3	4	103	7	Google Pixel 6	599
4	5	102	6	Samsung Galaxy S21	999

Dado que el método de fusión (merge) por defecto es intersecar las claves de unión (join) , puede formar la unión de ellas con una unión (join) externa:

In [70]:
```
pd.merge(left1, right1, left_on="key", right_index=True, how="outer")
```

Out[70]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Con datos indexados jerárquicamente, las cosas son más complicadas, ya que la unión (join) sobre índice equivale a una fusión (merge) de varias claves:

In [71]:
```
lefth = pd.DataFrame({"key1": ["Ohio", "Ohio", "Ohio",
                               "Nevada", "Nevada"],
                      "key2": [2000, 2001, 2002, 2001, 2002],
                      "data": pd.Series(range(5), dtype="Int64")})
lefth
```

Out[71]:

	key1	key2	data
0	Ohio	2000	0
1	Ohio	2001	1
2	Ohio	2002	2
3	Nevada	2001	3
4	Nevada	2002	4

In [72]:
```
righth_index = pd.MultiIndex.from_arrays(
    [
        ["Nevada", "Nevada", "Ohio", "Ohio", "Ohio", "Ohio"],
        [2001, 2000, 2000, 2000, 2001, 2002]
    ]
)
```

```
# Es un índice jerárquico (MultiIndex) creado a partir de dos
# listas: la primera contiene nombres de estados y la segunda contiene años.
```

In [73]:
```
righth = pd.DataFrame({"event1": pd.Series([0, 2, 4, 6, 8, 10], dtype="Int64",
                                           index=righth_index),
                       "event2": pd.Series([1, 3, 5, 7, 9, 11], dtype="Int64",
                                           index=righth_index)})
righth
```

Out[73]:

		event1	event2
Nevada	**2001**	0	1
	2000	2	3
Ohio	**2000**	4	5
	2000	6	7
	2001	8	9
	2002	10	11

Los valores de event1 y event2 son series de números enteros con el índice jerárquico righth_index.

In [74]:
```
pd.merge(lefth, righth, left_on=["key1", "key2"], right_index=True)
```

Out[74]:

	key1	key2	data	event1	event2
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11
3	Nevada	2001	3	0	1

- `pd.merge` combina los DataFrames lefth y righth utilizando las columnas key1 y key2 de lefth como claves para la unión.
- `right_index=True` especifica que el índice jerárquico de righth se usará como clave para la unión.

Si se desea indicar varias columnas para hacer un merge, debe pasarlas en forma de lista (tenga en cuenta el tratamiento de los valores de índice duplicados con `how="outer"`):

In [75]:
```
pd.merge(lefth, righth, left_on=["key1", "key2"],
         right_index=True, how="outer")
```

Out[75]:

	key1	key2	data	event1	event2
4	Nevada	2000	<NA>	2	3
3	Nevada	2001	3	0	1
4	Nevada	2002	4	<NA>	<NA>
0	Ohio	2000	0	4	5
0	Ohio	2000	0	6	7
1	Ohio	2001	1	8	9
2	Ohio	2002	2	10	11

Ejemplo 3.4

De vuelta a los dataframes:

In [76]:
```
# Dataframe ventas
ventas = pd.DataFrame({
    'Tienda': ['Tienda1', 'Tienda1', 'Tienda2', 'Tienda2', 'Tienda3'],
    'Modelo': ['iPhone 13', 'Samsung S21', 'iPhone 13', 'Google Pixel 6', 'Samsu
    'Capacidad': ['64GB', '128GB', '128GB', '128GB', '256GB'],  # Nueva columna
    'Fecha': ['2023-01-01', '2023-01-02', '2023-01-01', '2023-01-03', '2023-01-0
    'Cantidad': [5, 3, 2, 4, 1]
})
# Dataframe detalles
detalles_index = pd.MultiIndex.from_arrays(
    [
        ['iPhone 13', 'Samsung S21', 'Google Pixel 6', 'iPhone 13', 'Samsung S21
        ['64GB', '128GB', '128GB', '128GB', '256GB']
    ], names=['Modelo', 'Capacidad']
)

# Crear DataFrame de Detalles
detalles = pd.DataFrame({
    'Precio': [799, 999, 699, 899, 1099],
    'Color': ['Negro', 'Blanco', 'Negro', 'Blanco', 'Azul']
}, index=detalles_index)

print("DataFrame de Ventas:")
ventas
```

```
DataFrame de Ventas:
```

Out[76]:

	Tienda	Modelo	Capacidad	Fecha	Cantidad
0	Tienda1	iPhone 13	64GB	2023-01-01	5
1	Tienda1	Samsung S21	128GB	2023-01-02	3
2	Tienda2	iPhone 13	128GB	2023-01-01	2
3	Tienda2	Google Pixel 6	128GB	2023-01-03	4
4	Tienda3	Samsung S21	256GB	2023-01-04	1

In [77]:
```
print("\nDataFrame de Detalles:")
detalles
```

```
DataFrame de Detalles:
```

Out[77]:

		Precio	Color
Modelo	**Capacidad**		
iPhone 13	**64GB**	799	Negro
Samsung S21	**128GB**	999	Blanco
Google Pixel 6	**128GB**	699	Negro
iPhone 13	**128GB**	899	Blanco
Samsung S21	**256GB**	1099	Azul

Se pide combinar el dataframe `ventas` con el dataframe `detalles`. En este caso se usará la columna `Modelo` de la dataframe ventas, además se usará el dataframe `detalles` para la creación de indices jerárquicos.

In [78]:
```
# Realizar el merge
resultado = pd.merge(ventas, detalles, left_on=['Modelo', 'Capacidad'], right_in
print("\nResultado del Merge:")
resultado
```

```
Resultado del Merge:
```

Out[78]:

	Tienda	Modelo	Capacidad	Fecha	Cantidad	Precio	Color
0	Tienda1	iPhone 13	64GB	2023-01-01	5	799	Negro
1	Tienda1	Samsung S21	128GB	2023-01-02	3	999	Blanco
2	Tienda2	iPhone 13	128GB	2023-01-01	2	899	Blanco
3	Tienda2	Google Pixel 6	128GB	2023-01-03	4	699	Negro
4	Tienda3	Samsung S21	256GB	2023-01-04	1	1099	Azul

- `left_on=['Modelo', 'Capacidad']:`

Este parámetro especifica que se deben usar las columnas `Modelo` y `Capacidad` del DataFrame `ventas` como las claves para la unión.

- `right_index=True:`

Indica que el índice del DataFrame derecho (`detalles`) se utilizará como la clave para la unión. `detalles` tiene un índice jerárquico (MultiIndex) compuesto por `Modelo` y `Capacidad`.

Merge con todos los indices

También es posible utilizar los índices de ambos lados del merge:

In [79]:
```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                     index=["a", "c", "e"],
                     columns=["Ohio", "Nevada"]).astype("Int64")
left2
```

Out[79]:

	Ohio	Nevada
a	1	2
c	3	4
e	5	6

In [80]:
```
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                      index=["b", "c", "d", "e"],
                      columns=["Missouri", "Alabama"]).astype("Int64")
right2
```

Out[80]:

	Missouri	Alabama
b	7	8
c	9	10
d	11	12
e	13	14

In [81]:
```
pd.merge(left2, right2, how="outer", left_index=True, right_index=True)
```

Out[81]:

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	11	12
e	5	6	13	14

Uso de Join en Pandas

En `pandas.DataFrame` tambien hay un método de instancia `join` para simplificar la combinación por índice. También se puede utilizar para combinar varios objetos DataFrame que tengan índices iguales o similares pero columnas que no se solapen. En el ejemplo anterior, podríamos haber escrito:

In [82]:
```
left2.join(right2, how="outer")
```

Out[82]:

	Ohio	Nevada	Missouri	Alabama
a	1	2	<NA>	<NA>
b	<NA>	<NA>	7	8
c	3	4	9	10
d	<NA>	<NA>	11	12
e	5	6	13	14

Comparado con `pandas.merge`, el método `join` de DataFrame realiza una unión a la izquierda (left) en las claves de unión por defecto. También soporta unir el índice del DataFrame pasado en una de las columnas del DataFrame al que se está llamando:

In [83]:
```
left1.join(right1, on="key")
```

Out[83]:

	key	value	group_val
0	a	0	3.5
1	b	1	7.0
2	a	2	3.5
3	a	3	3.5
4	b	4	7.0
5	c	5	NaN

Puede pensar en este método como una unión de datos "dentro" del objeto cuyo método join fue llamado.

Por último, para simples fusiones (merge) índice sobre índice, puede pasar una lista de DataFrames a unir como alternativa al uso de la función más general `pandas.concat` descrita en la siguiente sección:

In [84]:
```
another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
                       index=["a", "c", "e", "f"],
                       columns=["New York", "Oregon"])
another
```

Out[84]:

	New York	Oregon
a	7.0	8.0
c	9.0	10.0
e	11.0	12.0
f	16.0	17.0

Repasemos que tienen left2 y right2

In [85]:
```
right2
```

Out[85]:

	Missouri	Alabama
b	7	8
c	9	10
d	11	12
e	13	14

In [86]:
```
left2
```

Out[86]:

	Ohio	Nevada
a	1	2
c	3	4
e	5	6

In [87]:
```
left2.join([right2, another])
```

Out[87]:

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1	2	<NA>	<NA>	7.0	8.0
c	3	4	9	10	9.0	10.0
e	5	6	13	14	11.0	12.0

In [88]:
```
left2.join([right2, another], how="outer")
```

Out[88]:

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1	2	<NA>	<NA>	7.0	8.0
c	3	4	9	10	9.0	10.0
e	5	6	13	14	11.0	12.0
b	<NA>	<NA>	7	8	NaN	NaN
d	<NA>	<NA>	11	12	NaN	NaN
f	<NA>	<NA>	<NA>	<NA>	16.0	17.0

Concatenar a lo largo de un eje

Otro tipo de operación de combinación de datos se denomina indistintamente concatenación o apilamiento (stacking). La función concatenar de NumPy puede hacer esto con matrices NumPy:

In [89]:
```
arr = np.arange(12).reshape((3, 4))
arr
```

Out[89]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [90]:
```
np.concatenate([arr, arr], axis=1)
```

Out[90]:
```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

En el contexto de objetos pandas como Series y DataFrame, tener ejes etiquetados permite generalizar aún más la concatenación de arrays. En particular, usted tiene un número de preocupaciones adicionales:

- Si los objetos están indexados de forma diferente en los otros ejes, ¿debemos combinar los elementos distintos en estos ejes o utilizar sólo los valores en común?
- ¿Es necesario que los trozos (chunk) de datos concatenados sean identificables como tales en el objeto resultante?
- ¿Contiene el "eje de concatenación" datos que deban conservarse? En muchos casos, es mejor descartar las etiquetas enteras por defecto de un DataFrame durante la concatenación.

La función `concat` en pandas proporciona una forma consistente de abordar cada una de estas cuestiones. Se mostrará una serie de ejemplos para ilustrar cómo funciona.

> El método `.concat()` de pandas se utiliza para concatenar (combinar) dos o más objetos pandas a lo largo de un eje especificado (filas o columnas). Es útil para combinar DataFrames u otros objetos pandas de manera flexible.

Código:

`pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)`

Argumentos:

`objs` : Una lista o diccionario de objetos pandas a concatenar.
`axis` : El eje a lo largo del cual se concatenan los objetos (0 para filas, 1 para columnas). Por defecto es 0.
`join` : 'outer' (por defecto) para unión externa o 'inner' para unión interna.
`ignore_index` : Si es True, no usa los índices de los ejes concatenados, sino que los ignora y genera uno nuevo.
`keys` : Secuencia para utilizar como nivel de clave para los objetos concatenados.
`levels` : Niveles específicos (solo aplicable si keys se usa y es un MultiIndex). names: Nombres para los niveles en el MultiIndex resultante.
`verify_integrity` : Si es True, verifica que no haya duplicados en el índice resultante.

`sort` : Si es True, ordena las claves de unión al realizar concatenaciones.
`copy` : Si es True, realiza una copia de los datos (por defecto True).

Ejemplo 3.5

Vamos a crear los dataframes.

DataFrame 1: Ventas en la región Norte (Q1 y Q2)

In [91]:
```
data_norte = {
    'Producto': ['iPhone', 'Samsung Galaxy', 'Google Pixel', 'OnePlus', 'Xiaomi'
    'Ventas_Q1': [100, 150, 200, 130, 180, 90],
    'Ventas_Q2': [120, 180, 210, 140, 190, 100]
}

df_norte = pd.DataFrame(data_norte)
print("Ventas Región Norte:")
df_norte
```

```
Ventas Región Norte:
```

Out[91]:

	Producto	Ventas_Q1	Ventas_Q2
0	iPhone	100	120
1	Samsung Galaxy	150	180
2	Google Pixel	200	210
3	OnePlus	130	140
4	Xiaomi	180	190
5	Sony Xperia	90	100

DataFrame 2: Ventas en la región Sur (Q1 y Q2)

In [92]:
```
data_sur = {
    'Producto': ['iPhone', 'Samsung Galaxy', 'Google Pixel', 'OnePlus', 'Xiaomi'
    'Ventas_Q1': [80, 160, 220, 110, 170, 95],
    'Ventas_Q2': [130, 170, 230, 120, 175, 105]
}

df_sur = pd.DataFrame(data_sur)
print("\nVentas Región Sur:")
df_sur
```

```
Ventas Región Sur:
```

Out[92]:

	Producto	Ventas_Q1	Ventas_Q2
0	iPhone	80	130
1	Samsung Galaxy	160	170
2	Google Pixel	220	230
3	OnePlus	110	120
4	Xiaomi	170	175
5	Sony Xperia	95	105

Se quiere obtener un único dataframe de las ventas de ambas regiones.

In [ ]:
```
# Concatenar los DataFrames a lo largo de filas
df_total = pd.concat([df_norte, df_sur], axis=0, ignore_index=True)
# El argumento ignore_index=True se utiliza para reiniciar el índice
# en el DataFrame resultante, creando un índice continuo.
print("\nVentas Totales:")
df_total
```

Repetir bloque anterior cambiando `axis=1` *y renombrar el dataframe por* `df_total_1`. *Explicar la salida.*

Ejercicio 9

Partiendo de los siguientes datos:

In [94]:
```
# Datos del departamento de Ingeniería
data_ingenieria = {
    'Empleado': ['Alice', 'Bob', 'Charlie', 'Dan', 'Ellen'],
    'Salario': [80000, 85000, 90000, 87000, 88000],
    'Departamento': ['Ingeniería'] * 5
}
# Datos del departamento de Marketing
data_marketing = {
    'Empleado': ['David', 'Eve', 'Frank', 'Gina', 'Hank'],
    'Salario': [70000, 75000, 80000, 72000, 78000],
    'Departamento': ['Marketing'] * 5
}
# Datos del departamento de Ventas
data_ventas = {
    'Empleado': ['Grace', 'Heidi', 'Ivan', 'Jack', 'Karen'],
    'Salario': [65000, 70000, 75000, 68000, 69000],
    'Departamento': ['Ventas'] * 5
}
# Datos del departamento de Recursos Humanos
data_rrhh = {
    'Empleado': ['Judy', 'Ken', 'Laura', 'Mike', 'Nina'],
    'Salario': [60000, 65000, 70000, 64000, 66000],
    'Departamento': ['Recursos Humanos'] * 5
}
```

Pregunta 9.1 Crear los dataframes de cada departamento y luego concatenarlos para obtener un único dataframe final

In [ ]:

> **Ejemplo con Series.** Supongamos que tenemos tres Series sin solapamiento de índices (index overlap):

In [98]:
```
s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")
s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64")
s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
```

Al llamar a `pandas.concat` con estos objetos en una lista se pegan los valores y los índices:

In [99]:
```
s1
```

Out[99]:
```
a    0
b    1
dtype: Int64
```

In [100…
```
s2
```

Out[100…
```
c    2
d    3
e    4
dtype: Int64
```

In [101…
```
s3
```

Out[101…
```
f    5
g    6
dtype: Int64
```

In [102…
```
pd.concat([s1, s2, s3])
```

Out[102…
```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: Int64
```

Por defecto, `pandas.concat` funciona con `axis="index"`, produciendo otra Serie. Si pasa `axis="columns"`, el resultado será un DataFrame:

In [103…
```
pd.concat([s1, s2, s3], axis="columns", keys=["one", "two", "three"])
```

Out[103…

	one	two	three
a	0	<NA>	<NA>
b	1	<NA>	<NA>
c	<NA>	2	<NA>
d	<NA>	3	<NA>
e	<NA>	4	<NA>
f	<NA>	<NA>	5
g	<NA>	<NA>	6

`Columna one` : Contiene los valores de s1 con índices a y b. Para los demás índices (c, d, e, f, g), los valores son `<NA>`.
`Columna two` : Contiene los valores de s2 con índices c, d y e. Para los demás índices (a, b, f, g), los valores son `<NA>`.
`Columna three` : Contiene los valores de s3 con índices f y g. Para los demás índices (a, b, c, d, e), los valores son `<NA>`.
De esta manera, `pd.concat` crea un DataFrame combinando las tres series, alineando los índices y asignando NaN para los índices faltantes en cada serie respectiva.

En este caso no hay solapamiento en el otro eje, que como puede ver es la unión (el join "externo") de los índices. En su lugar, puede intersecarlos pasando join="inner":

Supongamos que hay solapamiento de índices

In [104…
```
s4 = pd.concat([s1, s3])
s4
```

Out[104…
```
a    0
b    1
f    5
g    6
dtype: Int64
```

In [105…
```
s1
```

Out[105…
```
a    0
b    1
dtype: Int64
```

Concatenamos s1 con s4 por columnas:

In [109…
```
pd.concat([s1, s4], axis="columns")
```

Out[109…

	0	1
a	0	0
b	1	1
f	<NA>	5
g	<NA>	6

Añadimos `join='inner'`

In [110…
```
pd.concat([s1, s4], axis="columns", join="inner")
```

Out[110…

	0	1
a	0	0
b	1	1

En este último ejemplo, las etiquetas "f" y "g" han desaparecido debido a la opción `join="inner"`.

Un problema potencial es que las piezas concatenadas no son identificables en el resultado. Supongamos, en cambio, que desea crear un índice jerárquico en el eje de concatenación. Para ello, utilice el argumento keys:

Se quiere concaternar s1, s2 y s3 pero añadiendo identificadores en las filas:

In [114…
```
result = pd.concat([s1, s1, s3], keys=["one", "two", "three"])
result
```

Out[114…
```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: Int64
```

Llevemos nuestra serie a un dataframe para obtener otra vista de los datos utilizando `unstack`:

In [116…
```
result.unstack()
```

Out[116…

	a	b	f	g
one	0	1	<NA>	<NA>
two	0	1	<NA>	<NA>
three	<NA>	<NA>	5	6

La misma lógica se extiende a los objetos DataFrame:

Supongamos dataframes con indices repetidos:

In [117…
```
df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=["a", "b", "c"],
                   columns=["one", "two"])
df1
```

Out[117…

	one	two
a	0	1
b	2	3
c	4	5

In [118…
```
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=["a", "c"],
                   columns=["three", "four"])
df2
```

Out[118…

	three	four
a	5	6
c	7	8

Al hacer la concatenación, debemos añadir tambien `keys` para poder saber a cuál dataframe pertencen las columnas

In [121…
```
pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
```

Out[121…

	level1		level2	
	one	**two**	**three**	**four**
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Aquí el argumento keys se utiliza para crear un índice jerárquico donde el primer nivel puede utilizarse para identificar cada uno de los objetos DataFrame concatenados.

Repetir la celda anterior pero usando `axis='index'`. Explicar la salida

Si pasa un diccionario de objetos en lugar de una lista, se utilizarán las claves del diccionario para la opción keys:

In [124…
```
pd.concat({"level1": df1, "level2": df2}, axis="columns")
```

Out[124…

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Por ejemplo, podemos nombrar los niveles de eje creados con el argumento `names`:

In [126…
```
pd.concat([df1, df2], axis="columns", keys=["level1", "level2"],
          names=["upper", "lower"])
```

Out[126…

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Una última consideración se refiere a los DataFrames en los que el índice de fila no contiene ningún dato relevante:

In [127…
```
df1 = pd.DataFrame(np.random.standard_normal((3, 4)),
                   columns=["a", "b", "c", "d"])
df1
```

Out[127…

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837

In [130…
```
df2 = pd.DataFrame(np.random.standard_normal((2, 3)),
                   columns=["b", "d", "a"])
df2
```

Out[130…

	b	d	a
0	0.802926	0.575721	1.381918
1	0.000992	-0.143492	-0.206282

En este caso, puede pasar `ignore_index=True`, que descarta los índices de cada DataFrame y concatena los datos sólo en las columnas, asignando un nuevo índice por defecto:

In [131…
```
pd.concat([df1, df2], ignore_index=True)
```

Out[131…

	a	b	c	d
0	1.248804	0.774191	-0.319657	-0.624964
1	1.078814	0.544647	0.855588	1.343268
2	-0.267175	1.793095	-0.652929	-1.886837
3	1.381918	0.802926	NaN	0.575721
4	-0.206282	0.000992	NaN	-0.143492

Combinación de datos con solapamiento

Existe otra situación de combinación de datos que no puede expresarse como una operación de fusión (merge) o concatenación. Puede tener dos conjuntos de datos con índices que se solapan total o parcialmente. Como ejemplo , considere la función `where` de NumPy, que realiza el equivalente orientado en arrays de una expresión if-else:

In [132…
```
a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
              index=["f", "e", "d", "c", "b", "a"])
a
```

Out[132…
```
f    NaN
e    2.5
d    0.0
c    3.5
b    4.5
a    NaN
dtype: float64
```

In [133…
```
b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
              index=["a", "b", "c", "d", "e", "f"])
b
```

Out[133…
```
a    0.0
b    NaN
c    2.0
d    NaN
e    NaN
f    5.0
dtype: float64
```

In [135…
```
np.where(pd.isna(a), b, a)
```

Out[135…
```
array([0. , 2.5, 0. , 3.5, 4.5, 5. ])
```

- La función `pd.isna(a)` devuelve una Serie booleana indicando `True` donde a tiene valores `NaN` y `False` en caso contrario.

In [137…
```
pd.isna(a)
```

Out[137…
```
f     True
e    False
d    False
c    False
b    False
a     True
dtype: bool
```

- El método `np.where(condition, x, y)` selecciona elementos de `x` cuando condition es True y selecciona elementos de `y` cuando (o si la) condition es False.

En este caso, dado que `pd.isna(a)` es `True` en algunos índices, se selecciona el valor de `b`, y donde `pd.isna(a)` sea `False`, se selecciona el valor de `a`.

Aquí, siempre que los valores en a sean nulos, se seleccionan los valores de b, de lo contrario se seleccionan los valores no nulos de a. El uso de `numpy.where` no comprueba si las etiquetas de índice están alineadas o no (y ni siquiera requiere que los objetos tengan la misma longitud), así que si quieres alinear valores por índice, utiliza el método `combine_first` de la serie:

In [139…
```
a.combine_first(b)
```

Out[139…
```
a    0.0
b    4.5
c    3.5
d    0.0
e    2.5
f    5.0
dtype: float64
```

Con DataFrames, `combine_first` hace lo mismo columna por columna, por lo que se puede pensar en ello como "parchear" (patching) los datos que faltan en el objeto de llamada con los datos del objeto que se le pasan:

In [140…
```
df1 = pd.DataFrame({"a": [1., np.nan, 5., np.nan],
                    "b": [np.nan, 2., np.nan, 6.],
                    "c": range(2, 18, 4)})
df1
```

Out[140…

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

In [141…
```
df2 = pd.DataFrame({"a": [5., 4., np.nan, 3., 7.],
                    "b": [np.nan, 3., 4., 6., 8.]})
df2
```

Out[141…

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

In [142…
```
df1.combine_first(df2)
```

Out[142…

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

La salida de `combine_first` con objetos DataFrame tendrá la unión de todos los nombres de columna.

Ejercicio 10. Aplicación de `combine_first` para completar datos de prueba con datos de entrenamiento:

Supongamos que tenemos datos de entrenamiento y de prueba sobre ventas de productos, pero algunos valores están faltantes en el conjunto de prueba. Queremos combinar ambos conjuntos para rellenar los valores faltantes en el conjunto de prueba.

Datos

In [143…
```
# Datos de entrenamiento
train_data = {
    'Producto': ['A', 'B', 'C', 'D'],
    'Ventas': [100, 150, 200, 250]
}

# Datos de prueba con valores faltantes
test_data = {
    'Producto': ['A', 'B', 'C', 'E'],
    'Ventas': [np.nan, 160, np.nan, 180]
}
```

Pregunta 10.1: Crear los dataframes

In [ ]:
```

```

Pregunta 10.2: Combine los datos y guarde los resultados como df_test_combined

In [ ]:

Ejercicio 11: Actualización de Datos

Supongamos que tenemos datos antiguos y nuevos sobre el inventario de productos y queremos actualizar los datos antiguos con la nueva información.

Datos:

In [144…
```
old_inventory = {
    'Producto': ['Laptop', 'Smartphone', 'Tablet'],
    'Cantidad': [10, 20, 15],
    'Precio': [1000, 800, 300]
}

# Datos nuevos del inventario con actualizaciones
new_inventory = {
    'Producto': ['Laptop', 'Smartphone', 'Tablet'],
    'Cantidad': [12, 18, 17],
    'Precio': [950, 850, 310]
}
```

Pregunta 11.1: Crear los dataframes

In [ ]:

Pregunta 11.2: Actualizar usando combine_first

In [ ]:

3.3 Reshaping con `stack` y `unstack()`

Existen varias operaciones básicas para reorganizar datos tabulares. Se denominan operaciones de `reshape` o `pivot`.

Remodelación (Reshaping) con indexación jerárquica

La indexación jerárquica proporciona una forma coherente de reorganizar los datos en un DataFrame. Existen dos acciones principales:

`stack`: Esto "gira" o pivota de las columnas de los datos a las filas.

`unstack`: Pivota de las filas a las columnas.

Se ilustrarán estas operaciones con una serie de ejemplos. Consideremos un pequeño DataFrame con arrays de cadenas como índices de fila y columna:

In [146…
```
data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                    index=pd.Index(["Ohio", "Colorado"], name="state"),
                    columns=pd.Index(["Uno", "Dos", "Tres"],
```

```
                    name="numero"))
data
```

Out[146…

state	numero	Uno	Dos	Tres
Ohio		0	1	2
Colorado		3	4	5

Utilizando el método `stack` en estos datos, las columnas pivotan en las filas, produciendo una Serie:

In [147…

```
result = data.stack()
result
```

Out[147…

```
state     numero
Ohio      Uno       0
          Dos       1
          Tres      2
Colorado  Uno       3
          Dos       4
          Tres      5
dtype: int64
```

A partir de una Serie indexada jerárquicamente, puede reorganizar los datos de nuevo en un DataFrame con `unstack()`:

In [148…

```
result.unstack()
```

Out[148…

state	numero	Uno	Dos	Tres
Ohio		0	1	2
Colorado		3	4	5

Por defecto, el nivel más interno está desapilado (igual que la pila). Puedes desapilar(unstack) un nivel diferente pasando un número o nombre de nivel:

In [149…

```
result.unstack(level=0)
```

Out[149…

numero	state	Ohio	Colorado
Uno		0	3
Dos		1	4
Tres		2	5

In [150…

```
result.unstack(level="state")
```

Out[150…

state	Ohio	Colorado
numero		
Uno	0	3
Dos	1	4
Tres	2	5

El desapilamiento puede introducir datos que faltan si no se encuentran todos los valores del nivel en cada subgrupo:

In [151…
```
s1 = pd.Series([0, 1, 2, 3], index=["a", "b", "c", "d"], dtype="Int64")
s1
```

Out[151…
```
a    0
b    1
c    2
d    3
dtype: Int64
```

In [152…
```
s2 = pd.Series([4, 5, 6], index=["c", "d", "e"], dtype="Int64")
s2
```

Out[152…
```
c    4
d    5
e    6
dtype: Int64
```

In [154…
```
data2 = pd.concat([s1, s2], keys=["one", "two"])
data2
```

Out[154…
```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64
```

Si deseara filtrar por defecto los datos que faltan podemos usar `unstack`, por lo que la operación es más fácilmente invertible:

In [155…
```
data2.unstack()
```

Out[155…

	a	b	c	d	e
one	0	1	2	3	<NA>
two	<NA>	<NA>	4	5	6

si quiero volver a la forma inicial:

In [156…
```
data2.unstack().stack()
```

Out[156…
```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64
```

In [158…
```
data2.unstack().stack(dropna=False)
# convierte las columnas de nuevo en un índice jerárquico, manteniendo los NaN.
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_6868\3936770077.py:1: FutureWarning:
The previous implementation of stack is deprecated and will be removed in a futur
e version of pandas. See the What's New notes for pandas 2.1.0 for details. Speci
fy future_stack=True to adopt the new implementation and silence this warning.
  data2.unstack().stack(dropna=False)
```

Out[158…
```
one  a       0
     b       1
     c       2
     d       3
     e    <NA>
two  a    <NA>
     b    <NA>
     c       4
     d       5
     e       6
dtype: Int64
```

Cuando se desapila (unstack) en un DataFrame, el nivel desapilado se convierte en el nivel más bajo del resultado:

In [159…
```
df = pd.DataFrame({"left": result, "right": result + 5},
                  columns=pd.Index(["left", "right"], name="side"))
df
```

Out[159…

	side	left	right
state	numero		
Ohio	Uno	0	5
	Dos	1	6
	Tres	2	7
Colorado	Uno	3	8
	Dos	4	9
	Tres	5	10

In [100…
```
df.unstack(level="state")
```

Out[100…

side	left		right	
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

Al igual que con `unstack`, al llamar a `stack` podemos indicar el nombre del eje a apilar:

df.unstack(level="state").stack(level="side")

Ejercicio 12. Uso de stack y unstack para hacer 'reshaping' sobre dataframes:

Datos

In [161…

```
data = {
    'Región': ['Norte', 'Norte', 'Norte', 'Sur', 'Sur', 'Sur', 'Este', 'Este', '
    'Producto': ['Laptop', 'Smartphone', 'Tablet', 'Laptop', 'Smartphone', 'Tabl
    'Q1': [150, 200, 50, 100, 80, 30, 120, 60, 90, 110, 130, 40],
    'Q2': [170, 210, 60, 110, 85, 35, 125, 65, 95, 115, 135, 45],
    'Q3': [160, 220, 55, 105, 82, 32, 122, 62, 92, 112, 132, 42],
    'Q4': [180, 230, 65, 115, 90, 40, 130, 70, 100, 120, 140, 50]
}
```

Pregunta 12.1: A partir de `data` crear un dataframe multi-indice (en fila) que tenga esta forma:

Región	Producto	Q1	Q2	Q3	Q4
Norte	Laptop	150	170	160	180
	Smartphone	200	210	220	230
	Tablet	50	60	55	65
Sur	Laptop	100	110	105	115
	Smartphone	80	85	82	90
	Tablet	30	35	32	40
Este	Laptop	120	125	122	130
	Smartphone	60	65	62	70
	Tablet	90	95	92	100
Oeste	Laptop	110	115	112	120
	Smartphone	130	135	132	140
	Tablet	40	45	42	50

Donde: `Región` y `Producto` tienen una jerarquía de índice de nivel 2

In [ ]:

Pregunta 12.2: Usar `unstack` para transformar el índice de nivel inferior (Producto) en columnas.

In [ ]:

Pregunta 12.3: Revertir el `unstack` anterior usando `stack`

In [ ]:

Ejercicio 13. Datos de transacciones bancarias

Datos:

In [164…

```
data = {
    'Sucursal': ['Norte', 'Norte', 'Norte', 'Norte', 'Sur', 'Sur', 'Sur', 'Sur',
    'Cuenta': ['Corriente', 'Ahorros', 'Crédito', 'Inversiones', 'Corriente', 'A
    'Año': [2021, 2021, 2021, 2021, 2021, 2021, 2021, 2021, 2021, 2021, 2021, 20
    'Q1': [50000, 150000, 30000, 70000, 45000, 120000, 25000, 65000, 52000, 1300
    'Q2': [52000, 155000, 31000, 71000, 46000, 125000, 26000, 66000, 53000, 1350
    'Q3': [51000, 160000, 32000, 72000, 47000, 130000, 27000, 67000, 54000, 1400
    'Q4': [53000, 165000, 33000, 73000, 48000, 135000, 28000, 68000, 55000, 1450
}
```

Pregunta 13.1: Crear un dataframe con índices jerarquicos (en filas) de nivel 3

In [ ]:

Pregunta 13.2: Usar `unstack` para transformar el índice de nivel inferior ('Año') en columnas

In [ ]:

Pregunta 13.3: Revertir el dataframe anterior a su forma original, usar `stack`

In [ ]:

Pasar del formato "largo" (Long) al "ancho (Wide)

Una forma habitual de almacenar múltiples series temporales (series de fechas) en bases de datos y archivos CSV es lo que a veces se denomina formato largo o apilado (`stacked format`). En este formato, los valores individuales se representan mediante una única fila en una tabla, en lugar de múltiples valores por fila.

Carguemos algunos datos de ejemplo y hagamos una pequeña limpieza de series temporales y otros datos:

In [194…
```
data = pd.read_csv("macrodata.csv")
data.head()
```

Out[194…

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	tbilrate
0	1959	1	2710.349	1707.4	286.898	470.045	1886.9	28.98	139.7	2.82
1	1959	2	2778.801	1733.7	310.859	481.301	1919.7	29.15	141.7	3.08
2	1959	3	2775.488	1751.8	289.226	491.260	1916.4	29.35	140.5	3.82
3	1959	4	2785.204	1753.7	299.356	484.052	1931.3	29.37	140.0	4.33
4	1960	1	2847.699	1770.5	331.722	462.199	1955.5	29.54	139.6	3.50

Esta dataset contiene una variedad de indicadores económicos clave que se utilizan para analizar el estado de la economía, incluyendo medidas de producción, consumo, inversión, gasto gubernamental, oferta monetaria, tasas de interés, desempleo, inflación y población. Estos datos son cruciales para economistas y analistas que estudian el rendimiento económico y realizan pronósticos económicos.

Columnas:

- `year` : El año en el que se registraron los datos.
- `quarter` : El trimestre del año en el que se registraron los datos. Los trimestres están numerados del 1 al 4.
- `realgdp` : Producto Interno Bruto (PIB) real. Es una medida del valor de todos los bienes y servicios producidos en una economía, ajustada por inflación.

- `realcons` : Consumo real. Representa el gasto total de los hogares en bienes y servicios, ajustado por inflación.
- `realinv` : Inversión real. Incluye la inversión en bienes de capital, como maquinaria y edificios, ajustada por inflación.
- `realgovt` : Gasto gubernamental real. El gasto total del gobierno en bienes y servicios, ajustado por inflación.
- `realdpi` : Ingreso personal disponible real. Es el ingreso total que tienen los individuos después de impuestos, ajustado por inflación.
- `cpi` : Índice de Precios al Consumidor (CPI). Una medida que examina el promedio ponderado de los precios de una canasta de bienes y servicios de consumo, y se utiliza para medir la inflación.
- `m1` : Oferta monetaria M1. Incluye el efectivo en circulación y los depósitos a la vista en bancos, una medida de la oferta de dinero en la economía.
- `tbilrate` : Tasa de interés de los Bonos del Tesoro a 3 meses. La tasa de rendimiento de los bonos del Tesoro de Estados Unidos con vencimiento a 3 meses.
- `unemp` : Tasa de desempleo. El porcentaje de la población activa que está desempleada y buscando empleo.
- `pop` : Población. La población total del país en millones.
- `infl` : Tasa de inflación. La tasa a la que suben los precios de los bienes y servicios, normalmente medida como un cambio porcentual anual en el CPI.
- `realint` : Tasa de interés real. La tasa de interés nominal ajustada por la inflación.

In [195…
```
data = data.loc[:, ["year", "quarter", "realgdp", "infl", "unemp"]]
```

In [196…
```
data.head()
```

Out[196…

	year	quarter	realgdp	infl	unemp
0	1959	1	2710.349	0.00	5.8
1	1959	2	2778.801	2.34	5.1
2	1959	3	2775.488	2.74	5.3
3	1959	4	2785.204	0.27	5.6
4	1960	1	2847.699	2.31	5.2

En primer lugar, se va utilizar `pandas.PeriodIndex` para representar intervalos de tiempo en lugar de puntos en el tiempo, lo veremos con más detalle en el tema de Series temporales. La finalidad combinar las columnas de `year` y `quarter` y establecer el índice para que consista en valores `datetime` al final de cada trimestre:

In [197…
```
periods = pd.PeriodIndex(year=data.pop("year"),
                         quarter=data.pop("quarter"),
                         name="date")
periods
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_6868\1359989538.py:1: FutureWarning:
Constructing PeriodIndex from fields is deprecated. Use PeriodIndex.from_fields i
nstead.
  periods = pd.PeriodIndex(year=data.pop("year"),
```

Out[197…
```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203)
```

- `data.pop("year")` y `data.pop("quarter")`:
  - `data.pop("year")`: Esta función elimina la columna "year" del DataFrame data y devuelve los valores de esa columna.
  - `data.pop("quarter")`: Similar a pop("year"), esta función elimina la columna "quarter" del DataFrame data y devuelve los valores de esa columna.
  - Al usar `pop()`, las columnas "year" y "quarter" se eliminan del DataFrame original data, y sus valores se utilizan para crear el `PeriodIndex`.

In [198…
```
data.index = periods.to_timestamp("D")
```

El método `to_timestamp()` convierte un `PeriodIndex` a un `DatetimeIndex`.

- `"D"` es un argumento opcional que especifica la frecuencia del DatetimeIndex resultante. `"D"` significa que queremos que la frecuencia sea diaria.

En el caso de trimestres, `to_timestamp("D")` toma la fecha de inicio del trimestre como el timestamp.

In [199…
```
data.head()
```

Out[199…

date	realgdp	infl	unemp
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Aquí se ha utilizado el método `pop` en el DataFrame, que devuelve una columna al mismo tiempo que la elimina del DataFrame.

A continuación, se selecciona un subconjunto de columnas y se le da el nombre `"item"` al índice de columnas:

In [200…
```
data = data.reindex(columns=["realgdp", "infl", "unemp"])
data.columns.name = "item"
data.head()
```

Out[200…

item	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

Por último, se ha hecho un reshape con `stack`, convierte los nuevos niveles de índice en columnas con `reset_index` y, por último, se le da el nombre `"value"` a la columna que contiene los valores de los datos:

In [201…
```
long_data = (data.stack()
             .reset_index()
             .rename(columns={0: "value"}))
```

- `data.stack()`: Apila las columnas en una sola columna, transformando el DataFrame a un formato largo.
- `reset_index()`: Restablece el índice, convirtiendo los índices apilados en columnas.
- `rename(columns={0: "value"})`: Renombra la columna resultante de los valores apilados a `"value"`.

Ahora, `ldata` se ve así:

In [202…
```
long_data[:10]
```

Out[202…

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340
5	1959-04-01	unemp	5.100
6	1959-07-01	realgdp	2775.488
7	1959-07-01	infl	2.740
8	1959-07-01	unemp	5.300
9	1959-10-01	realgdp	2785.204

En este formato denominado largo para series temporales múltiples, cada fila de la tabla representa una única observación.

Los datos se almacenan con frecuencia de esta forma en bases de datos relacionales SQL, ya que un esquema fijo (nombres de columna y tipos de datos) permite que el número de valores distintos en la columna de `ítem` cambie a medida que se añaden datos a la tabla.

En el ejemplo anterior, `date` y `item` suelen ser las claves primarias (en el lenguaje de las bases de datos relacionales), lo que ofrece integridad relacional y facilita las uniones (joins). En algunos casos, puede ser más difícil trabajar con los datos en este formato; es posible que prefiera tener un DataFrame que contenga una columna por cada valor de elemento distinto indexado por marcas de tiempo (timestamps) en la columna de fecha(date). El método `pivot` de DataFrame realiza exactamente esta transformación:

In [203…

```
pivoted = long_data.pivot(index="date", columns="item",
                          values="value")
```

`long_data.pivot()` :
- El método `.pivot()` reorganiza el DataFrame, utilizando valores únicos de una columna como columnas nuevas y otra columna como índices.
- `index="date"` : Especifica que la columna "date" del DataFrame `long_data` se usará como índice del DataFrame resultante.
- `columns="item"` : Especifica que los valores únicos de la columna "item" (que contiene "realgdp", "infl", "unemp") se convertirán en nuevas columnas en el DataFrame resultante.
- `values="value"` : Especifica que los valores en la columna "value" del DataFrame long_data se usarán como valores en el DataFrame resultante.

In [204…

```
pivoted.head()
```

Out[204...

item	infl	realgdp	unemp
date			
1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2

Los dos primeros valores pasados son las columnas que se utilizarán, respectivamente, como índice de fila y de columna, y finalmente una columna de valor opcional para rellenar el DataFrame. Supongamos que tiene dos columnas de valores que desea remodelar (reshape) simultáneamente:

In [205...
```
long_data["value2"] = np.random.standard_normal(len(long_data))
long_data[:10]
```

Out[205...

	date	item	value	value2
0	1959-01-01	realgdp	2710.349	0.438166
1	1959-01-01	infl	0.000	0.329662
2	1959-01-01	unemp	5.800	-0.907067
3	1959-04-01	realgdp	2778.801	1.458386
4	1959-04-01	infl	2.340	-0.360448
5	1959-04-01	unemp	5.100	-1.378816
6	1959-07-01	realgdp	2775.488	0.712841
7	1959-07-01	infl	2.740	0.270545
8	1959-07-01	unemp	5.300	-0.693876
9	1959-10-01	realgdp	2785.204	-0.885817

In [206...
```
long_data.index.name = None
# Establece el nombre del índice a None.
# Esto elimina cualquier nombre asignado
# previamente al índice del DataFrame long_data.
```

Omitiendo el último argumento, se obtiene un DataFrame con columnas jerárquicas:

In [207...
```
pivoted = long_data.pivot(index="date", columns="item")
```

`pivot(index="date", columns="item")` : Reorganiza el DataFrame de modo que:
- `index="date"` : La columna "date" se convierte en el índice.
- `columns="item"` : Los valores únicos de la columna "item" se convierten en las nuevas columnas del DataFrame.

In [208…
```
pivoted.head()
```

Out[208…

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-01-01	0.00	2710.349	5.8	0.329662	0.438166	-0.907067
1959-04-01	2.34	2778.801	5.1	-0.360448	1.458386	-1.378816
1959-07-01	2.74	2775.488	5.3	0.270545	0.712841	-0.693876
1959-10-01	0.27	2785.204	5.6	-0.403006	-0.885817	0.179232
1960-01-01	2.31	2847.699	5.2	0.870955	0.488279	2.093177

In [211…
```
pivoted["value"].head()
```

Out[211…

item	infl	realgdp	unemp
date			
1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2

Tenga en cuenta que pivotar es equivalente a crear un índice jerárquico utilizando `set_index` seguido de una llamada a `unstack`:

In [212…
```
unstacked = long_data.set_index(["date", "item"]).unstack(level="item")
unstacked.head()
```

Out[212…

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-01-01	0.00	2710.349	5.8	0.329662	0.438166	-0.907067
1959-04-01	2.34	2778.801	5.1	-0.360448	1.458386	-1.378816
1959-07-01	2.74	2775.488	5.3	0.270545	0.712841	-0.693876
1959-10-01	0.27	2785.204	5.6	-0.403006	-0.885817	0.179232
1960-01-01	2.31	2847.699	5.2	0.870955	0.488279	2.093177

Ejercicio 14: Datos de ventas mensuales

In [223…
```
data = {
    'Tienda': ['Tienda_A', 'Tienda_A', 'Tienda_A', 'Tienda_A', 'Tienda_A', 'Tien
               'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tien
               'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tien
               'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tien
               'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tien
    'Producto': ['Manzana', 'Manzana', 'Manzana', 'Pera', 'Pera', 'Pera',
                 'Manzana', 'Manzana', 'Manzana', 'Pera', 'Pera', 'Pera',
                 'Lechuga', 'Lechuga', 'Lechuga', 'Tomate', 'Tomate', 'Tomate',
                 'Lechuga', 'Lechuga', 'Lechuga', 'Tomate', 'Tomate', 'Tomate',
                 'Frijoles', 'Frijoles', 'Frijoles', 'Zanahoria', 'Zanahoria', '
    'Año': [2020, 2020, 2020, 2020, 2020, 2020,
            2021, 2021, 2021, 2021, 2021, 2021,
            2020, 2020, 2020, 2020, 2020, 2020,
            2021, 2021, 2021, 2021, 2021, 2021,
            2020, 2020, 2020, 2020, 2020, 2020],
    'Mes': ['Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo'],
    'Ventas': [200, 220, 230, 150, 170, 180,
               180, 210, 220, 160, 190, 200,
               190, 200, 210, 170, 180, 190,
               210, 220, 230, 200, 210, 220,
               200, 210, 220, 210, 220, 230]
}
```

Pregunta 14.1: Crear el dataframe con la información de arriba

In [ ]:

Pregunta 14.2: Pivotar el DataFrame para que los meses sean columnas.

In [ ]:

Pregunta 14.3: Usar `unstack` para convertir el nivel Producto en columnas.

In [ ]:

Pregunta 14.4: Usar `stack` para revertir la transformación y volver al formato anterior.

In [ ]:

Pregunta 14.5: Crear un PeriodIndex y Convertirlo a Timestamps

- Se debe crear un índice de períodos basado en Año y Mes, y luego lo convertimos a timestamps.

In [ ]:

Pasar (Pivoting) del formato "ancho" (Wide) al "largo" (Long)

Una operación inversa al pivote para DataFrames es `pandas.melt`. En lugar de transformar una columna en muchas en un nuevo DataFrame, fusiona(merges) múltiples columnas en una, produciendo un DataFrame más largo que el de entrada. Veamos un ejemplo:

In [215…
```
df = pd.DataFrame({"key": ["foo", "bar", "baz"],
                   "A": [1, 2, 3],
                   "B": [4, 5, 6],
                   "C": [7, 8, 9]})
df
```

Out[215…

	key	A	B	C
0	foo	1	4	7
1	bar	2	5	8
2	baz	3	6	9

La columna "key" puede ser un indicador de grupo, y las otras columnas son valores de datos. Al utilizar `pandas.melt`, debemos indicar qué columnas (si las hay) son indicadores de grupo. Usemos aquí `"key"` como único indicador de grupo:

In [216…
```
melted = pd.melt(df, id_vars="key")
melted
```

Out[216…

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

Usando `pivot`, podemos volver (reshape) al diseño original:

In [217…
```
reshaped = melted.pivot(index="key", columns="variable",
                        values="value")
reshaped
```

Out[217…

variable	A	B	C
key			
bar	2	5	8
baz	3	6	9
foo	1	4	7

Dado que el resultado de `pivot` crea un índice a partir de la columna utilizada como etiquetas de fila, es posible que deseemos utilizar `reset_index` para volver a mover los datos a una columna:

In [218…
```
reshaped.reset_index()
```

Out[218…

	variable	key	A	B	C
0		bar	2	5	8
1		baz	3	6	9
2		foo	1	4	7

También puede especificar un subconjunto de columnas para utilizarlas como columnas de valores:

In [219…
```
pd.melt(df, id_vars="key", value_vars=["A", "B"])
```

Out[219…

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6

`pandas.melt` también se puede utilizar sin ningún identificador de grupo:

In [220…
```
pd.melt(df, value_vars=["A", "B", "C"])
```

Out[220…

	variable	value
0	A	1
1	A	2
2	A	3
3	B	4
4	B	5
5	B	6
6	C	7
7	C	8
8	C	9

In [221…

```
pd.melt(df, value_vars=["key", "A", "B"])
```

Out[221…

	variable	value
0	key	foo
1	key	bar
2	key	baz
3	A	1
4	A	2
5	A	3
6	B	4
7	B	5
8	B	6

Ejercicio 15: Datos de ventas. Uso de melt

In [224…

```
data = {
    'Tienda': ['Tienda_A', 'Tienda_A', 'Tienda_A', 'Tienda_A', 'Tienda_A', 'Tien
               'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tienda_B', 'Tien
               'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tienda_C', 'Tien
               'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tienda_D', 'Tien
               'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tienda_E', 'Tien
    'Producto': ['Ropa', 'Ropa', 'Ropa', 'Calzado', 'Calzado', 'Calzado',
                 'Ropa', 'Ropa', 'Ropa', 'Calzado', 'Calzado', 'Calzado',
                 'Lencería', 'Lencería', 'Lencería', 'Calzado', 'Calzado', 'Calz
                 'Lencería', 'Lencería', 'Lencería', 'Calzado', 'Calzado', 'Calz
                 'Ropa', 'Ropa', 'Ropa', 'Lencería', 'Lencería', 'Lencería'],
    'Año': [2020, 2020, 2020, 2020, 2020, 2020,
            2021, 2021, 2021, 2021, 2021, 2021,
            2020, 2020, 2020, 2020, 2020, 2020,
            2021, 2021, 2021, 2021, 2021, 2021,
            2020, 2020, 2020, 2020, 2020, 2020],
    'Mes': ['Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
```

```
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo',
            'Enero', 'Febrero', 'Marzo', 'Enero', 'Febrero', 'Marzo'],
    'Ventas': [200, 220, 230, 150, 170, 180,
               180, 210, 220, 160, 190, 200,
               190, 200, 210, 170, 180, 190,
               210, 220, 230, 200, 210, 220,
               200, 210, 220, 210, 220, 230],
    'Hora': ['10:00', '11:00', '12:00', '13:00', '14:00', '15:00',
             '16:00', '17:00', '18:00', '19:00', '20:00', '21:00',
             '09:00', '10:00', '11:00', '12:00', '13:00', '14:00',
             '15:00', '16:00', '17:00', '18:00', '19:00', '20:00',
             '10:00', '11:00', '12:00', '13:00', '14:00', '15:00']
}
```

Pregunta 15.1: Crear el Dataframe

In [ ]:

Pregunta 15.2: Crear un índice jerárquico utilizando las columnas Tienda, Producto y Año. Luego pivotamos el DataFrame para que los meses sean columnas.

In [ ]:

Pregunta 15.3: Usar `stack` para convertir las columnas de meses en un índice y luego `unstack` para convertir el nivel `Producto` en columnas.

In [ ]:

Pregunta 15.4: Reindexar el DataFrame para asegurarnos de que todas las combinaciones posibles de tiendas, productos y años estén presentes, rellenando con valores NaN donde no haya datos.

In [ ]:

Pregunta 15.5: Utilizar `melt` para transformar el DataFrame de vuelta a un formato largo.

In [ ]:

Pregunta 15.6: Crear un índice de períodos basado en Año, Mes y Hora, luego convertir a timestamps y ,finalmente, renombrar las columnas.

In [ ]: