

Pandas

¿Qué es Pandas?

Pandas es una biblioteca de código abierto para el lenguaje de programación Python, especializada en el manejo y análisis de datos. Es una herramienta fundamental para cualquier persona que trabaje con conjuntos de datos en Python.



Características principales de Pandas:

- Estructuras de datos potentes: Define nuevas estructuras de datos como DataFrames y Series, basadas en los arrays de NumPy, pero con funcionalidades más avanzadas para el manejo de datos tabulares y series temporales.
- Manipulación flexible de datos: Permite leer y escribir datos de diversos formatos comunes, como CSV, Excel, bases de datos SQL y archivos JSON.
- Operaciones de análisis avanzadas: Ofrece una amplia gama de funciones para filtrar, ordenar, agrupar, agregar, combinar y transformar datos de manera eficiente.
- Análisis de series temporales: Brinda herramientas específicas para trabajar con datos de series temporales, como el manejo de fechas, índices de tiempo y 'resampling'.
- Visualización de datos: Integra funciones básicas para la creación de gráficos y visualizaciones de datos.

Se utiliza para:

- Cargar y limpiar datos: Importar datos de diversas fuentes, eliminar valores faltantes y corregir errores.
- Manipular y transformar datos: Reordenar, filtrar, agrupar y agregar datos según diferentes criterios.

- Analizar datos: Realizar cálculos estadísticos, identificar patrones y tendencias en los datos.
- Visualizar datos: Crear gráficos y visualizaciones para comunicar los resultados del análisis.

In [3]: `#!pip install numpy`

Requirement already satisfied: numpy in c:\users\juanjuan\anaconda3\envs\pandas_2024\lib\site-packages (2.0.0)

In [2]: `#!pip install pandas`

Collecting pandas

Using cached pandas-2.2.2-cp311-cp311-win_amd64.whl.metadata (19 kB)

Collecting numpy>=1.23.2 (from pandas)

Downloading numpy-2.0.0-cp311-cp311-win_amd64.whl.metadata (60 kB)

----- 0.0/60.9 kB ? eta -:-:--

----- 30.7/60.9 kB 640.0 kB/s eta 0:00:01

----- 60.9/60.9 kB 804.1 kB/s eta 0:00:00

Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\juanjuan\anaconda3\envs\pandas_2024\lib\site-packages (from pandas) (2.9.0.post0)

Requirement already satisfied: pytz>=2020.1 in c:\users\juanjuan\anaconda3\envs\pandas_2024\lib\site-packages (from pandas) (2024.1)

Collecting tzdata>=2022.7 (from pandas)

Using cached tzdata-2024.1-py2.py3-none-any.whl.metadata (1.4 kB)

Requirement already satisfied: six>=1.5 in c:\users\juanjuan\anaconda3\envs\pandas_2024\lib\site-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)

Using cached pandas-2.2.2-cp311-cp311-win_amd64.whl (11.6 MB)

Downloading numpy-2.0.0-cp311-cp311-win_amd64.whl (16.5 MB)

----- 0.0/16.5 MB ? eta -:-:--

----- 0.4/16.5 MB 8.5 MB/s eta 0:00:02

----- 1.0/16.5 MB 12.7 MB/s eta 0:00:02

----- 1.9/16.5 MB 13.6 MB/s eta 0:00:02

----- 3.0/16.5 MB 17.1 MB/s eta 0:00:01

----- 4.7/16.5 MB 21.3 MB/s eta 0:00:01

----- 5.6/16.5 MB 22.4 MB/s eta 0:00:01

----- 7.5/16.5 MB 25.2 MB/s eta 0:00:01

----- 9.9/16.5 MB 28.8 MB/s eta 0:00:01

----- 12.6/16.5 MB 38.6 MB/s eta 0:00:01

----- 15.1/16.5 MB 46.7 MB/s eta 0:00:01

----- 16.5/16.5 MB 54.4 MB/s eta 0:00:01

----- 16.5/16.5 MB 46.7 MB/s eta 0:00:00

Using cached tzdata-2024.1-py2.py3-none-any.whl (345 kB)

Installing collected packages: tzdata, numpy, pandas

Successfully installed numpy-2.0.0 pandas-2.2.2 tzdata-2024.1

In [4]: `import numpy as np`
`import pandas as pd`

In [5]: `psg_players = pd.Series(['Navas', 'Mbappe', 'Neymar', 'Messi'], index=[1,7,10,30])`

In [9]: `psg_players`

```
Out[9]: 1      Navas
        7      Mbappe
        10     Neymar
        30     Messi
        dtype: object
```

Pandas, al crear una Series si no se la dan índices los asigna de forma automática:

```
In [10]: ingredientes = pd.Series(['Jamón', 'Aceitunas', 'Pan', 'Queso'])
```

```
In [11]: ingredientes
```

```
Out[11]: 0      Jamón
        1  Aceitunas
        2        Pan
        3      Queso
        dtype: object
```

Vamos a usar esta vez diccionarios

```
In [12]: dict = {1: 'Navas', 7: 'Mbappe', 10: 'Neymar', 30: 'Messi'}
        pd.Series(dict)
```

```
Out[12]: 1      Navas
        7      Mbappe
        10     Neymar
        30     Messi
        dtype: object
```

¿Que pasa si quiero definir un array con mas valores?

```
In [13]: dict_1 = {'Jugador': ['Navas', 'Mbappe', 'Neymar', 'Messi'],
                  'Altura': [183.0, 170.0, 185.0, 165.0],
                  'Goles': [2, 150, 180, 200]}
```

```
In [14]: pd.DataFrame(dict_1, index=[1,7,10,30])
```

```
Out[14]:
```

	Jugador	Altura	Goles
1	Navas	183.0	2
7	Mbappe	170.0	150
10	Neymar	185.0	180
30	Messi	165.0	200

Sin definir índices

```
In [38]: pd.DataFrame(dict_1)
```

Out[38]:

	Jugador	Altura	Goles
0	Navas	183.0	2
1	Mbappe	170.0	150
2	Neymar	185.0	180
3	Messi	165.0	200

```
In [39]: df_Players = pd.DataFrame(dict_1)
```

```
In [40]: df_Players
```

Out[40]:

	Jugador	Altura	Goles
0	Navas	183.0	2
1	Mbappe	170.0	150
2	Neymar	185.0	180
3	Messi	165.0	200

```
In [62]: df_Players.columns
```

Out[62]: Index(['Jugador', 'Altura', 'Goles'], dtype='object')

```
In [63]: df_Players.index
```

Out[63]: RangeIndex(start=0, stop=4, step=1)

```
In [64]: df_Players
```

Out[64]:

	Jugador	Altura	Goles
0	Navas	183.0	2
1	Mbappe	170.0	150
2	Neymar	185.0	180
3	Messi	165.0	200

Si tienes datos contenidos en un diccionario de Python, puedes crear una **Series** a partir de ellos pasándole el diccionario:

```
In [17]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}  
obj3 = pd.Series(sdata)  
obj3
```

```
Out[17]: Ohio      35000
         Texas     71000
         Oregon    16000
         Utah      5000
         dtype: int64
```

Una `Series` puede convertirse de nuevo en un diccionario con su método `to_dict` :

```
In [66]: obj3.to_dict()
```

```
Out[66]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

Cuando sólo se pasa un diccionario, el índice de la `Series` resultante respetará el orden de las claves según el método `keys` del diccionario, que depende del orden de inserción de las claves.

Puede anular esto pasando un índice con las claves del diccionario en el orden en que desea que aparezcan en la `Series` resultante:

```
In [16]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}
         states = ["California", "Ohio", "Oregon", "Texas"]
         obj4 = pd.Series(sdata, index=states)
         obj4
```

```
Out[16]: California      NaN
         Ohio            35000.0
         Oregon          16000.0
         Texas           71000.0
         dtype: float64
```

Aquí, tres valores encontrados en `sdata` se colocaron en los lugares apropiados, pero como no se encontró ningún valor para "California", aparece como `NaN` (Not a Number), que se considera en pandas para marcar valores perdidos o NA. Como "Utah" no se incluyó en estados, se excluye del objeto resultante.

Utilizaremos los términos "missing", "NA" (Not Available) o "null" indistintamente para referirnos a los datos que faltan. Las funciones `isna` y `notna` de pandas deben utilizarse para detectar datos omitidos:

```
In [68]: pd.isna(obj4)
```

```
Out[68]: California    True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool
```

```
In [69]: pd.notna(obj4)
```

```
Out[69]: California    False
         Ohio          True
         Oregon        True
         Texas         True
         dtype: bool
```

Series también los tiene como métodos de instancia:

```
In [41]: obj4.isna()
```

```
Out[41]: state
         California    True
         Ohio          False
         Oregon        False
         Texas         False
         Name: population, dtype: bool
```

Mas detalles sobre limpieza y depuración se verá mas adelante.

Una característica de Series útil para muchas aplicaciones es que alinea automáticamente por etiqueta de índice en operaciones aritméticas:

```
In [42]: obj3
```

```
Out[42]: Ohio          35000
         Texas         71000
         Oregon        16000
         Utah           5000
         dtype: int64
```

```
In [43]: obj4
```

```
Out[43]: state
         California    NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         Name: population, dtype: float64
```

```
In [44]: obj3 + obj4
```

```
Out[44]: California      NaN
Ohio      70000.0
Oregon    32000.0
Texas     142000.0
Utah      NaN
dtype: float64
```

Tanto el propio objeto `Series` como su índice tienen un atributo `name`, que se integra con otras áreas de funcionalidad de pandas:

```
In [45]: obj4.name = "population"
```

```
In [46]: obj4.index.name = "state"
obj4
```

```
Out[46]: state
California      NaN
Ohio      35000.0
Oregon    16000.0
Texas     71000.0
Name: population, dtype: float64
```

El índice de una `Series` puede modificarse "in situ" mediante asignación:

Definamos el siguiente obj:

```
In [47]: obj = pd.Series([4, 7, -5, 3])
obj
```

```
Out[47]: 0    4
1    7
2   -5
3    3
dtype: int64
```

Asignamos "in situ" los índices:

```
In [48]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]
obj
```

```
Out[48]: Bob      4
Steve    7
Jeff     -5
Ryan     3
dtype: int64
```

DataFrame

Un DataFrame representa una tabla rectangular de datos y contiene una colección ordenada y nombrada de columnas, cada una de las cuales puede ser un tipo de valor diferente (numérico, cadena, booleano, etc.). El DataFrame tiene tanto un índice de fila como de columna; puede considerarse como un diccionario de Series que comparten el mismo índice. Hay muchas maneras de construir un DataFrame, aunque una de las más comunes es a partir de un diccionario de listas de igual longitud o arrays de NumPy:

```
In [49]: data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
                "year": [2000, 2001, 2002, 2001, 2002, 2003],
                "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

El DataFrame resultante tendrá su índice asignado automáticamente, como con Series, y las columnas se colocan según el orden de las claves en los datos (que depende de su orden de inserción en el diccionario):

```
In [51]: frame
```

```
Out[51]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Para DataFrames grandes, el método `head` selecciona sólo las cinco primeras filas:

```
In [52]: frame.head()
```

```
Out[52]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

Del mismo modo, `tail` devuelve las cinco últimas filas:

```
In [29]: frame.tail()
```


Out[29]:

	state	year	pop
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Si especifica una secuencia de columnas, las columnas del DataFrame se ordenarán en ese orden:

```
In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])
```

Out[53]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

Si pasa una columna que no está contenida en el diccionario, aparecerá con valores ausentes en el resultado:

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])
frame2
```

Out[54]:

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

```
In [55]: frame2.columns
```

Out[55]: Index(['year', 'state', 'pop', 'debt'], dtype='object')

Una columna de un DataFrame puede recuperarse como una Serie mediante notación tipo diccionario o utilizando la notación de atributo `.` (dot notation):

```
In [56]: frame2["state"]
```

```
Out[56]: 0      Ohio
          1      Ohio
          2      Ohio
          3    Nevada
          4    Nevada
          5    Nevada
          Name: state, dtype: object
```

```
In [57]: frame2.year
```

```
Out[57]: 0      2000
          1      2001
          2      2002
          3      2001
          4      2002
          5      2003
          Name: year, dtype: int64
```

`frame2[column]` funciona para cualquier nombre de columna, pero `frame2.column` sólo funciona cuando el nombre de la columna es un nombre de variable Python válido y no entra en conflicto con ninguno de los nombres de método de DataFrame. Por ejemplo, si el nombre de una columna contiene espacios en blanco o símbolos que no sean guiones bajos, no se puede acceder a ella con el método de atributo dot.

Observe que las `Series` devueltas tienen el mismo índice que el DataFrame, y su atributo `name` se ha configurado adecuadamente.

Las filas también pueden recuperarse por posición o nombre con los atributos especiales `iloc` y `loc`.

```
In [58]: frame2
```

```
Out[58]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

```
In [59]: frame2.loc[1]
```

```
Out[59]: year      2001
         state     Ohio
         pop       1.7
         debt      NaN
         Name: 1, dtype: object
```

```
In [60]: frame2.iloc[2]
```

```
Out[60]: year      2002
         state     Ohio
         pop       3.6
         debt      NaN
         Name: 2, dtype: object
```

Las columnas pueden modificarse por asignación. Por ejemplo, a la columna `debt` vacía se le puede asignar un valor escalar o un array de valores:

```
In [61]: frame2["debt"] = 16.5
         frame2
```

```
Out[61]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	16.5
1	2001	Ohio	1.7	16.5
2	2002	Ohio	3.6	16.5
3	2001	Nevada	2.4	16.5
4	2002	Nevada	2.9	16.5
5	2003	Nevada	3.2	16.5

Cuando asigne listas o arrays a una columna, la longitud del valor debe coincidir con la longitud del DataFrame. Si asigna una `Series`, sus etiquetas se realinearán exactamente con el índice del DataFrame, insertando los valores que falten en cualquier valor del índice que no esté presente:

```
In [62]: val = pd.Series([-1.2, -1.5, -1.7], index=[2, 4, 5])
         frame2["debt"] = val
         frame2
```

```
Out[62]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	-1.2
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	-1.5
5	2003	Nevada	3.2	-1.7

Al asignar una columna que no existe se creará una columna nueva. La palabra clave `del` borrará columnas como con un diccionario. Como ejemplo, primero añado una nueva columna de valores booleanos donde la columna `state` es igual a `"Ohio"`:

```
In [91]: frame2["eastern"] = frame2["state"] == "Ohio"
         frame2
```

```
Out[91]:
```

	year	state	pop	debt	eastern
0	2000	Ohio	1.5	NaN	True
1	2001	Ohio	1.7	NaN	True
2	2002	Ohio	3.6	-1.2	True
3	2001	Nevada	2.4	NaN	False
4	2002	Nevada	2.9	-1.5	False
5	2003	Nevada	3.2	-1.7	False

El método `del` se puede utilizar para eliminar esta columna:

```
In [92]: del frame2["eastern"]
         frame2.columns
```

```
Out[92]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Cuidado.

La columna devuelta al indexar un DataFrame es una vista de los datos subyacentes, no una copia. Por lo tanto, cualquier modificación en la `Series` se reflejará en el DataFrame. La columna puede copiarse explícitamente con el método `copy` de la `Series`.

Otra forma común de datos es un diccionario anidado de diccionarios:

```
In [93]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6}, "Nevada": {2001: 2.4, 200
```

Si el diccionario anidado se pasa al DataFrame, pandas interpretará las claves externas del diccionario como las columnas, y las claves internas como los índices de fila:

```
In [94]: frame3 = pd.DataFrame(populations)
frame3
```

```
Out[94]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

Puede transponer el DataFrame (intercambiar filas y columnas) con una sintaxis similar a la de un array NumPy:

```
In [95]: frame3.T
```

```
Out[95]:
```

	2000	2001	2002
Ohio	1.5	1.7	3.6
Nevada	NaN	2.4	2.9

Cuidado

Tenga en cuenta que la transposición descarta los tipos de datos de columna si las columnas no tienen todas el mismo tipo de datos, por lo que transponer y luego volver a transponer puede perder la información del tipo anterior. En este caso, las columnas se convierten en matrices de objetos Python puros.

Las claves de los diccionarios internos se combinan para formar el índice del resultado. Esto no es cierto si se especifica un índice explícito:

```
In [96]: pd.DataFrame(populations, index=[2001, 2002, 2003])
```

```
Out[96]:
```


	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9
2003	NaN	NaN

Los diccionarios de `Series` reciben un tratamiento muy similar:

```
In [97]: pdata = {"Ohio": frame3["Ohio"][:-1], "Nevada": frame3["Nevada"][:2]}
pd.DataFrame(pdata)
```

```
Out[97]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

 No description has been provided for this image

Si el índice y las columnas de un DataFrame tienen definidos atributos de nombre, éstos también se mostrarán:

```
In [98]: frame3.index.name = "year"
frame3.columns.name = "state"
frame3
```

```
Out[98]:
```

state	Ohio	Nevada
year		
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

A diferencia de `Series`, `DataFrame` no tiene atributo `name`. El método `to_numpy` de `DataFrame` devuelve los datos contenidos en el `DataFrame` como un `ndarray` bidimensional:

```
In [99]: frame3.to_numpy()
```

```
Out[99]: array([[1.5, nan],
               [1.7, 2.4],
               [3.6, 2.9]])
```

Si las columnas del `DataFrame` son de diferentes tipos de datos, el tipo de datos del array devuelto se elegirá para acomodar todas las columnas:

```
In [100... frame2.to_numpy()
```

```
Out[100... array([[2000, 'Ohio', 1.5, nan],
               [2001, 'Ohio', 1.7, nan],
               [2002, 'Ohio', 3.6, -1.2],
               [2001, 'Nevada', 2.4, nan],
               [2002, 'Nevada', 2.9, -1.5],
               [2003, 'Nevada', 3.2, -1.7]], dtype=object)
```

Objetos índice

Los objetos `Index` de pandas son responsables de mantener las etiquetas de los ejes (incluyendo los nombres de las columnas de un `DataFrame`) y otros metadatos (como el

nombre o nombres de los ejes). Cualquier array u otra secuencia de etiquetas que se utilice al construir una `Series` o `DataFrame` se convierte internamente en un Índice:

```
In [101...] obj = pd.Series(np.arange(3), index=["a", "b", "c"])
obj
```

```
Out[101...] a    0
           b    1
           c    2
           dtype: int32
```

```
In [102...] index = obj.index
index
```

```
Out[102...] Index(['a', 'b', 'c'], dtype='object')
```

```
In [103...] index[1:]
```

```
Out[103...] Index(['b', 'c'], dtype='object')
```

Los objetos índice son inmutables y, por tanto, no pueden ser modificados por el usuario:

```
In [104...] index[1] = "d" # TypeError
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[104], line 1
----> 1 index[1] = "d"

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexes\base.py:5347, in Index.__setitem__(self, key, value)
    5345 @final
    5346 def __setitem__(self, key, value) -> None:
-> 5347     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

La inmutabilidad hace que sea más seguro compartir objetos índice entre estructuras de datos:

```
In [105...] labels = pd.Index(np.arange(3))
labels
```

```
Out[105...] Index([0, 1, 2], dtype='int32')
```

```
In [106...] obj2 = pd.Series([1.5, -2.5, 0], index=labels)
obj2
```

```
Out[106...] 0    1.5
           1   -2.5
           2    0.0
           dtype: float64
```

```
In [107... obj2.index is labels
```

```
Out[107... True
```

Además de ser similar a una matriz, un índice también se comporta como un conjunto (set) de tamaño fijo:

```
In [108... frame3
```

```
Out[108... state Ohio Nevada
```

year		
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

```
In [109... frame3.columns
```

```
Out[109... Index(['Ohio', 'Nevada'], dtype='object', name='state')
```

```
In [110... "Ohio" in frame3.columns
```

```
Out[110... True
```

```
In [111... 2003 in frame3.index
```


```
Out[111... False
```

A diferencia de los conjuntos de Python, un índice de pandas puede contener etiquetas duplicadas:

```
In [112... pd.Index(["foo", "foo", "bar", "bar"])
```

```
Out[112... Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Las selecciones con etiquetas duplicadas tomarán todas las apariciones de esa etiqueta. Cada Índice tiene una serie de métodos y propiedades para la lógica de conjuntos, que responden a otras preguntas habituales sobre los datos que contiene. Algunas de las más útiles se resumen en:

 No description has been provided for this image

Funciones esenciales

Esta sección es una guía a través de la mecánica fundamental de la interacción con los datos contenidos en una Serie o DataFrame.

Reindexación (Reindexing)

Un método importante en los objetos pandas es `reindex`, que significa crear un nuevo objeto con los valores reordenados para alinearlos con el nuevo índice. Consideremos un ejemplo:

```
In [113...] obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])
obj
```

```
Out[113...] d    4.5
            b    7.2
            a   -5.3
            c    3.6
            dtype: float64
```

Si se llama a `reindex` en esta `Series`, los datos se reordenan de acuerdo con el nuevo índice, introduciendo los valores que faltan si alguno de los valores del índice no estaba ya presente:

```
In [114...] obj2 = obj.reindex(["a", "b", "c", "d", "e"])
obj2
```

```
Out[114...] a   -5.3
            b    7.2
            c    3.6
            d    4.5
            e    NaN
            dtype: float64
```

En el caso de datos ordenados como series temporales, es posible que desee realizar alguna interpolación o relleno de valores al reindexar. La opción de método nos permite hacerlo, utilizando un método como `ffill`, que rellena los valores hacia delante:

```
In [115...] obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])
obj3
```

```
Out[115...] 0    blue
            2   purple
            4   yellow
            dtype: object
```

```
In [116...] obj3.reindex(np.arange(6), method="ffill")
```

```
Out[116...] 0    blue
            1    blue
            2   purple
            3   purple
            4   yellow
            5   yellow
            dtype: object
```

Con DataFrame, `reindex` puede alterar el índice (de filas), las columnas o ambos. Si sólo se le pasa una secuencia, reindexa las filas del resultado:

```
In [117...] frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=["a", "c", "d"], columns=["O", "T", "C"])
frame
```

```
Out[117...]
   Ohio  Texas  California
a      0      1          2
c      3      4          5
d      6      7          8
```

```
In [118...] frame2 = frame.reindex(index=["a", "b", "c", "d"])
frame2
```

```
Out[118...]
   Ohio  Texas  California
a    0.0    1.0          2.0
b    NaN    NaN          NaN
c    3.0    4.0          5.0
d    6.0    7.0          8.0
```

The columns can be reindexed with the `columns` keyword:

```
In [119...] states = ["Texas", "Utah", "California"]
frame.reindex(columns=states)
```

```
Out[119...]
   Texas  Utah  California
a      1   NaN          2
c      4   NaN          5
d      7   NaN          8
```

Como "Ohio" no estaba en los `states`, los datos de esa columna se eliminan del resultado. Otra forma de hacer `reindex` en un eje concreto es pasar las nuevas etiquetas de eje como argumento posicional y, a continuación, especificar el eje que se va a reindexar con la palabra clave `axis`:

```
In [120...] frame.reindex(states, axis="columns")
```

Out[120...

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Consulte la siguiente tabla para obtener más información sobre los argumentos para reindexar.



No description has been provided for this image

Como exploraremos más adelante en Selección en DataFrame con `loc` e `iloc`, también puede reindexar utilizando el operador `loc`, y muchos usuarios prefieren hacerlo siempre de esta manera. Esto sólo funciona si todas las nuevas etiquetas de índice ya existen en el DataFrame (mientras que `reindex` insertará los datos que falten para las nuevas etiquetas):

In [121...

```
frame.loc[["a", "d", "c"], ["California", "Texas"]]
```

Out[121...

	California	Texas
a	2	1
d	8	7
c	5	4

Eliminar entradas de un eje

Eliminar una o más entradas de un eje es sencillo si ya tienes una matriz o lista de índices sin esas entradas, ya que puedes utilizar el método `reindex` o la indexación basada en `.loc`. Como eso puede requerir un poco de lógica el método `drop` devolverá un nuevo objeto con el valor o valores indicados eliminados de un eje:

In [122...

```
obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])
obj
```

Out[122...

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [123...

```
new_obj = obj.drop("c")
new_obj
```

```
Out[123...  a    0.0
           b    1.0
           d    3.0
           e    4.0
           dtype: float64
```

```
In [124... obj.drop(["d", "c"])
```

```
Out[124...  a    0.0
           b    1.0
           e    4.0
           dtype: float64
```

Con DataFrame, los valores índice se pueden eliminar de cualquiera de los ejes. Para ilustrar esto, primero creamos un DataFrame de ejemplo:

```
In [125... data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                        index=["Ohio", "Colorado", "Utah", "New York"],
                        columns=["one", "two", "three", "four"])
data
```

```
Out[125...      one  two  three  four
Ohio      0    1     2    3
Colorado   4    5     6    7
Utah       8    9    10   11
New York  12   13    14   15
```

Si se llama a `drop` con una secuencia de etiquetas, se eliminarán los valores de las etiquetas de fila (eje 0):

```
In [126... data.drop(index=["Colorado", "Ohio"])
```

```
Out[126...      one  two  three  four
Utah       8    9    10   11
New York  12   13    14   15
```

Para eliminar las etiquetas de las columnas, utilice en su lugar la palabra clave `columns` :

```
In [127... data.drop(columns=["two"])
```

Out[127...

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

También puede eliminar valores de las columnas pasando `axis=1` (como NumPy) o `axis="columns"` :

In [128...

```
data.drop("two", axis=1)
```

Out[128...

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [129...

```
data.drop(["two", "four"], axis="columns")
```

Out[129...

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Indexación, selección y filtrado

La indexación de series (`obj[...]`) funciona de forma análoga a la indexación de matrices de NumPy, con la diferencia de que puedes utilizar los valores índice de la serie en lugar de sólo enteros. He aquí algunos ejemplos:

In [130...

```
obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])  
obj
```

Out[130...

```
a    0.0  
b    1.0  
c    2.0  
d    3.0  
dtype: float64
```

```
In [131... obj["b"]
```

```
Out[131... 1.0
```

```
In [132... obj[1]
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16212\2469632899.py:1: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`  
obj[1]
```

```
Out[132... 1.0
```

```
In [133... obj[2:4]
```

```
Out[133... c    2.0  
d    3.0  
dtype: float64
```

```
In [134... obj[["b", "a", "d"]]
```

```
Out[134... b    1.0  
a    0.0  
d    3.0  
dtype: float64
```

```
In [135... obj[[1, 3]]
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16212\2982346117.py:1: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`  
obj[[1, 3]]
```

```
Out[135... b    1.0  
d    3.0  
dtype: float64
```

```
In [136... obj[obj < 2]
```

```
Out[136... a    0.0  
b    1.0  
dtype: float64
```

Aunque puede seleccionar datos por etiqueta de esta manera, la forma preferida de seleccionar valores de índice es con el operador especial `loc` :

```
In [137... obj.loc[["b", "a", "d"]]
```

```
Out[137... b    1.0  
a    0.0  
d    3.0  
dtype: float64
```

La razón para preferir `loc` es el tratamiento diferente de los enteros cuando se indexa con `[]`. La indexación normal basada en `[]` tratará los enteros como etiquetas si el índice contiene enteros, por lo que el comportamiento difiere dependiendo del tipo de datos del índice. Por ejemplo:

```
In [138...] obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])
obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])
obj1
```

```
Out[138...] 2    1
            0    2
            1    3
dtype: int64
```

```
In [139...] obj2
```

```
Out[139...] a    1
            b    2
            c    3
dtype: int64
```

```
In [140...] obj1[[0, 1, 2]]
```

```
Out[140...] 0    2
            1    3
            2    1
dtype: int64
```

```
In [141...] obj2[[0, 1, 2]]
```

```
C:\Users\juanjuan\AppData\Local\Temp\ipykernel_16212\2599987575.py:1: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
obj2[[0, 1, 2]]
```

```
Out[141...] a    1
            b    2
            c    3
dtype: int64
```

Al utilizar `loc`, la expresión `obj.loc[[0, 1, 2]]` fallará cuando el índice no contenga enteros:

```
In [142...] obj2.loc[[0, 1]]
```

KeyError

Traceback (most recent call last)

Cell In[142], line 1

----> 1 obj2.loc[[0, 1]]

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexing.py:1153, in `_LocationIndexer.__getitem__(self, key)`

```
1150 axis = self.axis or 0
1152 maybe_callable = com.apply_if_callable(key, self.obj)
-> 1153 return self._getitem_axis(maybe_callable, axis=axis)
```

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexing.py:1382, in `_iLocIndexer._getitem_axis(self, key, axis)`

```
1379     if hasattr(key, "ndim") and key.ndim > 1:
1380         raise ValueError("Cannot index with multidimensional key")
-> 1382     return self._getitem_iterable(key, axis=axis)
1384 # nested tuple slicing
1385 if is_nested_tuple(key, labels):
```

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexing.py:1322, in `_iLocIndexer._getitem_iterable(self, key, axis)`

```
1319 self._validate_key(key, axis)
1321 # A collection of keys
-> 1322 keyarr, indexer = self._get_listlike_indexer(key, axis)
1323 return self.obj._reindex_with_indexers(
1324     {axis: [keyarr, indexer]}, copy=True, allow_dups=True
1325 )
```

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexing.py:1520, in `_iLocIndexer._get_listlike_indexer(self, key, axis)`

```
1517 ax = self.obj._get_axis(axis)
1518 axis_name = self.obj._get_axis_name(axis)
-> 1520 keyarr, indexer = ax._get_indexer_strict(key, axis_name)
1522 return keyarr, indexer
```

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexes\base.py:6114, in `Index._get_indexer_strict(self, key, axis_name)`

```
6111 else:
6112     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 6114 self._raise_if_missing(keyarr, indexer, axis_name)
6116 keyarr = self.take(indexer)
6117 if isinstance(key, Index):
6118     # GH 42790 - Preserve name from an Index
```

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexes\base.py:6175, in `Index._raise_if_missing(self, key, indexer, axis_name)`

```
6173     if use_interval_msg:
6174         key = list(key)
-> 6175     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
6177 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
6178 raise KeyError(f"{not_found} not in index")
```

KeyError: "None of [Index([0, 1], dtype='int32')] are in the [index]"

Dado que el operador `loc` indexa exclusivamente con etiquetas, existe también un operador `iloc` que indexa exclusivamente con enteros para trabajar de forma consistente tanto si el índice contiene enteros como si no:

```
In [ ]: obj1.iloc[[0, 1, 2]]
```

```
In [ ]: obj2.iloc[[0, 1, 2]]
```

Precaución: También se puede rebanar (slice) con etiquetas, pero funciona de forma diferente al rebanado(slicing) normal de Python, ya que el punto final es inclusivo:

```
In [ ]: obj2.loc["b":"c"]
```

La asignación de valores mediante estos métodos modifica la sección correspondiente de la `Series` :

```
In [143...] obj2.loc["b":"c"] = 5
obj2
```

```
Out[143...] a    1
           b    5
           c    5
           dtype: int64
```

La indexación en un `DataFrame` recupera una o más columnas, ya sea con un único valor o con una secuencia:

```
In [144...] data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                                index=["Ohio", "Colorado", "Utah", "New York"],
                                columns=["one", "two", "three", "four"])
data
```

```
Out[144...]      one  two  three  four
Ohio         0    1     2     3
Colorado     4    5     6     7
Utah         8    9    10    11
New York    12   13    14    15
```

```
In [145...] data["two"]
```

```
Out[145...] Ohio         1
           Colorado     5
           Utah         9
           New York    13
           Name: two, dtype: int32
```

```
In [146...] data[["three", "one"]]
```

Out[146...

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Este tipo de indexación tiene algunos casos especiales. El primero es el corte (slicing) o la selección de datos con una array booleano:

In [147...

data[:2]

Out[147...

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [148...

data[data["three"] > 5]

Out[148...

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

La sintaxis de selección de filas `data[:2]` se proporciona por comodidad. Al pasar un único elemento o una lista al operador `[]` se seleccionan columnas. Otro caso de uso es la indexación con un DataFrame booleano, como el producido por una comparación escalar. Considere un DataFrame con todos los valores booleanos producidos por comparación con un valor escalar:

In [149...

data < 5

Out[149...

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

Podemos utilizar este DataFrame para asignar el valor 0 a cada ubicación con el valor `True`, así:

```
In [150... data[data < 5] = 0
data
```

```
Out[150...
      one  two  three  four
Ohio    0    0     0     0
Colorado 0    5     6     7
Utah    8    9    10    11
New York 12   13    14    15
```

Selección en un DataFrame con `loc` e `iloc`

Al igual que `Series`, los `DataFrame` tiene atributos especiales `loc` e `iloc` para la indexación basada en etiquetas y en enteros, respectivamente. Como `DataFrame` es bidimensional, puede seleccionar un subconjunto de filas y columnas con notación tipo NumPy utilizando etiquetas de eje (`loc`) o enteros (`iloc`).

Como primer ejemplo, vamos a seleccionar una sola fila por etiqueta:

```
In [151... data
```

```
Out[151...
      one  two  three  four
Ohio    0    0     0     0
Colorado 0    5     6     7
Utah    8    9    10    11
New York 12   13    14    15
```

```
In [152... data.loc["Colorado"]
```

```
Out[152... one    0
two     5
three   6
four    7
Name: Colorado, dtype: int32
```

El resultado de seleccionar una sola fila es una `Series` con un índice que contiene las etiquetas de las columnas del `DataFrame`. Para seleccionar múltiples papeles, creando un nuevo `DataFrame`, se le pasa una secuencia de etiquetas:

```
In [153... data.loc[["Colorado", "New York"]]
```

```
Out[153...
```

	one	two	three	four
Colorado	0	5	6	7
New York	12	13	14	15

Puede combinar la selección de filas y columnas en `loc` separando las selecciones con una coma:

```
In [154... data.loc["Colorado", ["two", "three"]]
```

```
Out[154... two      5
three     6
Name: Colorado, dtype: int32
```

A continuación, realizaremos algunas selecciones similares con enteros utilizando `iloc`:

```
In [155... data.iloc[2]
```

```
Out[155... one      8
two      9
three    10
four     11
Name: Utah, dtype: int32
```

```
In [156... data.iloc[[2, 1]]
```

```
Out[156...
```

	one	two	three	four
Utah	8	9	10	11
Colorado	0	5	6	7

```
In [157... data.iloc[2, [3, 0, 1]]
```

```
Out[157... four     11
one        8
two        9
Name: Utah, dtype: int32
```

```
In [158... data.iloc[[1, 2], [3, 0, 1]]
```

```
Out[158...
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Ambas funciones de indexación funcionan con trozos (slices) además de con etiquetas individuales o listas de etiquetas:

```
In [159... data.loc[:"Utah", "two"]]
```

```
Out[159...] Ohio      0
           Colorado  5
           Utah      9
           Name: two, dtype: int32
```

```
In [160...] data.iloc[:, :3][data.three > 5]
```

```
Out[160...]
           one  two  three
Colorado    0    5     6
           Utah    8    9    10
New York   12   13    14
```

Los arrays Booleanos pueden ser usados con `loc` pero no `iloc` :

```
In [161...] data.loc[data.three >= 2]
```

```
Out[161...]
           one  two  three  four
Colorado    0    5     6     7
           Utah    8    9    10    11
New York   12   13    14    15
```

Hay muchas formas de seleccionar y reordenar los datos contenidos en un objeto pandas. Para un `DataFrame` , la siguiente tabla proporciona un breve resumen de muchas de ellas. Como se verá más adelante, hay una serie de opciones adicionales para trabajar con índices jerárquicos.



No description has been provided for this image

Errores en la indexación de números enteros

Trabajar con objetos pandas indexados por enteros puede ser un escollo para los nuevos usuarios ya que funcionan de forma diferente a las estructuras de datos incorporadas en Python como listas y tuplas. Por ejemplo, es posible que no espere que el siguiente código genere un error:

```
In [162...] ser = pd.Series(np.arange(3.))
           ser
```

```
Out[162...] 0    0.0
           1    1.0
           2    2.0
           dtype: float64
```

```
In [163...] ser[-1]
```

```
-----
ValueError                                Traceback (most recent call last)
File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexes\range.py:41
4, in RangeIndex.get_loc(self, key)
    413 try:
--> 414     return self._range.index(new_key)
    415 except ValueError as err:
```

ValueError: -1 is not in range

The above exception was the direct cause of the following exception:

```
KeyError                                Traceback (most recent call last)
Cell In[163], line 1
----> 1 ser[-1]

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\series.py:1040, in
Series.__getitem__(self, key)
    1037     return self._values[key]
    1039 elif key_is_scalar:
-> 1040     return self._get_value(key)
    1042 # Convert generator to list before going through hashable part
    1043 # (We will iterate through the generator there to check for slices)
    1044 if is_iterator(key):

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\series.py:1156, in
Series._get_value(self, label, takeable)
    1153     return self._values[label]
    1155 # Similar to Index.get_value, but we do not fall back to positional
-> 1156 loc = self.index.get_loc(label)
    1158 if is_integer(loc):
    1159     return self._values[loc]

File ~\anaconda3\envs\Testing_Rise\lib\site-packages\pandas\core\indexes\range.py:41
6, in RangeIndex.get_loc(self, key)
    414     return self._range.index(new_key)
    415     except ValueError as err:
--> 416         raise KeyError(key) from err
    417 if isinstance(key, Hashable):
    418     raise KeyError(key)
```

KeyError: -1

En este caso, pandas podría "recurrir" a la indexación por enteros, pero es difícil hacer esto en general sin introducir errores sutiles en el código de usuario. Aquí tenemos un índice que contiene 0, 1 y 2, pero pandas no quiere adivinar lo que quiere el usuario (indexación basada en etiquetas o basada en posiciones):

```
In [ ]: ser
```

En cambio, con un índice no entero, no existe tal ambigüedad:

```
In [ ]: ser2 = pd.Series(np.arange(3.), index=["a", "b", "c"])
ser2[-1]
```

Si tienes un índice de eje que contiene enteros, la selección de datos siempre estará orientada a etiquetas. Como se ha dicho anteriormente, si se utiliza `loc` (para etiquetas) o `iloc` (para enteros) se obtendrá exactamente lo que se desea:

```
In [ ]: ser.iloc[-1]
```

Por otra parte, el corte (slicing) con números enteros siempre está orientado a números enteros:

```
In [ ]: ser[:2]
```

Como consecuencia de estos errores, es mejor preferir siempre la indexación con `loc` e `iloc` para evitar ambigüedades.

Errores de la indexación encadenada

En la sección anterior vimos cómo se pueden hacer selecciones flexibles en un DataFrame utilizando `loc` e `iloc`. Estos atributos de indexación también se pueden utilizar para modificar objetos DataFrame "in situ", pero hacerlo requiere cierto cuidado.

Por ejemplo, en el ejemplo DataFrame anterior, podemos asignar a una columna o fila por etiqueta o posición entera:

```
In [164... data.loc[:, "one"] = 1
data
```

```
Out[164...

```

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	1	9	10	11
New York	1	13	14	15

```
In [165... data.iloc[2] = 5
data
```

Out[165...

	one	two	three	four
Ohio	1	0	0	0
Colorado	1	5	6	7
Utah	5	5	5	5
New York	1	13	14	15

In [166... `data.loc[data["four"] > 5] = 3`
`data`

Out[166...

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

Un error común para los nuevos usuarios de pandas es encadenar selecciones cuando se asignan de esta manera:

In [167... `data.loc[data.three == 5]["three"] = 6`

C:\Users\juanjuan\AppData\Local\Temp\ipykernel_16212\867481848.py:1: SettingWithCopyWarning:
 A value is trying to be set on a copy of a slice from a DataFrame.
 Try using `.loc[row_indexer,col_indexer] = value` instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
`data.loc[data.three == 5]["three"] = 6`

Dependiendo del contenido de los datos, esto puede imprimir una advertencia especial `SettingWithCopyWarning`, que le advierte de que está intentando modificar un valor temporal (el resultado no vacío de `data.loc[data.three == 5]`) en lugar del DataFrame original `data`, que podría ser lo que pretendía. En este caso, `data` no se ha modificado:

In [168... `data`


```
Out[168...
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	5	5
New York	3	3	3	3

En estos casos, la solución consiste en reescribir la asignación encadenada para utilizar una única operación `loc` :

```
In [169... data.loc[data.three == 5, "three"] = 6
data
```

```
Out[169...
```

	one	two	three	four
Ohio	1	0	0	0
Colorado	3	3	3	3
Utah	5	5	6	5
New York	3	3	3	3

Aritmética y alineación de datos

Pandas puede simplificar mucho el trabajo con objetos que tienen índices diferentes. Por ejemplo, al sumar objetos, si algún par de índices no es el mismo, el índice respectivo en el resultado será la unión de los pares de índices. Veamos un ejemplo:

```
In [170... s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a", "c", "d", "e"])
s1
```

```
Out[170... a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

```
In [171... s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
                 index=["a", "c", "e", "f", "g"])
s2
```

```
Out[171... a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [172...

s1 + s2

Out[172...

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

La alineación interna de los datos introduce valores perdidos en las ubicaciones de las etiquetas que no se solapan. Los valores perdidos se propagarán en los cálculos aritméticos posteriores.

En el caso de un `DataFrame`, la alineación se realiza tanto en las filas como en las columnas:

In [173...

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
                    index=["Ohio", "Texas", "Colorado"])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
                    index=["Utah", "Ohio", "Texas", "Oregon"])
```

In [174...

df1

Out[174...

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [175...

df2

Out[175...

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [176...

df1 + df2

Out[176...

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Como las columnas "c" y "e" no se encuentran en ambos objetos DataFrame, aparecen como ausentes en el resultado. Lo mismo ocurre con las filas con etiquetas que no son comunes a ambos objetos.

Si añade objetos DataFrame sin etiquetas de columna o fila en común, el resultado contendrá todos nulos:

In [177...

```
df1 = pd.DataFrame({"A": [1, 2]})
df2 = pd.DataFrame({"B": [3, 4]})
df1
```

Out[177...

	A
0	1
1	2

In [178...

```
df2
```

Out[178...

	B
0	3
1	4

In [179...

```
df1 + df2
```

Out[179...

	A	B
0	NaN	NaN
1	NaN	NaN

Métodos aritméticos con valores de relleno

En operaciones aritméticas entre objetos indexados de forma diferente, es posible que desee rellenar con un valor especial, como 0, cuando una etiqueta de eje se encuentra en un

objeto pero no en el otro. He aquí un ejemplo en el que establecemos un valor particular como NA (nulo) asignándole `np.nan` :

```
In [180...] df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),  
                      columns=list("abcd"))  
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),  
                  columns=list("abcde"))
```

```
In [181...] df1
```

```
Out[181...]
```

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

```
In [182...] df2
```

```
Out[182...]
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

```
In [183...] df1 + df2
```

```
Out[183...]
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Utilizando el método `add` en `df1`, se pasa `df2` y un argumento a `fill_value` , que sustituye el valor pasado por cualquier valor que falte en la operación:

```
In [184...] df1.add(df2, fill_value=0)
```

```
Out[184...
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Véase en la siguiente tabla un listado de los métodos `Series` y `DataFrame` para aritmética. Cada uno tiene una contrapartida, que empieza por la letra `r`, que tiene los argumentos invertidos. Por lo tanto, estas dos sentencias son equivalentes:

```
In [185... 1 / df1
```

```
Out[185...
```

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

```
In [186... df1.rdiv(1)
```

```
Out[186...
```


	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

Al reindexar una `Series` o un `DataFrame`, también puede especificar un valor de relleno (fill value) diferente:

```
In [187... df1.reindex(columns=df2.columns, fill_value=0)
```

```
Out[187...
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

 No description has been provided for this image

Operaciones entre `DataFrame` y `Series`

Al igual que con las matrices NumPy de diferentes dimensiones, también se define la aritmética entre DataFrame y Series. En primer lugar, como ejemplo, considere la diferencia entre una matriz bidimensional y una de sus filas:

```
In [188...] arr = np.arange(12.).reshape((3, 4))
arr
```

```
Out[188...] array([[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.]])
```

```
In [189...] arr[0]
```

```
Out[189...] array([0., 1., 2., 3.])
```

```
In [190...] arr - arr[0]
```

```
Out[190...] array([[0., 0., 0., 0.],
          [4., 4., 4., 4.],
          [8., 8., 8., 8.]])
```

Cuando restamos `arr[0]` de `arr`, la resta se realiza una vez por cada fila. Esto se denomina difusión y se explica con más detalle en lo que se refiere a las matrices generales de NumPy Avanzado. Las operaciones entre un DataFrame y una Serie son similares:

```
In [191...] frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                                columns=list("bde"),
                                index=["Utah", "Ohio", "Texas", "Oregon"])
series = frame.iloc[0]
```

```
In [192...] frame
```

```
Out[192...]
      b  d  e
Utah  0.0  1.0  2.0
Ohio  3.0  4.0  5.0
Texas  6.0  7.0  8.0
Oregon  9.0 10.0 11.0
```

```
In [193...] series
```

```
Out[193...] b    0.0
            d    1.0
            e    2.0
            Name: Utah, dtype: float64
```

Por defecto, la aritmética entre el `DataFrame` y la `Series` coincide con el índice de la `Series` en las columnas del `DataFrame`, difundiéndose por las filas:

In [194... frame - series

Out[194...

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Si no se encuentra un valor de índice ni en las columnas del `DataFrame` ni en el índice de la `Series`, los objetos se reindexarán para formar la unión:

In [195...
series2 = pd.Series(np.arange(3), index=["b", "e", "f"])
series2

Out[195...
b 0
e 1
f 2
dtype: int32

In [196... frame + series2

Out[196...

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

Si, en cambio, se desea trabajar sobre las columnas, coincidiendo en las filas, debe utilizar uno de los métodos aritméticos y especificar que coincida sobre el índice. Por ejemplo:

In [197...
series3 = frame["d"]
frame

Out[197...

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [198... series3

```
Out[198...] Utah      1.0
Ohio      4.0
Texas     7.0
Oregon    10.0
Name: d, dtype: float64
```

```
In [199...] frame.sub(series3, axis="index")
```

```
Out[199...]      b  d  e
Utah    -1.0  0.0  1.0
Ohio    -1.0  0.0  1.0
Texas   -1.0  0.0  1.0
Oregon  -1.0  0.0  1.0
```

El eje que se pasa es el eje sobre el que se va a realizar la comparación. En este caso nos referimos a coincidir en el índice de fila del `DataFrame` (`axis="index"`) y trabajará a través de las columnas.

Aplicación y asignación de funciones

Los `ufuncs` de NumPy (métodos de array por elementos) también funcionan con objetos pandas:

```
In [200...] frame = pd.DataFrame(np.random.standard_normal((4, 3)),
                                columns=list("bde"),
                                index=["Utah", "Ohio", "Texas", "Oregon"])
frame
```

```
Out[200...]      b      d      e
Utah   -1.243329  0.946710 -0.936494
Ohio    0.134194  0.947723 -0.667054
Texas  -1.347003 -1.220009 -0.698152
Oregon -0.790608  0.570923 -1.256654
```

```
In [201...] np.abs(frame)
```


Out[201...

	b	d	e
Utah	1.243329	0.946710	0.936494
Ohio	0.134194	0.947723	0.667054
Texas	1.347003	1.220009	0.698152
Oregon	0.790608	0.570923	1.256654

Otra operación frecuente es aplicar una función en arrays unidimensionales a cada columna o fila. El método `apply` de `DataFrame` hace exactamente esto:

In [202...

```
def f1(x):
    return x.max() - x.min()
frame.apply(f1)
```

Out[202...

```
b    1.481198
d    2.167731
e    0.589600
dtype: float64
```

Aquí la función `f`, que calcula la diferencia entre el máximo y el mínimo de una `Series`, se invoca una vez en cada columna de `frame`. El resultado es una `Series` que tiene como índice las columnas de `frame`. Si se pasa `axis="columns"` al método `apply`, la función se invocará una vez por fila. Una forma útil de pensar en esto es como "aplicar a través de las columnas":

In [203...

```
frame.apply(f1, axis="columns")
```

Out[203...

```
Utah    2.190038
Ohio    1.614777
Texas   0.648852
Oregon  1.827577
dtype: float64
```

Muchos de los estadísticos de array más comunes (como suma y media) son métodos de `DataFrame`, por lo que no es necesario utilizar `apply`. No es necesario que la función pasada a `apply` devuelva un valor escalar; también puede devolver una `Serie` con múltiples valores:

In [204...

```
def f2(x):
    return pd.Series([x.min(), x.max()], index=["min", "max"])
frame.apply(f2)
```

Out[204...

	b	d	e
min	-1.347003	-1.220009	-1.256654
max	0.134194	0.947723	-0.667054

También se pueden utilizar funciones Python por elementos. Supongamos que desea calcular una cadena formateada a partir de cada valor de coma flotante de `frame`. Puede hacerlo con `applymap`:

```
In [205... def my_format(x):  
             return f"{x:.2f}"  
             frame.applymap(my_format)
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16212\1380983260.py:3: FutureWarning: DataFrame.applymap has been deprecated. Use DataFrame.map instead.  
frame.applymap(my_format)
```

```
Out[205...      b    d    e  
Utah -1.24  0.95 -0.94  
Ohio  0.13  0.95 -0.67  
Texas -1.35 -1.22 -0.70  
Oregon -0.79  0.57 -1.26
```

La razón del nombre `applymap` es que `Series` tiene un método `map` para aplicar una función `element-wise`:

```
In [206... frame["e"].map(my_format)
```

```
Out[206... Utah      -0.94  
Ohio      -0.67  
Texas     -0.70  
Oregon    -1.26  
Name: e, dtype: object
```

Clasificación y ordenación

Ordenar un conjunto de datos por algún criterio es otra importante operación incorporada. Para ordenar lexicográficamente por etiqueta de fila o columna, utilice el método `sort_index`, que devuelve un nuevo objeto ordenado:

```
In [207... obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])  
obj
```

```
Out[207... d    0  
a    1  
b    2  
c    3  
dtype: int32
```

```
In [208... obj.sort_index()
```

```
Out[208... a    1
           b    2
           c    3
           d    0
           dtype: int32
```

Con un `DataFrame` , se puede ordenar por índice en cualquiera de los ejes:

```
In [209... frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                        index=["three", "one"],
                        columns=["d", "a", "b", "c"])

frame
```

```
Out[209...    d  a  b  c
three  0  1  2  3
one    4  5  6  7
```

```
In [210... frame.sort_index()
```

```
Out[210...    d  a  b  c
one    4  5  6  7
three  0  1  2  3
```

```
In [211... frame.sort_index(axis="columns")
```

```
Out[211...    a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

Por defecto, los datos se ordenan en orden ascendente, pero también pueden ordenarse en orden descendente:

```
In [212... frame.sort_index(axis="columns", ascending=False)
```

```
Out[212...    d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Para ordenar una `Serie` por sus valores, utilice su método `sort_values` :

```
In [213... obj = pd.Series([4, 7, -3, 2])
```

```
In [214...] obj.sort_values()
```

```
Out[214...] 2   -3
            3    2
            0    4
            1    7
            dtype: int64
```

Los valores que faltan se ordenan por defecto al final de la serie:

```
In [215...] obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj
```

```
Out[215...] 0    4.0
            1   NaN
            2    7.0
            3   NaN
            4   -3.0
            5    2.0
            dtype: float64
```

```
In [216...] obj.sort_values()
```

```
Out[216...] 4   -3.0
            5    2.0
            0    4.0
            2    7.0
            1   NaN
            3   NaN
            dtype: float64
```

Los valores faltantes pueden ordenarse al principio utilizando la opción `na_position` :

```
In [217...] obj.sort_values(na_position="first")
```

```
Out[217...] 1   NaN
            3   NaN
            4   -3.0
            5    2.0
            0    4.0
            2    7.0
            dtype: float64
```

Al ordenar un DataFrame, puede utilizar los datos de una o varias columnas como claves de ordenación. Para ello, pase uno o más nombres de columna a `sort_values` :

```
In [218...] frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})
frame
```

```
Out[218...
```

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

```
In [219... frame.sort_values("b")
```

```
Out[219...
```

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

Para ordenar por varias columnas, pase una lista de nombres:

```
In [220... frame.sort_values(["a", "b"])
```

```
Out[220...
```

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Miércoles 27/09/2023

La clasificación **Ranking** asigna rangos desde uno hasta el número de puntos de datos válidos en una array, empezando por el valor más bajo. Los métodos **rank** para **Series** y **DataFrame** son el lugar donde buscar; por defecto, **rank** rompe los empates asignando a cada grupo el rango medio:

```
In [221... obj = pd.Series([7, -5, 7, 4, 2, 0, 4])  
obj
```

```
Out[221...] 0    7
            1   -5
            2    7
            3    4
            4    2
            5    0
            6    4
            dtype: int64
```

```
In [222...] obj.rank()
```

```
Out[222...] 0    6.5
            1    1.0
            2    6.5
            3    4.5
            4    3.0
            5    2.0
            6    4.5
            dtype: float64
```

También se pueden asignar rangos según el orden en que se observan en los datos:

```
In [223...] obj.rank(method="first")
```

```
Out[223...] 0    6.0
            1    1.0
            2    7.0
            3    4.0
            4    3.0
            5    2.0
            6    5.0
            dtype: float64
```

Aquí, en lugar de utilizar el rango medio 6,5 para las entradas 0 y 2, se han fijado en 6 y 7 porque la etiqueta 0 precede a la etiqueta 2 en los datos.

También puedes clasificar en orden descendente:

```
In [224...] obj.rank(ascending=False)
```

```
Out[224...] 0    1.5
            1    7.0
            2    1.5
            3    3.5
            4    5.0
            5    6.0
            6    3.5
            dtype: float64
```

Consulte la siguiente Tabla para ver una lista de los métodos de "desempate" (tie-breaking) disponibles.



No description has been provided for this image

DataFrame puede calcular rangos sobre las filas o las columnas:

```
In [225...] frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],  
                                "c": [-2, 5, 8, -2.5]})  
frame
```

```
Out[225...]
```

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

```
In [226...] frame.rank(axis="columns")
```

```
Out[226...]
```

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

Índices de ejes con etiquetas duplicadas

Hasta ahora casi todos los ejemplos que hemos visto tienen etiquetas de eje únicas (valores de índice). Aunque muchas funciones de pandas (como `reindex`) requieren que las etiquetas sean únicas, no es obligatorio. Consideremos una pequeña serie con índices duplicados:

```
In [227...] obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])  
obj
```

```
Out[227...] a    0  
a    1  
b    2  
b    3  
c    4  
dtype: int32
```

La propiedad `is_unique` del índice puede indicarle si sus etiquetas son únicas o no:

```
In [228...] obj.index.is_unique
```

```
Out[228...] False
```

La selección de datos es una de las principales cosas que se comporta de forma diferente con los duplicados. La indexación de una etiqueta con varias entradas devuelve una Serie,

mientras que las entradas únicas devuelven un valor escalar:

```
In [229...] obj["a"]
```

```
Out[229...] a    0  
a    1  
dtype: int32
```

```
In [230...] obj["c"]
```

```
Out[230...] 4
```

Esto puede complicar su código, ya que el tipo de salida de la indexación puede variar en función de si una etiqueta se repite o no. La misma lógica se extiende a la indexación de filas (o columnas) en un DataFrame:

```
In [231...] df = pd.DataFrame(np.random.standard_normal((5, 3)),  
                             index=["a", "a", "b", "b", "c"])  
df
```

```
Out[231...]      0      1      2  
a -0.348367 -1.143884  0.422000  
a  0.136637  0.374342 -1.758208  
b  0.242002  0.449275  0.523375  
b -0.275171  0.765594 -0.209137  
c -0.109974  0.284170  2.527234
```

```
In [232...] df.loc["b"]
```

```
Out[232...]      0      1      2  
b  0.242002  0.449275  0.523375  
b -0.275171  0.765594 -0.209137
```

```
In [233...] df.loc["c"]
```

```
Out[233...] 0    -0.109974  
1      0.284170  
2      2.527234  
Name: c, dtype: float64
```

Resumir y calcular estadísticas descriptivas

Los objetos pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes. La mayoría de ellos entran en la categoría de reducciones o estadísticas de resumen, métodos que extraen un único valor (como la suma o la media) de

una Serie, o una Serie de valores de las filas o columnas de un DataFrame. En comparación con los métodos similares que se encuentran en las matrices NumPy, tienen incorporado el manejo de los datos que faltan. Consideremos un pequeño DataFrame:

```
In [234...] df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],  
                      [np.nan, np.nan], [0.75, -1.3]],  
                      index=["a", "b", "c", "d"],  
                      columns=["one", "two"])  
  
df
```

```
Out[234...]
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

La llamada al método `sum` de DataFrame devuelve una `Serie` que contiene las sumas de las columnas:

```
In [235...] df.sum()
```

```
Out[235...] one    9.25  
            two   -5.80  
            dtype: float64
```

Si se pasa `axis="columnas"` o `axis=1`, se suman las columnas:

```
In [236...] df.sum(axis="columns")
```

```
Out[236...] a    1.40  
            b    2.60  
            c    0.00  
            d   -0.55  
            dtype: float64
```

Cuando una fila o columna entera contiene todos valores NA, la suma es 0, mientras que si algún valor no es NA, entonces el resultado es NA. Esto se puede desactivar con la opción `skipna`, en cuyo caso cualquier valor NA en una fila o columna nombra NA al resultado correspondiente:

```
In [237...] df.sum(axis="index", skipna=False)
```

```
Out[237...] one    NaN  
            two    NaN  
            dtype: float64
```

```
In [238...] df.sum(axis="columns", skipna=False)
```


```
Out[238...] a    NaN
            b    2.60
            c    NaN
            d   -0.55
            dtype: float64
```

Algunas agregaciones, como la media `mean` , requieren al menos un valor no-NA para producir un resultado de valor, así que aquí tenemos:

```
In [239...] df.mean(axis="columns")
```

```
Out[239...] a    1.400
            b    1.300
            c     NaN
            d   -0.275
            dtype: float64
```

Véase la siguiente tabla una lista de opciones habituales para cada método de reducción:

 No description has been provided for this image

Algunos métodos, como `idxmin` e `idxmax` , devuelven estadísticas indirectas, como el valor del índice donde se alcanzan los valores mínimo o máximo:

```
In [240...] df.idxmax()
```

```
Out[240...] one    b
            two    d
            dtype: object
```

Otros métodos son las acumulaciones:

```
In [241...] df.cumsum()
```

```
Out[241...]   one  two
a    1.40  NaN
b    8.50  -4.5
c    NaN  NaN
d    9.25  -5.8
```

```
In [242...] df
```

Out[242...

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Algunos métodos no son ni reducciones ni acumulaciones. `describe` es un ejemplo de ello, ya que produce múltiples estadísticas de resumen de una sola vez:

In [243...

```
df.describe()
```

Out[243...

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

En datos no numéricos, `describe` produce estadísticas de resumen alternativas:

In [244...

```
obj = pd.Series(["a", "a", "b", "c"] * 4)
obj
```


Out[244...

```
0    a
1    a
2    b
3    c
4    a
5    a
6    b
7    c
8    a
9    a
10   b
11   c
12   a
13   a
14   b
15   c
dtype: object
```

```
In [245... obj.describe()
```

```
Out[245... count      16
unique       3
top          a
freq         8
dtype: object
```

Véase la siguiente tabla para una lista completa de estadísticas de síntesis y métodos relacionados.

No description has been provided for this image

Correlación y covarianza

Algunos estadísticos de resumen, como la correlación y la covarianza, se calculan a partir de pares de argumentos. Consideremos algunos DataFrames de precios y volúmenes de acciones obtenidos originalmente de Yahoo! Finance:

```
In [246... price = pd.read_pickle("yahoo_price.pkl")
volume = pd.read_pickle("yahoo_volume.pkl")
```

Por ejemplo, se podría calcular las variaciones porcentuales de los precios:

```
In [248... returns = price.pct_change()
returns.tail()
```

```
Out[248...      AAPL    GOOG    IBM    MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

El método `corr` de Series calcula la correlación de los valores superpuestos, no-NA, alineados-por-índice en dos Series. Por su parte, `cov` calcula la covarianza:

```
In [249... returns["MSFT"].corr(returns["IBM"])
```

```
Out[249... 0.4997636114415114
```

```
In [250... returns["MSFT"].cov(returns["IBM"])
```

```
Out[250... 8.870655479703545e-05
```

Los métodos `corr` y `cov` de un DataFrame, por otro lado, devuelven una matriz de correlación o covarianza completa como un DataFrame, respectivamente:

In [251...

```
returns.corr()
```

Out[251...

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.407919	0.386817	0.389695
GOOG	0.407919	1.000000	0.405099	0.465919
IBM	0.386817	0.405099	1.000000	0.499764
MSFT	0.389695	0.465919	0.499764	1.000000

In [252...

```
returns.cov()
```

Out[252...

	AAPL	GOOG	IBM	MSFT
AAPL	0.000277	0.000107	0.000078	0.000095
GOOG	0.000107	0.000251	0.000078	0.000108
IBM	0.000078	0.000078	0.000146	0.000089
MSFT	0.000095	0.000108	0.000089	0.000215

Utilizando el método `corrwith` de DataFrame, puede calcular correlaciones por pares entre las columnas o filas de un DataFrame con otra Serie o DataFrame. Al pasar una Serie se devuelve una Serie con el valor de correlación calculado para cada columna:

In [253...

```
returns.corrwith(returns["IBM"])
```

Out[253...

```
AAPL    0.386817
GOOG    0.405099
IBM      1.000000
MSFT    0.499764
dtype: float64
```

Al pasar un DataFrame se calculan las correlaciones de los nombres de columna coincidentes. Aquí, se calculan las correlaciones de los cambios porcentuales con el volumen:

In [254...

```
returns.corrwith(volume)
```

Out[254...

```
AAPL    -0.075565
GOOG    -0.007067
IBM      -0.204849
MSFT    -0.092950
dtype: float64
```

Si se pasa `axis="columns"`, se hace fila por fila. En todos los casos, los puntos de datos se alinean por etiqueta antes de calcular la correlación.

Valores únicos, recuento de valores y afiliación

Otra clase de métodos relacionados extrae información sobre los valores contenidos en una Serie unidimensional. Para ilustrarlos, considere este ejemplo:

```
In [255...] obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
```

La primera función es `unique`, que nos da un array de los valores únicos de una Serie:

```
In [257...] uniques = obj.unique()  
uniques
```

```
Out[257...] array(['c', 'a', 'd', 'b'], dtype=object)
```

Los valores únicos no se devuelven necesariamente en el orden en que aparecen por primera vez, y no en orden ordenado, pero podrían ordenarse a posteriori si fuera necesario (`uniques.sort()`). Por otro lado, `value_counts` calcula una serie que contiene las frecuencias de los valores:

```
In [258...] obj.value_counts()
```

```
Out[258...] c    3  
a    3  
b    2  
d    1  
Name: count, dtype: int64
```

La serie se ordena por valor en orden descendente por conveniencia. `value_counts` también está disponible como un método pandas de nivel superior que se puede utilizar con arrays NumPy u otras secuencias de Python:

```
In [259...] pd.value_counts(obj.to_numpy(), sort=False)
```

C:\Users\juanj\AppData\Local\Temp\ipykernel_16212\2238870943.py:1: FutureWarning: pandas.value_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value_counts() instead.

```
pd.value_counts(obj.to_numpy(), sort=False)
```

```
Out[259...] c    3  
a    3  
d    1  
b    2  
Name: count, dtype: int64
```

`isin` realiza una comprobación de pertenencia a un conjunto vectorizado y puede ser útil para filtrar un conjunto de datos a un subconjunto de valores en una Serie o columna en un DataFrame:

```
In [260...] obj
```

```
Out[260...] 0    c
            1    a
            2    d
            3    a
            4    a
            5    b
            6    b
            7    c
            8    c
dtype: object
```

```
In [262...] mask = obj.isin(["b", "c"])
            mask
```

```
Out[262...] 0     True
            1    False
            2    False
            3    False
            4    False
            5     True
            6     True
            7     True
            8     True
dtype: bool
```

```
In [263...] obj[mask]
```

```
Out[263...] 0    c
            5    b
            6    b
            7    c
            8    c
dtype: object
```

Relacionado con `isin` está el método `Index.get_indexer`, que te proporciona una array de índices desde un array de valores posiblemente no distintos a otro array de valores distintos:

```
In [268...] to_match = pd.Series(["c", "a", "b", "b", "c", "a"])
            to_match
```

```
Out[268...] 0    c
            1    a
            2    b
            3    b
            4    c
            5    a
dtype: object
```

```
In [269...] unique_vals = pd.Series(["c", "b", "a"])
            unique_vals
```

```
Out[269... 0    c
          1    b
          2    a
          dtype: object
```

```
In [275... indices = pd.Index(unique_vals).get_indexer(to_match)
          indices
```

```
Out[275... array([0, 2, 1, 1, 0, 2], dtype=int64)
```

En algunos casos, es posible que desee calcular un histograma en varias columnas relacionadas en un DataFrame. He aquí un ejemplo:

```
In [276... data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
                        "Qu2": [2, 3, 1, 2, 3],
                        "Qu3": [1, 5, 2, 4, 4]})
          data
```

```
Out[276...   Qu1  Qu2  Qu3
0     1     2     1
1     3     3     5
2     4     1     2
3     3     2     4
4     4     3     4
```

Podemos calcular los recuentos de valores para una sola columna, de la siguiente manera:

```
In [277... data["Qu1"].value_counts().sort_index()
```

```
Out[277... Qu1
1     1
3     2
4     2
Name: count, dtype: int64
```

Para calcular esto para todas las columnas, pase `pandas.value_counts` al método `apply` del DataFrame:

```
In [278... result = data.apply(pd.value_counts).fillna(0)
          result
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16212\1382616601.py:1: FutureWarning: pa
ndas.value_counts is deprecated and will be removed in a future version. Use pd.Seri
es(obj).value_counts() instead.
    result = data.apply(pd.value_counts).fillna(0)
```


Out[278...

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

Aquí, las etiquetas de fila en el resultado son los valores distintos que aparecen en todas las columnas. Los valores son los recuentos respectivos de estos valores en cada columna.

También existe un método `DataFrame.value_counts`, pero éste calcula los recuentos considerando cada fila del DataFrame como una tupla para determinar el número de ocurrencias de cada fila distinta:

In [279...

```
data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})
data
```

Out[279...

	a	b
0	1	0
1	1	0
2	1	1
3	2	0
4	2	0

In [280...

```
data.value_counts()
```

Out[280...

```
a  b
1  0    2
2  0    2
1  1    1
Name: count, dtype: int64
```

In []: