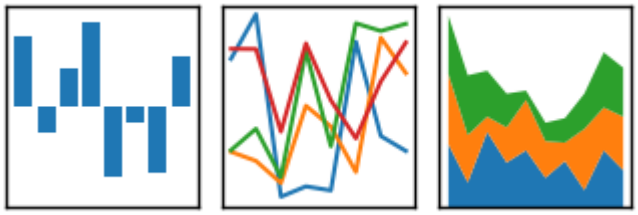
pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$





---

# Introducción a Pandas

`Pandas` es una herramienta esencial para el análisis de datos. Contiene estructuras de datos y herramientas de manipulación de datos diseñadas para que la limpieza y el análisis de datos sea rápido y fácil en Python. Pandas se usa a menudo junto con herramientas de computación numérica como `NumPy` y `SciPy`, bibliotecas analíticas como `statsmodels` y `scikit-learn`, y bibliotecas de visualización de datos como `matplotlib`. Pandas adopta partes significativas del estilo idiomático de NumPy como la computación basada en matrices, especialmente las funciones basadas en matrices y una preferencia por el procesamiento de datos sin bucles.

Si bien en Pandas se adoptan muchos elementos de codificación de NumPy, la mayor diferencia es que Pandas está diseñado para trabajar con datos tabulares o heterogéneos, mientra que NumPy, por contraste, es más adecuado para trabajar con datos de matrices numéricas homogéneas.

In [1]:
```
import pandas as pd
pd.__version__
```

Out[1]:
```
'0.24.2'
```

# Estructuras de datos

Las dos estructuras de datos principales de Pandas son las series ( `Series` ) y los marcos de datos ( `DataFrame` ). Si bien no son una solución universal para todos los problemas, proporcionan una base sólida y fácil de usar para la mayoría de las aplicaciones.

## Series

Una serie es un objeto similar a una matriz unidimensional que contiene una secuencia de valores (de tipos similares a los tipos NumPy) y una matriz asociada de etiquetas de datos, denominada índice. La serie más simple está formada a partir de una matriz de datos. Si no especificamos un índice para los datos, se crea uno predeterminado que consiste en números enteros de 0 a N - 1 (donde N es la longitud de los datos):

In [2]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

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

Se pueden obtener los valores y el índice de la serie a través de sus atributos `values` e `index` respectivamente:

In [3]:
```
obj.values
```

Out[3]:
```
array([ 4,  7, -5,  3])
```

In [4]:
```
obj.index  # como range(4)
```

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

Aunque el índice de una serie se puede modificar *in situ* por asignación:

In [5]:
```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
obj
```

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

Lo habitual es proporcionar el índice a utilizar cuando se crea la serie:

In [6]:
```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

Out[6]:
```
d    4
b    7
a   -5
c    3
dtype: int64
```

In [7]:
```
obj2.index
```

Out[7]:
```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

A diferencia de los arrays en Numpy, se pueden utilizar las etiquetas de los índices para acceder a valores o conjuntos de valores:

In [8]:
```
obj2['a']
```

Out[8]:
```
-5
```

In [9]:
```
obj2['d'] = 6
obj2
```

Out[9]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [10]:
```
obj2[['c', 'a', 'd']]
```

Out[10]:
```
c    3
a   -5
d    6
dtype: int64
```

Aquí `['c', 'a', 'd']` se interpreta como una lista de índices, aunque contenga cadenas en lugar de números enteros.

El uso de funciones NumPy u operaciones similares a NumPy, como el filtrado con una matriz booleana, la multiplicación escalar o la aplicación de funciones matemáticas, siempre mantiene el valor del índice:

```
In [11]: obj2[obj2 > 0]
```

```
Out[11]: d    6
         b    7
         c    3
         dtype: int64
```

```
In [12]: obj2 * 2
```

```
Out[12]: d    12
         b    14
         a   -10
         c     6
         dtype: int64
```

Si los datos que se proporcionan para crear un serie es un valor escalar, se debe proporcionar un índice. El valor se repetirá para que coincida con la longitud del índice.

```
In [13]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

```
Out[13]: a    5.0
         b    5.0
         c    5.0
         d    5.0
         e    5.0
         dtype: float64
```

Cuando se trabaja con matrices NumPy sin procesar, normalmente no es necesario realizar un bucle de valor por valor. Lo mismo ocurre cuando se trabaja con Series en pandas. Las series también se pueden pasar a la mayoría de los métodos NumPy que esperan un ndarray.

```
In [14]: from numpy import exp
         exp(obj2)
```

```
Out[14]: d     403.428793
         b    1096.633158
         a       0.006738
         c      20.085537
         dtype: float64
```

Una diferencia clave entre `series` y `ndarray` es que las operaciones entre `series` alinean automáticamente los datos en función de la etiqueta. Por lo tanto, se pueden escribir cálculos sin tener en cuenta si las `series` involucradas tienen las mismas etiquetas o si éstas están en el mismo orden. Similar a las operaciones de unión de las bases de datos relacionales.

In [15]:
```
obj2
```

Out[15]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [16]:
```
obj3 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
obj3
```

Out[16]:
```
a    1
b    2
c    3
d    4
dtype: int64
```

In [17]:
```
obj2 + obj3
```

Out[17]:
```
a   -4
b    9
c    6
d   10
dtype: int64
```

El resultado de una operación entre Series no alineadas tendrá la unión de los índices involucrados. Si no se encuentra una etiqueta en una Serie u otra, el resultado se marcará como falta `NaN` . Ser capaz de escribir código sin hacer una alineación de datos explícita otorga inmensa libertad y flexibilidad en el análisis e investigación de datos interactivos. Las características integradas de alineación de datos de las estructuras de datos de pandas diferencian a los pandas de la mayoría de las herramientas relacionadas para trabajar con datos etiquetados.

In [18]:
```
obj4 = pd.Series([1, 1, 1, 1], index=['x', 'y', 'c', 'd'])
obj4
```

Out[18]:
```
x    1
y    1
c    1
d    1
dtype: int64
```

In [19]:
```
obj2 + obj4
```

Out[19]:
```
a    NaN
b    NaN
c    4.0
d    7.0
x    NaN
y    NaN
dtype: float64
```

Otra forma de pensar en una serie es como un diccionario ordenado de longitud fija, ya que es un mapeo de valores de índice a valores de datos. Se puede utilizar en muchos contextos en los que puede usar un diccionario ( `dict` ):

In [20]:
```
'b' in obj2
```

Out[20]:
```
True
```

In [21]:
```
'e' in obj2
```

Out[21]:
```
False
```

Podemos crear series a partir de diccionarios Python:

In [22]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Uta
h': 5000}
obj3 = pd.Series(sdata)
obj3
```

Out[22]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Cuando se pasa un diccionario, el índice en la serie resultante tendrá las claves del diccionario ordenadas. Se puede anular el orden pasando las claves del diccionario en el orden en que desea que aparezcan en la serie resultante:

In [23]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

Out[23]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

En el ejemplo anterior se encuentras tres valores en `sdata` que se ubican en las posiciones apropiadas, pero como no se encontró ningún valor para 'California', aparece como `NaN` (*Not an Number*) que se utiliza en pandas para marcar valores faltantes o `NA` (*Not Available*). Dado que 'Utah' no se incluyó en los estados, se excluye del objeto resultante.

> En pandas, se adopta la convención utilizada en el lenguaje de programación `R` al referirnos a datos faltantes como `NA` , que significa no disponible. En aplicaciones de estadísticas, los datos de NA pueden ser datos que no existen o que existen pero que no se observaron.

Las funciones y/o métodos `isnull` y `notnull` en pandas se usan para detectar datos faltantes:

```
In [24]: pd.isnull(obj4) # equivalente a obj4.isnull()
```

```
Out[24]: California     True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool
```

```
In [25]: pd.notnull(obj4) # equivalente a obj4.notnull()
```

```
Out[25]: California    False
         Ohio           True
         Oregon         True
         Texas          True
         dtype: bool
```

Tanto las series como su índice tienen un atributo de nombre ( `name` ), que se integra con otras áreas clave de la funcionalidad de los pandas:

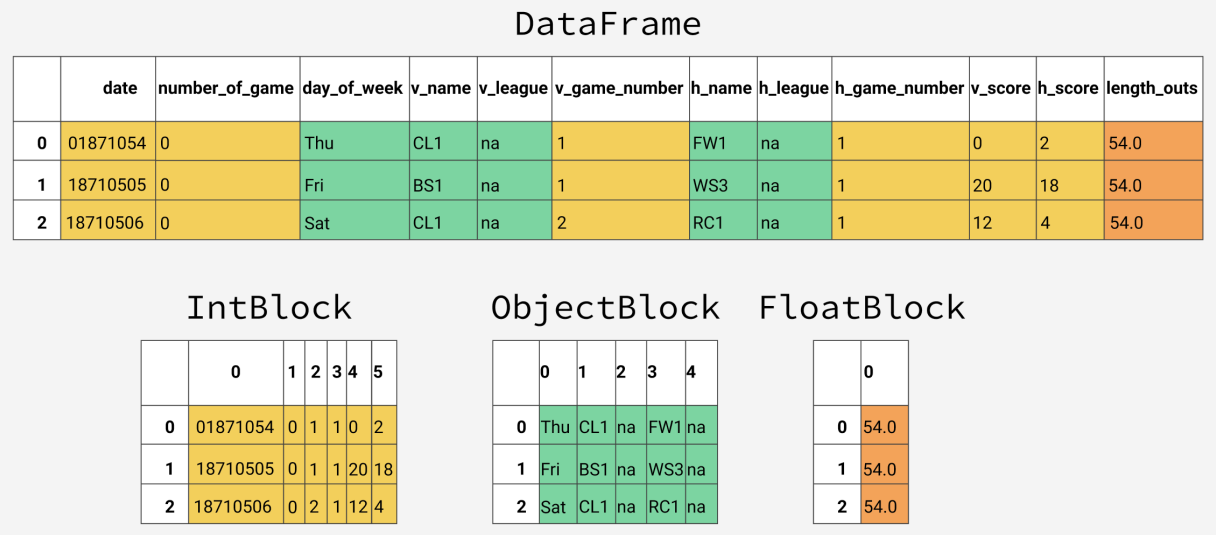
```
In [26]: obj4.name = 'population'
         obj4.index.name = 'state'
         obj4
```

```
Out[26]: state
         California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         Name: population, dtype: float64
```

## DataFrame

Un marco de datos o `DataFrame` representa una tabla de datos rectangular y contiene una colección ordenada de columnas, cada una de las cuales puede ser un tipo de valor diferente (numérico, de cadena, booleano, etc.). El DataFrame tiene un índice de fila y columna. Se puede considerar como un diccionario de series que comparten el mismo índice.

Tal y como se visualiza en la siguiente imagen, internamente los datos se almacenan como uno o más bloques bidimensionales (del mismo tipo de datos) en lugar de una lista, diccionario, o alguna otra colección de arrays unidimensionales.




DataFrame

| | date | number_of_game | day_of_week | v_name | v_league | v_game_number | h_name | h_league | h_game_number | v_score | h_score | length_outs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 01871054 | 0 | Thu | CL1 | na | 1 | FW1 | na | 1 | 0 | 2 | 54.0 |
| 1 | 18710505 | 0 | Fri | BS1 | na | 1 | WS3 | na | 1 | 20 | 18 | 54.0 |
| 2 | 18710506 | 0 | Sat | CL1 | na | 2 | RC1 | na | 1 | 12 | 4 | 54.0 |

IntBlock

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 01871054 | 0 | 1 | 1 | 0 | 2 |
| 1 | 18710505 | 0 | 1 | 1 | 20 | 18 |
| 2 | 18710506 | 0 | 2 | 1 | 12 | 4 |

ObjectBlock

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | Thu | CL1 | na | FW1 | na |
| 1 | Fri | BS1 | na | WS3 | na |
| 2 | Sat | CL1 | na | RC1 | na |

FloatBlock

| | 0 |
|---|---|
| 0 | 54.0 |
| 1 | 54.0 |
| 2 | 54.0 |

> Aunque un `DataFrame` es físicamente bidimensional, se puede usar para representar datos de dimensiones más altas en un formato tabular utilizando la indexación jerárquica.

Los bloques no mantienen referencias a los nombres de las columnas. Esto se debe a que los bloques están optimizados para almacenar los valores del marco de datos. La clase `BlockManager` es responsable de mantener el mapeo entre los índices de fila y columna y los bloques reales. Actúa como una API que proporciona acceso a los datos subyacentes. Siempre que seleccionamos, editamos o eliminamos valores, la clase `DataFrame` interactúa con la clase `BlockManager` para traducir nuestras solicitudes en llamadas a funciones y métodos.

Cada tipo tiene una clase especializada en el módulo `pandas.core.internals` . Pandas usa la clase `ObjectBlock` para representar el bloque que contiene columnas de cadena y la clase `FloatBlock` para representar el bloque que contiene columnas flotantes. Para bloques que representan valores numéricos como enteros y flotantes, pandas combina las columnas y las almacena como un `ndarray NumPy` . Como sabemos, el ndarray de NumPy se basa en una matriz C y los valores se almacenan en un bloque de memoria contiguo. Debido a este esquema de almacenamiento, acceder a una porción de valores es increíblemente rápido.

La siguiente tabla muestra los tipos de datos más comunes de pandas (heredados de `numpy` ), los subtipos listados usan 2, 4, 8 y 16 bytes respectivamente:

| memory usage | float | int | uint | datetime | bool | object |
|---|---|---|---|---|---|---|
| 1 bytes | | int8 | uint8 | | bool | |
| 2 bytes | float16 | int16 | uint16 | | | |
| 4 bytes | float32 | int32 | uint32 | | | |
| 8 bytes | float64 | int64 | uint64 | datetime64 | | |
| variable | | | | | | object |

El uso de los tipos de datos adecuados es el primer paso para aprovechar al máximo Pandas. Actualmente hay dos tipos de datos para datos textuales, `object` y `StringDtype` (a partir de Pandas 1.0). El tipo de datos `object` tiene un alcance más amplio y permite almacenar prácticamente cualquier cosa. Una cosa importante a tener en cuenta es que el tipo de datos `object` sigue siendo el tipo de datos predeterminado para las cadenas. Para usar `StringDtype` , necesitamos indicarlo explícitamente.

> Por cuestiones de compatibilidad el tipo `object` representa valores utilizando cadenas de Python. Esta limitación hace que las cadenas se almacenen de forma fragmentada, lo que consume más memoria y el acceso a las mismas sea es más lento. Cada elemento de una columna de tipo `object` es en realidad un puntero que contiene la "dirección" de la ubicación del valor real en la memoria. El tipo de datos `StringDtype` no es superior a `object` en términos de rendimiento, sin embargo se espera que, con futuras actualizaciones de pandas, el rendimiento del tipo `StringDtype` aumente y su consumo de memoria disminuya.

Hay muchas formas de construir un `DataFrame` , aunque una de las más comunes es desde un diccionario de listas de igual longitud o desde matrices NumPy:

In [27]:
```
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 ocurre con las series, y las columnas se colocan de forma ordenada:

In [28]:
```
frame
```

Out[28]:

| | 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 |

In [29]:
```
# El método head permite mostrar las 5 primeras filas por def
ecto
frame.head()
```

Out[29]:

| | 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 |

In [30]:
```
# El método tail permite mostrar las 5 últimas filas por defecto
frame.tail(3)
```

Out[30]:

| | state | year | pop |
|---|---|---|---|
| 3 | Nevada | 2001 | 2.4 |
| 4 | Nevada | 2002 | 2.9 |
| 5 | Nevada | 2003 | 3.2 |

Si especifica una secuencia de columnas, el `DataFrame` mostrará sus columnas en la secuencia indicada:

In [31]:
```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[31]:

| | 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 se indica una columna que no está contenida en el diccionario, aparecerá con valores `NA` ( `NaN` ) en el resultado:

In [32]:
```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                      index=['one', 'two', 'three', 'four', 'five', 'six'])
frame2
```

Out[32]:

| | year | state | pop | debt |
|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | NaN |
| two | 2001 | Ohio | 1.7 | NaN |
| three | 2002 | Ohio | 3.6 | NaN |
| four | 2001 | Nevada | 2.4 | NaN |
| five | 2002 | Nevada | 2.9 | NaN |
| six | 2003 | Nevada | 3.2 | NaN |

In [33]:
```
frame2.columns
```

Out[33]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Una columna en un `DataFrame` se puede recuperar como una `Serie`, ya sea con notación tipo diccionario o vía atributo:

In [34]:
```
frame2['state']
```

Out[34]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

In [35]:
```
# sólo si el nombre de la columna no coincide con una propied
ad o método del DataFrame
frame2.year
```

Out[35]:
```
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

Las series devueltas tienen el mismo índice que el DataFrame, y su atributo `name` se ha establecido correctamente.

Las filas también se pueden recuperar por posición o nombre con el atributo especial `loc`:

In [36]:
```
frame2.loc['three']
```

Out[36]:
```
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

Las columnas pueden ser modificadas por asignación:

In [37]:
```
# asignación de un valor escalar
frame2['debt'] = 16.5
frame2
```

Out[37]:

| | year | state | pop | debt |
|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | 16.5 |
| two | 2001 | Ohio | 1.7 | 16.5 |
| three | 2002 | Ohio | 3.6 | 16.5 |
| four | 2001 | Nevada | 2.4 | 16.5 |
| five | 2002 | Nevada | 2.9 | 16.5 |
| six | 2003 | Nevada | 3.2 | 16.5 |

In [38]:
```
# asignación de un vector
import numpy as np
frame2['debt'] = np.arange(6.)
frame2
```

Out[38]:

| | year | state | pop | debt |
|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | 0.0 |
| two | 2001 | Ohio | 1.7 | 1.0 |
| three | 2002 | Ohio | 3.6 | 2.0 |
| four | 2001 | Nevada | 2.4 | 3.0 |
| five | 2002 | Nevada | 2.9 | 4.0 |
| six | 2003 | Nevada | 3.2 | 5.0 |

Cuando se están asignando listas o matrices a una columna, la longitud de valores debe coincidir con la longitud del DataFrame. Si se asigna una serie, sus etiquetas se realinearán exactamente al índice del DataFrame, insertando valores `NA` en el resto de huecos:

In [39]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
frame2['debt'] = val
frame2
```

Out[39]:

| | year | state | pop | debt |
|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | NaN |
| two | 2001 | Ohio | 1.7 | -1.2 |
| three | 2002 | Ohio | 3.6 | NaN |
| four | 2001 | Nevada | 2.4 | -1.5 |
| five | 2002 | Nevada | 2.9 | -1.7 |
| six | 2003 | Nevada | 3.2 | NaN |

Asignar una columna que no existe creará una nueva columna. La palabra clave `del` borrará columnas como con un diccionario:

In [40]:
```
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

Out[40]:

| | year | state | pop | debt | eastern |
|---|---|---|---|---|---|
| one | 2000 | Ohio | 1.5 | NaN | True |
| two | 2001 | Ohio | 1.7 | -1.2 | True |
| three | 2002 | Ohio | 3.6 | NaN | True |
| four | 2001 | Nevada | 2.4 | -1.5 | False |
| five | 2002 | Nevada | 2.9 | -1.7 | False |
| six | 2003 | Nevada | 3.2 | NaN | False |

In [41]:
```
del frame2['eastern']
frame2.columns
```

Out[41]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

La columna devuelta de la indexación de un DataFrame es una vista de los datos subyacentes, no una copia. Por lo tanto, cualquier modificación *in situ* de la serie se reflejará en el DataFrame. La columna se puede copiar explícitamente con el método `copy` de la serie.

Otra forma común de crear un DataFrame es utilizar diccionarios anidados. Pandas interpretará las claves del diccionario externo como las columnas y las claves de diccionario interno como los índices de fila:

In [42]:
```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

frame3 = pd.DataFrame(pop)
frame3
```

Out[42]:

| | Nevada | Ohio |
|---|---|---|
| 2001 | 2.4 | 1.7 |
| 2002 | 2.9 | 3.6 |
| 2000 | NaN | 1.5 |

Se puede transponer el DataFrame (intercambiar filas y columnas) con una sintaxis similar a una matriz NumPy:

In [43]:
```
frame3.T
```

Out[43]:

| | 2001 | 2002 | 2000 |
|---|---|---|---|
| Nevada | 2.4 | 2.9 | NaN |
| Ohio | 1.7 | 3.6 | 1.5 |

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

In [44]:
```
pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[44]:

| | Nevada | Ohio |
|---|---|---|
| 2001 | 2.4 | 1.7 |
| 2002 | 2.9 | 3.6 |
| 2003 | NaN | NaN |

Los diccionarios de series se tratan de la misma manera:

In [45]:
```
pdata = {'Ohio': frame3['Ohio'][:-1],
         'Nevada': frame3['Nevada'][:2]}
pd.DataFrame(pdata)
```

Out[45]:

| | Ohio | Nevada |
|---|---|---|
| 2001 | 1.7 | 2.4 |
| 2002 | 3.6 | 2.9 |

Si el atributo `name` del índice y de las columnas de un DataFrame están fijados, sus valores serán mostrados:

In [46]:
```
frame3.index.name = 'year'
frame3.columns.name = 'state'
frame3
```

Out[46]:

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

Al igual que con las series, el atributo `values` devuelve los datos contenidos en el DataFrame como un ndarray bidimensional:

In [47]:
```
frame3.values
```

Out[47]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

Si las columnas del DataFrame son de tipos diferentes, el tipo de la matriz de valores se elegirá para acomodar todas las columnas:

In [48]:
```
frame2.values
```

Out[48]:
```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

La siguiente tabla muestra las posibles entradas para la construcción de un `DataFrame` :

| Tipo | Descripción |
|---|---|
| 2D ndarray | Una matriz de datos, pasando las etiquetas opcionales de fila y columna |
| dict de matrices, lists, o tuples | Cada secuencia se convierte en una columna en el DataFrame; Todas las secuencias deben tener la misma longitud |
| NumPy structured/record array | Se tratan como el caso del "diccionario de matrices" |
| dict de Series | Cada valor se convierte en una columna; los índices de cada Serie se unen para formar el índice de la fila del resultado si no se pasa un índice explícito |
| dict de dicts | Cada diccionario interno se convierte en una columna; Las claves están unidas para formar el índice de fila como en el caso "dict de Series" |
| List de dicts o Series | Cada elemento se convierte en una fila en el marco de datos; La unión de las claves de dict o los índices de serie se convierten en las etiquetas de columna de DataFrame |
| List de lists o tuples | Tratada como el caso "2D ndarray" |
| Another DataFrame | Los índices de DataFrame se utilizan a menos que se pasen diferentes |
| NumPy MaskedArray | Al igual que en el caso de "ndarray 2D", los valores enmascarados se convierten en NA/faltantes en el resultado del marco de datos |

## Objetos Índice

Los objetos índice ( `Index` ) de pandas son responsables de mantener las etiquetas del eje y otros metadatos (como el nombre o los nombres del eje). Cualquier matriz u otra secuencia de etiquetas que utilice al construir una serie o un DataFrame se convierte internamente en un `Index` :

In [49]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index
```

Out[49]:
```
Index(['a', 'b', 'c'], dtype='object')
```

Los índices son inmutables:

In [50]:
```
index[1:]
```

Out[50]:
```
Index(['b', 'c'], dtype='object')
```

In [51]:
```
index[1] = 'd'  # TypeError
```

```
---------------------------------------------------------------
---------------
TypeError                                 Traceback (most re
cent call last)
<ipython-input-51-d11f5623d88a> in <module>
----> 1 index[1] = 'd'  # TypeError

/opt/anaconda3/lib/python3.7/site-packages/pandas/core/index
es/base.py in __setitem__(self, key, value)
   4258
   4259     def __setitem__(self, key, value):
-> 4260         raise TypeError("Index does not support muta
ble operations")
   4261
   4262     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

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

In [52]:
```
import numpy as np
labels = pd.Index(np.arange(3))
labels
```

Out[52]:
```
Int64Index([0, 1, 2], dtype='int64')
```

In [53]:
```
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
obj2
```

Out[53]:
```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

In [54]:
```
obj2.index is labels
```

Out[54]:
```
True
```

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

In [55]:
```
frame3
```

Out[55]:

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

In [56]:
```
frame3.columns
```

Out[56]:
```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

In [57]:
```
'Ohio' in frame3.columns
```

Out[57]:
```
True
```

In [58]:
```
2003 in frame3.index
```

Out[58]:
```
False
```

A diferencia de Python, los índices de pandas pueden contaner valores duplicados. Las selecciones con etiquetas duplicadas seleccionarán todas las apariciones de esa etiqueta.

In [59]:
```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
dup_labels
```

Out[59]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Los índices pueden tener múltiples niveles:

In [60]:
```
arrays = [[0, 0, 1, 1, 2, 2],
          ['one', 'two', 'one', 'two', 'one', 'two']]
tuples = list(zip(*arrays))
index = pd.MultiIndex.from_tuples(tuples, names=['first', 'se
cond'])
index
```

Out[60]:
```
MultiIndex([(0, 'one'),
            (0, 'two'),
            (1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['first', 'second'])
```

In [61]:
```
index = pd.MultiIndex.from_product([range(3), ['one', 'two
']], names=['first', 'second'])
index
```

Out[61]:
```
MultiIndex([(0, 'one'),
            (0, 'two'),
            (1, 'one'),
            (1, 'two'),
            (2, 'one'),
            (2, 'two')],
           names=['first', 'second'])
```

In [62]:
```
import numpy as np
pd.Series(np.random.randn(6), index=index)
```

Out[62]:
```
first  second
0      one       3.176108
       two       0.643229
1      one       0.043312
       two       0.398061
2      one       0.412433
       two       1.039750
dtype: float64
```

In [63]:
```
index.levels[1]
```

Out[63]:
```
Index(['one', 'two'], dtype='object', name='second')
```

In [64]:
```
index.names
```

Out[64]:
```
FrozenList(['first', 'second'])
```

In [65]:
```
index.set_levels(["a", "b"], level=1)
```

Out[65]:
```
MultiIndex([(0, 'a'),
            (0, 'b'),
            (1, 'a'),
            (1, 'b'),
            (2, 'a'),
            (2, 'b')],
           names=['first', 'second'])
```

In [66]:
```
index.get_level_values(0)
```

Out[66]:
```
Int64Index([0, 0, 1, 1, 2, 2], dtype='int64', name='first')
```

> **Importante**: Aunque un índice puede contener valores perdidos ( `NaN` ), debe evitarse si no se desea ningún resultado inesperado. Por ejemplo, algunas operaciones excluyen implícitamente los valores perdidos.

Los índices tiene una serie de métodos y propiedades para establecer la lógica y obtener informaciones comunes sobre los datos que contienen, los más útiles se resumen en la siguiente tabla:

| Método | Descripción |
|---|---|
| append | Concatena los índices produciendo un nuevo índice |
| difference | Devuelve la diferencia de conjuntos como un índice |
| intersection | Devuelve la intersección de conjuntos como un índice |
| union | Devuelve el conjunto unión como un índice |
| isin | Define una matriz booleana que indica si cada valor está contenido en la colección pasada |
| delete | Define un nuevo índice eliminando el elemento `i` |
| drop | Define un nuevo índice eliminando los elementos indicados |
| insert | Define un nuevo índice insertando los elementos indicados |
| is_monotonic | Devuelve `True` si cada elemento es mayor o igual que el elemento anterior |
| is_unique | Devuele `True` si el Índice no tiene valores duplicados |
| unique | Define un nuevo índice sin elementos duplicados |

# Funcionalidad básica

Esta sección incluye los mecanismos fundamentales de la interacción con los datos contenidos en una serie ( `Serie` ) o un marco de datos ( `DataFrame` ).

## Reindexación

El método `set_index` que toma un nombre de columna (para un índice regular) o una lista de nombres de columna (para un `MultiIndex` ). Para crear un nuevo DataFrame re-indexado:

In [67]:
```
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],
        'info': [5, 7, 6, 4, 9, 2]}
frame = pd.DataFrame(data)
frame
```

Out[67]:

| | state | year | pop | info |
|---|---|---|---|---|
| 0 | Ohio | 2000 | 1.5 | 5 |
| 1 | Ohio | 2001 | 1.7 | 7 |
| 2 | Ohio | 2002 | 3.6 | 6 |
| 3 | Nevada | 2001 | 2.4 | 4 |
| 4 | Nevada | 2002 | 2.9 | 9 |
| 5 | Nevada | 2003 | 3.2 | 2 |

In [68]:
```
indexed1 = frame.set_index('state')
indexed1
```

Out[68]:

| | year | pop | info |
|---|---|---|---|
| state | | | |
| Ohio | 2000 | 1.5 | 5 |
| Ohio | 2001 | 1.7 | 7 |
| Ohio | 2002 | 3.6 | 6 |
| Nevada | 2001 | 2.4 | 4 |
| Nevada | 2002 | 2.9 | 9 |
| Nevada | 2003 | 3.2 | 2 |

In [69]:
```
indexed2 = frame.set_index(['state', 'year'])
indexed2
```

Out[69]:

| | | pop | info |
|---|---|---|---|
| state | year | | |
| | 2000 | 1.5 | 5 |
| Ohio | 2001 | 1.7 | 7 |
| | 2002 | 3.6 | 6 |
| | 2001 | 2.4 | 4 |
| Nevada | 2002 | 2.9 | 9 |
| | 2003 | 3.2 | 2 |

In [70]:
```
indexed3 = frame.set_index('state', drop=False)
indexed3
```

Out[70]:

| | state | year | pop | info |
|---|---|---|---|---|
| **state** | | | | |
| Ohio | Ohio | 2000 | 1.5 | 5 |
| Ohio | Ohio | 2001 | 1.7 | 7 |
| Ohio | Ohio | 2002 | 3.6 | 6 |
| Nevada | Nevada | 2001 | 2.4 | 4 |
| Nevada | Nevada | 2002 | 2.9 | 9 |
| Nevada | Nevada | 2003 | 3.2 | 2 |

In [71]:
```
# indexed3 = indexed3.set_index('year', append=True)
indexed3.set_index('year', append=True, inplace=True)
```

In [72]:
```
indexed3
```

Out[72]:

| | | state | pop | info |
|---|---|---|---|---|
| **state** | **year** | | | |
| | 2000 | Ohio | 1.5 | 5 |
| Ohio | 2001 | Ohio | 1.7 | 7 |
| | 2002 | Ohio | 3.6 | 6 |
| | 2001 | Nevada | 2.4 | 4 |
| Nevada | 2002 | Nevada | 2.9 | 9 |
| | 2003 | Nevada | 3.2 | 2 |

Hay una nueva función en DataFrame llamada `reset_index` que transfiere los valores del índice a las columnas del DataFrame y establece un índice entero simple. Esta es la operación inversa de `set_index` .

In [73]:
```
indexed2.reset_index()
```

Out[73]:

| | state | year | pop | info |
|---|---|---|---|---|
| 0 | Ohio | 2000 | 1.5 | 5 |
| 1 | Ohio | 2001 | 1.7 | 7 |
| 2 | Ohio | 2002 | 3.6 | 6 |
| 3 | Nevada | 2001 | 2.4 | 4 |
| 4 | Nevada | 2002 | 2.9 | 9 |
| 5 | Nevada | 2003 | 3.2 | 2 |

In [74]:
```
indexed2.reset_index(level=1)
```

Out[74]:

| | year | pop | info |
|---|---|---|---|
| state | | | |
| Ohio | 2000 | 1.5 | 5 |
| Ohio | 2001 | 1.7 | 7 |
| Ohio | 2002 | 3.6 | 6 |
| Nevada | 2001 | 2.4 | 4 |
| Nevada | 2002 | 2.9 | 9 |
| Nevada | 2003 | 3.2 | 2 |

Otro método importante en pandas es `reindex` que permite crear un nuevo objeto con los datos ajustados a un nuevo índice. Cuando reindexamos una serie, se reorganizan los datos de acuerdo con el nuevo índice, introduciendo valores `NaN` para los ínices no presentes:

In [75]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a',
'c'])
obj
```

Out[75]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

In [76]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2
```

Out[76]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Para datos ordenados como series de tiempo, puede ser conveniente hacer una interpolación o llenado de valores al reindexar. La opción `method` nos permite hacer esto, utilizando un método como `ffill`, que rellena los valores hacia adelante:

In [77]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2,
4])
obj3
```

Out[77]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [78]:
```
obj3.reindex(range(6), method='ffill')
```

Out[78]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

Con DataFrame, la reindexación puede alterar el índice (fila), las columnas o ambos. Cuando se pasa solo una secuencia, se reindexan las filas en el resultado:

In [79]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                     columns=['Ohio', 'Texas', 'California'])
frame
```

Out[79]:

| | Ohio | Texas | California |
|---|---|---|---|
| a | 0 | 1 | 2 |
| c | 3 | 4 | 5 |
| d | 6 | 7 | 8 |

In [80]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

Out[80]:

| | 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 |

Las columnas pueden ser reindexadas con la palabra clave `columns`:

In [81]:
```
states = ['Texas', 'Utah', 'California']
frame2.reindex(columns=states)
```

Out[81]:

| | Texas | Utah | California |
|---|---|---|---|
| a | 1.0 | NaN | 2.0 |
| b | NaN | NaN | NaN |
| c | 4.0 | NaN | 5.0 |
| d | 7.0 | NaN | 8.0 |

La siguiente tabla muestra diferentes argumentos de la función de reindexación:

| Argumento | Descripción |
|---|---|
| index | Nueva secuencia para usar como índice. Puede ser una instancia de índice o cualquier otra estructura de datos de Python similar a una secuencia. Un índice se utilizará exactamente como está sin ninguna copia |
| method | Método de interpolación (relleno): `ffill` se llena hacia adelante, mientras que `bfill` se llena hacia atrás |
| fill_value | Valor de reemplazo que se utilizará cuando se introducen datos faltantes mediante la reindexación |
| limit | Cuando se interpola hacia adelante o hacia atrás, el espacio máximo se puede llenar (en número de elementos) |
| tolerance | Cuando se interpola hacia adelante o hacia atrás, la separación máxima de tamaño (en distancia numérica absoluta) se debe completar para coincidencias inexactas |
| level | Ajusta un índice simple con el nivel de MultiIndex; en otro caso se selecciona un subconjunto del mismo |
| copy | Si es `True`, siempre se copian los datos subyacentes, incluso si el nuevo índice es equivalente al índice anterior; si es `False`, no copia los datos cuando los índices son equivalentes |

## Eliminando entradas de un eje

Eliminar una o más entradas de un eje es fácil si ya se tiene una matriz o lista de índices sin esas entradas. El método `drop` devuelve un nuevo objeto con el valor o valores eliminados del eje indicado:

In [82]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e
'])
obj
```

Out[82]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [83]:
```
new_obj = obj.drop('c')
new_obj
```

Out[83]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [84]:
```
obj.drop(['d', 'c'])
```

Out[84]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

En un DataFrame, los valores de índice se pueden eliminar de cualquiera de los ejes.

In [85]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New Y
ork'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[85]:

| | 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 las filas (eje 0):

In [86]:
```
data.drop(['Colorado', 'Ohio'])
```

Out[86]:

| | one | two | three | four |
|---|---|---|---|---|
| Utah | 8 | 9 | 10 | 11 |
| New York | 12 | 13 | 14 | 15 |

Se pueden eliminar valores de las columnas pasando `axis = 1` o `axis = 'columns'` :

In [87]:
```
data.drop('two', axis=1)
data.drop(['two', 'four'], axis='columns')
```

Out[87]:

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

Muchas funciones, como `drop` , que modifican el tamaño o la forma de una serie o marco de datos, pueden manipular un objeto in situ sin devolver un nuevo objeto:. Hay que tener cuidado con `inplace` , ya que destruye los datos que se eliminan.

In [88]:
```
obj.drop('c', inplace=True)
obj
```

Out[88]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

## Indexación, Selección y filtrado

La información de etiquetado del eje en los objetos pandas sirve para muchos propósitos:

- Identifica datos (es decir, proporciona metadatos) utilizando indicadores conocidos, importantes para el análisis, la visualización y la visualización de la consola interactiva.
- Permite la alineación automática y explícita de los datos.
- Permite obtener y configurar intuitivamente los subconjuntos del conjunto de datos.

La indexación de series ( `obj [...]` ) funciona de manera análoga a la indexación de matrices en NumPy, excepto que pueden usar los valores de índice de la serie en lugar de solo números enteros. También actúa como un diccionario estándar de Python. Si tenemos en cuenta estas dos analogías, nos ayudará a comprender los patrones de indexación y selección de datos en estas matrices.

In [89]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```

Out[89]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

La selección de objetos ha tenido varias adiciones solicitadas por el usuario para admitir una indexación más explícita basada en la ubicación. Pandas ahora admite diferentes tipos de indexación de ejes múltiples.

- Una sola etiqueta, por ej. 5 o 'a' (Tenga en cuenta que 5 se interpreta como una etiqueta del índice. Este uso no es una posición entera a lo largo del índice).
- Una lista o conjunto de etiquetas ['a', 'b', 'c'], [1, 2, 3].
- Un objeto de división con etiquetas 'a' : 'f' (Tenga en cuenta que, a diferencia de las secciones típicas de Python, se incluyen tanto el inicio como el final, cuando están presentes en el índice) o '1 : 3'
- Una matriz booleana
- Una función que se puede llamar con un argumento (la serie que llama o el marco de datos) y que devuelve un resultado válido para la indexación (uno de los anteriores).

In [90]:
```
# Una sola etiqueta
obj['b']
```

Out[90]:
```
1.0
```

In [91]:
```
# Una sola etiqueta
obj[1]
```

Out[91]:
```
1.0
```

In [92]:
```
# Un objeto de división con etiquetas
# el filtrado con etiquetas se comporta de manera diferente a
l de Python en que el punto final es inclusivo
obj['a':'c']
```

Out[92]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [93]:
```
# Un objeto de división con enteros
obj[0:3]
```

Out[93]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [94]:
```
# Un objeto de división con enteros
obj[::-1]
```

Out[94]:
```
d    3.0
c    2.0
b    1.0
a    0.0
dtype: float64
```

In [95]:
```
# Una lista o conjunto de etiquetas
obj[['b', 'a', 'd']]
```

Out[95]:
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [96]:
```
# Una lista o conjunto de enteros
obj[[1, 0, 3]]
```

Out[96]:
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [97]:
```
# Una matriz booleana
obj[obj < 2]
```

Out[97]:
```
a    0.0
b    1.0
dtype: float64
```

In [98]:
```
# Una función
obj[lambda s: s < 2]
```

Out[98]:
```
a    0.0
b    1.0
dtype: float64
```

La indexación en un DataFrame permite recuperar una o más columnas con un solo valor o secuencia:

In [99]:
```
import numpy as np
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[99]:

| | 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 [100]:
```
# Una sola etiqueta
# data.two
data['two']
```

Out[100]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

In [101]:
```
# Un conjunto de etiquetas
data[['three', 'one']]
```

Out[101]:

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

Esta indexación tiene algunos casos especiales. Primero, al pasar un solo elemento o una lista al operador `[]`, selecciona las columnas.

In [102]:
```
# Un objeto de división con enteros
data[:2]
```

Out[102]:

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

In [103]:
```
# Una matriz booleana
data[data < 5]
```

Out[103]:

| | one | two | three | four |
|---|---|---|---|---|
| Ohio | 0.0 | 1.0 | 2.0 | 3.0 |
| Colorado | 4.0 | NaN | NaN | NaN |
| Utah | NaN | NaN | NaN | NaN |
| New York | NaN | NaN | NaN | NaN |

La selección de valores de una serie con un vector booleano generalmente devuelve un subconjunto de los datos. Para garantizar que la salida de selección tenga la misma forma que los datos originales, puede usar el método `where` en Series y DataFrame.

Para devolver solo las filas seleccionadas:

In [104]:
```
data[data['three'] > 5]
```

Out[104]:

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

Para devolver una serie de la misma forma que el original:

In [105]:
```
data.where(data['three'] > 5)
```

Out[105]:

| | one | two | three | four |
|---|---|---|---|---|
| Ohio | NaN | NaN | NaN | NaN |
| Colorado | 4.0 | 5.0 | 6.0 | 7.0 |
| Utah | 8.0 | 9.0 | 10.0 | 11.0 |
| New York | 12.0 | 13.0 | 14.0 | 15.0 |

Además, `where` toma un otro argumento opcional para la sustitución de valores donde la condición es `False` , en la copia devuelta.

In [106]:
```
data.where(data['three'] > 5, -1)
```

Out[106]:

| | one | two | three | four |
|---|---|---|---|---|
| Ohio | -1 | -1 | -1 | -1 |
| Colorado | 4 | 5 | 6 | 7 |
| Utah | 8 | 9 | 10 | 11 |
| New York | 12 | 13 | 14 | 15 |

Es posible que desee establecer valores basados en algunos criterios booleanos. Esto se puede hacer de manera intuitiva así:

In [107]:
```
data[data < 5] = 0
data
```

Out[107]:

| | 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 |

Por defecto, `where` devuelve una copia modificada de los datos. Hay un parámetro opcional `inplace` los datos originales puedan modificarse sin crear una copia:

In [108]:
```
data.where(data > 5, -1, inplace=True)
data
```

Out[108]:

| | one | two | three | four |
|---|---|---|---|---|
| Ohio | -1 | -1 | -1 | -1 |
| Colorado | -1 | -1 | 6 | 7 |
| Utah | 8 | 9 | 10 | 11 |
| New York | 12 | 13 | 14 | 15 |

### Selección con `loc` e `iloc`

Estas convenciones de segmentación e indización pueden ser una fuente de confusión. Por ejemplo, si la serie tiene un índice entero explícito, una operación de indexación como `data[1]` utilizará los índices explícitos, mientras que una operación de segmentación como `data[1 : 3]` usará el índice implícito del estilo de Python.

```
In [109]: other_data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
          other_data
```

```
Out[109]: 1    a
          3    b
          5    c
          dtype: object
```

```
In [110]: # índice explícito
          other_data[1]
```

```
Out[110]: 'a'
```

```
In [111]: # índice implícito
          other_data[1:3]
```

```
Out[111]: 3    b
          5    c
          dtype: object
```

Debido a esta posible confusión en el caso de los índices enteros, Pandas proporciona algunos atributos de indexación especiales. Primero, el atributo `loc` permite indexar y segmentar con referencia al índice explícito.

**`.loc`** se basa principalmente en etiquetas, pero también se puede utilizar con una matriz booleana. `.loc` generará `KeyError` cuando no se encuentren los elementos. Las entradas permitidas son:

```
In [112]: other_data.loc[1]
```

```
Out[112]: 'a'
```

```
In [113]: other_data.loc[1:3]
```

```
Out[113]: 1    a
          3    b
          dtype: object
```

In [114]:
```
data.loc[lambda df: df.two > 5, :]
```

Out[114]:

| | one | two | three | four |
|---|---|---|---|---|
| Utah | 8 | 9 | 10 | 11 |
| New York | 12 | 13 | 14 | 15 |

El atributo `iloc` permite indexar y segmentar con referencia al índice implícito del estilo de Python.

**`.iloc`** se basa principalmente en la posición de enteros (de 0 a longitud-1 del eje), pero también se puede usar con una matriz booleana. `.iloc` generará `IndexError` si un indizador solicitado está fuera de los límites, excepto los indizadores de segmentos que permiten la indexación fuera de los límites. (Esto se ajusta a la semántica de segmentos de Python/NumPy).

In [115]:
```
other_data.iloc[1]
```

Out[115]:
```
'b'
```

In [116]:
```
other_data.iloc[1:3]
```

Out[116]:
```
3    b
5    c
dtype: object
```

In [117]:
```
# Selecciona todas las filas y dos columnas
data.iloc[:, [0, 1]]
```

Out[117]:

| | one | two |
|---|---|---|
| Ohio | -1 | -1 |
| Colorado | -1 | -1 |
| Utah | 8 | 9 |
| New York | 12 | 13 |

Adicionalmente `loc` permite la utilización de etiquetas, mientras que `iloc` sólo permite enteros:

In [118]:
```
# Selecciona una fila y varias columnas
data.loc['Colorado', ['two', 'three']]
```

Out[118]:
```
two     -1
three    6
Name: Colorado, dtype: int64
```

In [119]:
```
data.iloc[2, [3, 0, 1]]
```

Out[119]:
```
four    11
one      8
two      9
Name: Utah, dtype: int64
```

In [120]:
```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[120]:

| | four | one | two |
|---|---|---|---|
| Colorado | 7 | -1 | -1 |
| Utah | 11 | 8 | 9 |

Ambas funciones de indexación funcionan con segmentos, además de etiquetas individuales o listas de etiquetas:

In [121]:
```
data.loc[:'Utah', 'two']
```

Out[121]:
```
Ohio        -1
Colorado    -1
Utah         9
Name: two, dtype: int64
```

In [122]:
```
data.iloc[:, :3][data.three > 5]
```

Out[122]:

| | one | two | three |
|---|---|---|---|
| Colorado | -1 | -1 | 6 |
| Utah | 8 | 9 | 10 |
| New York | 12 | 13 | 14 |

In [123]:
```
data.one.loc[lambda s: s > 4]
```

Out[123]:
```
Utah          8
New York     12
Name: one, dtype: int64
```

Dado que la indexación con `[]` debe manejar muchos casos (acceso de etiqueta única, segmentación, indexación booleana, etc.), tiene un poco de sobrecarga para poder averiguar lo que está pidiendo. Si solo desea acceder a un valor escalar, la forma más rápida es utilizar los métodos `at` e `iat`, que se implementan en todas las estructuras de datos.

De forma similar a `loc`, `at` proporciona búsquedas escalares basadas en etiquetas, mientras que `iat` proporciona búsquedas basadas en enteros de manera análoga a `iloc`.

In [124]:
```
data.at['Utah', 'two']
```

Out[124]: 9

In [125]:
```
data.iat[2, 1]
```

Out[125]: 9

Por último, los objetos DataFrame tienen un método `query` que permite la selección usando una expresión.

In [126]:
```
# equivalente a data[data.one<data.two & data.two<data.three]
data.query('one<two & two<three')
```

Out[126]:

| | one | two | three | four |
|---|---|---|---|---|
| Utah | 8 | 9 | 10 | 11 |
| New York | 12 | 13 | 14 | 15 |

In [127]:
```
# equivalente a data[data.one.isin(data.two)]
data.query('one in two')
```

Out[127]:

| | one | two | three | four |
|---|---|---|---|---|
| Ohio | -1 | -1 | -1 | -1 |
| Colorado | -1 | -1 | 6 | 7 |

> **Nota**: DataFrame `query` usando expresiones numéricas es ligeramente más rápido que Python para marcos de datos grandes.

La siguiente tabla muestra algunas de los modos de indexación de un DataFrame:

| Tipo | Descripción |
|---|---|
| df[val] | Selecciona una sola columna o secuencia de columnas del DataFrame |
| df.loc[val] | Selecciona una sola fila o un subconjunto de filas del DataFrame por etiqueta |
| df.loc[:, val] | Selecciona una sola columna o subconjunto de columnas por etiqueta |
| df.loc[val1, val2] | Seleccione ambas filas y columnas por etiqueta |
| df.iloc[where] | Selecciona una sola fila o un subconjunto de filas del marco de datos por posición entera |
| df.iloc[:, where] | Selecciona una sola columna o un subconjunto de columnas por posición entera |
| df.iloc[where_i, where_j] | Selecciona tanto filas como columnas por posición entera |
| df.at[label_i, label_j] | Seleccione un solo valor escalar por fila y columna etiquetada |
| df.iat[i, j] | Seleccione un único valor escalar por fila y posición de columna (enteros) |
| reindex method | Seleccione filas o columnas etiquetadas |
| get_value, set_value methods | Seleccione un solo valor por fila y columna etiquetada |

### Indexación avanzada con índice jerárquico

La integración sintáctica de `MultiIndex` en la indexación avanzada con `.loc` es complicada. En general, las claves `MultiIndex` toman la forma de tuplas. Por ejemplo, lo siguiente funciona como cabría esperar:

In [128]:
```
arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo',
'qux', 'qux']),
          np.array(['one', 'two', 'one', 'two', 'one', 'two',
'one', 'two'])]
df = pd.DataFrame(np.random.randn(8, 3), index=arrays, column
s=['A', 'B', 'C'])
df
```

Out[128]:

| | | A | B | C |
|---|---|---|---|---|
| bar | one | -0.994393 | 0.149102 | 1.576706 |
| | two | -0.164314 | 0.560221 | 1.226952 |
| baz | one | 0.107101 | -0.512716 | 0.302121 |
| | two | 0.716107 | 1.112374 | 0.934282 |
| foo | one | 0.883358 | -1.012106 | -0.896801 |
| | two | 0.079051 | 1.725177 | -1.519495 |
| qux | one | 0.045276 | -0.353687 | -0.859885 |
| | two | -0.244868 | 0.606489 | -1.697922 |

In [129]:
```
# df.loc ['bar', 'two'] también funcionaría, pero esta notación puede llevar a la ambigüedad en general.
df.loc[('bar', 'two')]
```

Out[129]:
```
A   -0.164314
B    0.560221
C    1.226952
Name: (bar, two), dtype: float64
```

Si se desea indexar una columna específica con `.loc` , se debe usar una tupla como:

In [130]:
```
df.loc[('bar', 'two'), 'A']
```

Out[130]:
```
-0.16431411923857847
```

In [131]:
```
# equivalente a df.loc[('bar',),]
df.loc['bar']
```

Out[131]:

| | A | B | C |
|---|---|---|---|
| one | -0.994393 | 0.149102 | 1.576706 |
| two | -0.164314 | 0.560221 | 1.226952 |

In [132]:
```
df.loc['baz':'foo']
```

Out[132]:

| | | A | B | C |
|---|---|---|---|---|
| baz | one | 0.107101 | -0.512716 | 0.302121 |
| | two | 0.716107 | 1.112374 | 0.934282 |
| foo | one | 0.883358 | -1.012106 | -0.896801 |
| | two | 0.079051 | 1.725177 | -1.519495 |

In [133]:
```
df.loc[('baz', 'two'):('qux', 'one')]
```

Out[133]:

| | | A | B | C |
|---|---|---|---|---|
| baz | two | 0.716107 | 1.112374 | 0.934282 |
| foo | one | 0.883358 | -1.012106 | -0.896801 |
| | two | 0.079051 | 1.725177 | -1.519495 |
| qux | one | 0.045276 | -0.353687 | -0.859885 |

In [134]:
```
df.loc[('baz', 'two'):'foo']
```

Out[134]:

| | | A | B | C |
|---|---|---|---|---|
| baz | two | 0.716107 | 1.112374 | 0.934282 |
| foo | one | 0.883358 | -1.012106 | -0.896801 |
| | two | 0.079051 | 1.725177 | -1.519495 |

In [135]:
```
df.loc[[('bar', 'two'), ('qux', 'one')]]
```

Out[135]:

| | | A | B | C |
|---|---|---|---|---|
| bar | two | -0.164314 | 0.560221 | 1.226952 |
| qux | one | 0.045276 | -0.353687 | -0.859885 |

Por último, el método `xs` de DataFrame con su argumento `level` facilita la selección de datos a un nivel particular de un MultiIndex.

In [136]:
```
df.xs('one', level=1)
```

Out[136]:

| | A | B | C |
|---|---|---|---|
| bar | -0.994393 | 0.149102 | 1.576706 |
| baz | 0.107101 | -0.512716 | 0.302121 |
| foo | 0.883358 | -1.012106 | -0.896801 |
| qux | 0.045276 | -0.353687 | -0.859885 |

## Aritmetica y Alineación de datos

Una característica importante de pandas para algunas aplicaciones es el comportamiento de la aritmética entre objetos con índices diferentes. Cuando se agregan objetos, si algún par de índices no es el mismo, el índice respectivo en el resultado será la unión de los pares de índices. Para los usuarios con experiencia en bases de datos, esto es similar a una combinación externa automática en las etiquetas de índice.

In [137]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', '
e'])
s1
```

Out[137]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [138]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e
', 'f', 'g'])
s2
```

Out[138]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [139]:
```
s1 + s2
```

Out[139]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

La alineación de datos internos introduce valores `NA` en las ubicaciones de las etiquetas que no se superponen. Los valores `NA` se propagarán en otros cálculos aritméticos.

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

In [140]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=lis
t('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df1
```

Out[140]:

| | 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 [141]:
```
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=li
st('bde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df2
```

Out[141]:

| | 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 [142]:
```
df1 + df2
```

Out[142]:

| | 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 |

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

In [143]:
```
df1 = pd.DataFrame({'A': [1, 2]})
df1
```

Out[143]:

| | A |
|---|---|
| 0 | 1 |
| 1 | 2 |

In [144]:
```
df2 = pd.DataFrame({'B': [3, 4]})
df2
```

Out[144]:

| | B |
|---|---|
| 0 | 3 |
| 1 | 4 |

In [145]:
```
df1 - df2
```

Out[145]:

| | A | B |
|---|---|---|
| 0 | NaN | NaN |
| 1 | NaN | NaN |

### Métodos aritméticos con rellenado de valores

En las operaciones aritméticas entre objetos indexados de forma diferente, es posible que desee rellenar con un valor especial, como 0, cuando se encuentra una etiqueta de eje en un objeto pero no en el otro:

In [146]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=li
st('abcd'))
df1
```

Out[146]:

| | 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 [147]:
```
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=li
st('abcde'))
df2.loc[1, 'b'] = np.nan
df2
```

Out[147]:

| | a | b | c | d | e |
|---|---|---|---|---|---|
| 0 | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 |
| 1 | 5.0 | NaN | 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 [148]:
```
df1 + df2
```

Out[148]:

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

In [149]:
```
df2.add(df1, fill_value=0)
```

Out[149]:

| | a | b | c | d | e |
|---|---|---|---|---|---|
| 0 | 0.0 | 2.0 | 4.0 | 6.0 | 4.0 |
| 1 | 9.0 | 5.0 | 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 |

La siguiente tabla muestra diferentes métodos aritméticos flexibles. Cada uno de ellos tiene una contraparte, comenzando con la letra r, que tiene argumentos invertidos.

| Método | Descripción |
|---|---|
| add, radd | Métodos de suma (+) |
| sub, rsub | Métodos de resta (-) |
| div, rdiv | Métodos de división (/) |
| floordiv, rfloordiv | Métodos de floor división (//) |
| mul, rmul | Métodos de multiplicación (*) |
| pow, rpow | Métodos de exponenciación (**) |

De manera relacionada, al reindexar una serie o un marco de datos, también se puede especificar un valor de relleno diferente:

In [150]:
```
df1.reindex(columns=df2.columns, fill_value=0)
```

Out[150]:

| | 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 |

### Operaciones entre DataFrame y Series

De igual forma que Numpy permitía operaciones entre arrays de diferentes dimensiones mediante la técnica de difusión (*broadcasting*), las operaciones entre DataFrames y Series son similares:

In [151]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```

Out[151]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [152]:
```
arr[0]
```

Out[152]:
```
array([0., 1., 2., 3.])
```

In [153]:
```
arr - arr[0]
```

Out[153]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

In [154]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon
'])
frame
```

Out[154]:

| | 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 [155]:
```
series = frame.iloc[0]
series
```

Out[155]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

De forma predeterminada, la aritmética entre el DataFrame y la serie coincide con el índice de la serie en las columnas del marco de datos, transmitiéndose por las filas:

In [156]:
```
frame - series
```

Out[156]:

| | 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 en las columnas del marco de datos o en el índice de la serie, los objetos se volverán a indexar para formar la unión:

In [157]:
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
frame + series2
```

Out[157]:

| | 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 se desea difundir sobre las columnas, haciendo coincidir las filas, se debe utilizar uno de los métodos aritméticos. Por ejemplo:

In [158]:
```
series3 = frame['d']
series3
```

Out[158]:
```
Utah     1.0
Ohio     4.0
Texas    7.0
Oregon   10.0
Name: d, dtype: float64
```

In [159]:
```
frame
```

Out[159]:

| | 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 [160]:
```
frame.sub(series3, axis='index')
```

Out[160]:

| | 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 número de eje que se pasa es el eje con el que se desea coincidir. En este caso, queremos hacer coincidir en el índice de la fila del DataFrame (axis = 'index' o axis = 0) y transmitir a través del mismo.

## Aplicación y mapeo de funciones

Las funciones universales Numpy (ufuncs) también funcionan sobre objetos pandas.

In [161]:
```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[161]:

| | b | d | e |
|---|---|---|---|
| Utah | 0.389281 | 0.224701 | -0.431611 |
| Ohio | -1.528661 | 1.077165 | 0.074643 |
| Texas | 1.380576 | 0.462639 | 1.814201 |
| Oregon | -0.955021 | -0.249938 | -0.377966 |

In [162]:
```
np.abs(frame)
```

Out[162]:

| | b | d | e |
|---|---|---|---|
| Utah | 0.389281 | 0.224701 | 0.431611 |
| Ohio | 1.528661 | 1.077165 | 0.074643 |
| Texas | 1.380576 | 0.462639 | 1.814201 |
| Oregon | 0.955021 | 0.249938 | 0.377966 |

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

In [163]:
```
f = lambda x: x.max() - x.min()
frame.apply(f)
```

Out[163]:
```
b    2.909237
d    1.327103
e    2.245811
dtype: float64
```

Aquí, la función `f` , que calcula la diferencia entre el máximo y el mínimo de una serie, se invoca una vez en cada columna en `frame` . El resultado es una Serie que tiene las columnas de `frame` como su índice.

Si pasa el `axis = 'columns'` a `apply` , la función se invocará una vez por fila:

In [164]:
```
frame.apply(f, axis='columns')
```

Out[164]:
```
Utah      0.820891
Ohio      2.605826
Texas     1.351562
Oregon    0.705083
dtype: float64
```

Muchas de las estadísticas de matriz más comunes (como suma y media) son métodos DataFrame, por lo que no es necesario usar `apply`. La función pasada a `apply` no necesita devolver un valor escalar, también puede devolver una serie con múltiples valores:

In [165]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max
'])

frame.apply(f)
```

Out[165]:

| | b | d | e |
|---|---|---|---|
| min | -1.528661 | -0.249938 | -0.431611 |
| max | 1.380576 | 1.077165 | 1.814201 |

También se pueden usar funciones de Python aplicables a elementos. Supongamos que desea obtener una cadena con formato de cada valor de punto flotante en `frame`. Se puede hacer esto usando `applymap`:

In [166]:
```
format = lambda x: '%.2f' % x
frame.applymap(format)
```

Out[166]:

| | b | d | e |
|---|---|---|---|
| Utah | 0.39 | 0.22 | -0.43 |
| Ohio | -1.53 | 1.08 | 0.07 |
| Texas | 1.38 | 0.46 | 1.81 |
| Oregon | -0.96 | -0.25 | -0.38 |

La razón para el nombre `applymap` es que las Series tiene un método `map` para aplicar una función de elementos:

In [167]:
```
frame['e'].map(format)
```

Out[167]:
```
Utah      -0.43
Ohio       0.07
Texas      1.81
Oregon    -0.38
Name: e, dtype: object
```

## Ordenación y clasificación

La clasificación de un conjunto de datos por algún criterio es otra operación incorporada importante. Para ordenar lexicográficamente por índice de fila o columna, se usa el método `sort_index` , que devuelve un nuevo objeto ordenado.

In [168]:
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()
```

Out[168]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

Los DataFrame pueden ordenarse por el índice o por los ejes. Los datos son ordenados de forma ascendente por defecto.

In [169]:
```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
frame.sort_index()
```

Out[169]:

| | d | a | b | c |
|---|---|---|---|---|
| one | 4 | 5 | 6 | 7 |
| three | 0 | 1 | 2 | 3 |

In [170]:
```
frame.sort_index(axis=1)
```

Out[170]:

| | a | b | c | d |
|---|---|---|---|---|
| three | 1 | 2 | 3 | 0 |
| one | 5 | 6 | 7 | 4 |

In [171]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[171]:

| | d | c | b | a |
|---|---|---|---|---|
| three | 0 | 3 | 2 | 1 |
| one | 4 | 7 | 6 | 5 |

Las series pueden ordenarse por sus valores haciendo uso del método `sort_values` :

In [172]:
```
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

Out[172]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

Los valors `NA` ( `NaN` ) se ordenan por defecto al final de las series:

In [173]:
```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

Out[173]:
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

Al ordenar un DataFrame, se pueden usar los datos en una o más columnas como claves de clasificación. Para hacerlo, se pasa uno o más nombres de columna a la opción `by` de `sort_values` :

In [174]:
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[174]:

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

In [175]:
```
frame.sort_values(by='b')
```

Out[175]:

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

In [176]:
```
frame.sort_values(by=['a', 'b'])
```

Out[176]:

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

La clasificación asigna rangos de uno a la cantidad de puntos de datos válidos en una matriz. Los métodos de clasificación ( `rank` ) para Series y DataFrame calculan rangos de datos numéricos ( `1 a N` ) a lo largo del eje. A los valores iguales se les asigna un rango que es el promedio de los rangos de esos valores.

In [177]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj.rank()
```

Out[177]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

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

In [178]:
```
obj.rank(method='first')
```

Out[178]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

In [179]:
```
# Asigna valores de empate al rango máximo en el grupo
obj.rank(ascending=False, method='max')
```

Out[179]:
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

Los DataFrame pueden definir rangos sobre filas y sobre columnas:

In [180]:
```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0,
1],
                      'c': [-2, 5, 8, -2.5]})
frame
```

Out[180]:

| | 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 [181]:
```
frame.rank(axis='columns')
```

Out[181]:

| | 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 |

La siguiente lista muestra los métodos para la generación de los rangos de clasificación:

| Métdodo | Descripción |
|---|---|
| 'average' | Predeterminado: asigna el rango promedio a cada entrada igual |
| 'min' | Utiliza el rango mínimo para todo el grupo |
| 'max' | Utiliza el rango máximo para todo el grupo |
| 'first' | Asignar rangos en el orden en que aparecen los valores en los datos |
| 'dense' | Como 'min', pero los rangos siempre aumentan en 1 entre los grupos en lugar del número de elementos iguales en un grupo |

## Índices sobre ejes con etiquetas duplicadas

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

In [182]:
```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
obj
```

Out[182]:
```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

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

In [183]:
```
obj.index.is_unique
```

Out[183]:
```
False
```

La selección de datos es una de las principales cosas que se comporta de manera diferente con los duplicados. La indexación de una etiqueta con varias entradas devuelve una Serie, mientras que las entradas individuales devuelven un valor escalar:

In [184]:
```
obj['a']
```

Out[184]:
```
a    0
a    1
dtype: int64
```

La misma lógica se extiende a la indexación de filas en un DataFrame:

In [185]:
```
df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b
', 'b'])
df
```

Out[185]:

| | 0 | 1 | 2 |
|---|---|---|---|
| a | -1.587674 | 1.230593 | 0.126846 |
| a | -1.431222 | -0.663194 | 0.322645 |
| b | 1.604719 | 0.809617 | -1.588775 |
| b | 1.664304 | 1.054497 | 0.750832 |

In [186]:
```
df.loc['b']
```

Out[186]:

| | 0 | 1 | 2 |
|---|---|---|---|
| b | 1.604719 | 0.809617 | -1.588775 |
| b | 1.664304 | 1.054497 | 0.750832 |

## Datos duplicados

Si desea identificar y eliminar filas duplicadas en un DataFrame, existen dos métodos que lo ayudarán: `duplicate` y `drop_duplicates` . Cada uno toma como argumento las columnas a usar para identificar filas duplicadas.

- `duplicates` devuelve un vector booleano cuya longitud es el número de filas y que indica si una fila está duplicada.
- `drop_duplicates` elimina las filas duplicadas.

De forma predeterminada, la primera fila observada de un conjunto duplicado se considera única, pero cada método tiene un parámetro de mantenimiento para especificar los objetivos que se mantendrán.

- `keep = 'first'` (predeterminado): marca / suelta duplicados excepto la primera vez que ocurre.
- `keep = 'last'` : marca / suelta duplicados excepto la última aparición.
- `keep = False` : marca / suelta todos los duplicados.

In [187]:
```
df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two',
'three', 'four'],
                    'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
                    'c': np.random.randn(7)})
df2
```

Out[187]:

| | a | b | c |
|---|---|---|---|
| 0 | one | x | 0.989015 |
| 1 | one | y | 1.003844 |
| 2 | two | x | 0.842918 |
| 3 | two | y | 0.438247 |
| 4 | two | x | -2.553005 |
| 5 | three | x | 0.384511 |
| 6 | four | x | -0.021962 |

In [188]:
```
df2.duplicated('a')
```

Out[188]:
```
0    False
1     True
2    False
3     True
4     True
5    False
6    False
dtype: bool
```

In [189]:
```
df2.duplicated(['a', 'b'])
```

Out[189]:
```
0    False
1    False
2    False
3    False
4     True
5    False
6    False
dtype: bool
```

In [190]:
```
df2.drop_duplicates('a', keep='last')
```

Out[190]:

| | a | b | c |
|---|---|---|---|
| 1 | one | y | 1.003844 |
| 4 | two | x | -2.553005 |
| 5 | three | x | 0.384511 |
| 6 | four | x | -0.021962 |

Para eliminar los duplicados por valor de índice, use `Index.duplicated` y luego realice el corte. El mismo conjunto de opciones está disponible para el parámetro `keep` .

In [191]:
```
df3 = pd.DataFrame({'a': np.arange(6),
                    'b': np.random.randn(6)},
                   index=['a', 'a', 'b', 'c', 'b', 'a'])
df3
```

Out[191]:

| | a | b |
|---|---|---|
| a | 0 | 2.208294 |
| a | 1 | -1.765099 |
| b | 2 | -0.887422 |
| c | 3 | 2.120559 |
| b | 4 | -0.357941 |
| a | 5 | 0.372641 |

In [192]:
```
df3.index.duplicated()
```

Out[192]:
```
array([False,  True, False, False,  True,  True])
```

In [193]:
```
df3[~df3.index.duplicated()]
```

Out[193]:

| | a | b |
|---|---|---|
| a | 0 | 2.208294 |
| b | 2 | -0.887422 |
| c | 3 | 2.120559 |

In [194]:
```
df3[~df3.index.duplicated(keep='last')]
```

Out[194]:

| | a | b |
|---|---|---|
| c | 3 | 2.120559 |
| b | 4 | -0.357941 |
| a | 5 | 0.372641 |

## Usando estadísticas descriptivas

Los objetos pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes. La mayoría de éstos pertenecen a la categoría de reducciones o estadísticas de resumen, es decir, métodos que extraen un solo valor (como la suma o la media) de una serie o de los valores de las filas o columnas de un DataFrame. En comparación con los métodos similares que se encuentran en las matrices NumPy, tienen un manejo integrado de los datos faltantes.

In [195]:
```
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[195]:

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

In [196]:
```
df.sum()
```

Out[196]:
```
one    9.25
two   -5.80
dtype: float64
```

In [197]:
```
df.sum(axis='columns')
```

Out[197]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

Los valores `NaN` se excluyen a menos que la sección completa (fila o columna en este caso) sea `NaN` . Esto se puede desactivar con la opción `skipna` :

In [198]:
```
df.mean(axis='columns', skipna=False)
```

Out[198]:
```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

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

In [199]:
```
df.idxmax()
```

Out[199]:
```
one    b
two    d
dtype: object
```

Otros métodos son acumulaciones:

In [200]:
```
df.cumsum()
```

Out[200]:

| | one | two |
|---|---|---|
| a | 1.40 | NaN |
| b | 8.50 | -4.5 |
| c | NaN | NaN |
| d | 9.25 | -5.8 |

Otros tipos de métodos no son una reducción ni una acumulación, `describe` es uno de esos ejemplos, que produce múltiples estadísticas de resumen en una sola petición:

In [201]:
```
df.describe()
```

Out[201]:

| | 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 |

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

In [202]:
```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj.describe()
```

Out[202]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

La siguiente lista muestra las estadísticas de resumen y descriptivas más habituales:

| Métdodo | Desccripción |
|---|---|
| count | Número de valores no `NaN` |
| describe | Conjunto de estadísticas de resumen para Series o cada columna DataFrame |
| min, max | Valores mínimos y máximos |
| argmin, argmax | Índices (enteros) en los que se obtuvo el valor mínimo o máximo, respectivamente |
| idxmin, idxmax | Índices (etiquetas) en los que se obtuvo el valor mínimo o máximo, respectivamente |
| quantile | cuantil de la muestra de 0 a 1 |
| sum | Suma de valores |
| mean | Media de valores |
| median | Media aritmética (50% cuantil) de valores |
| mad | Desviación absoluta media del valor medio |
| prod | Producto de todos los valores |
| var | Muestra la varianza de los valores |
| std | Muestra la desviación estándar de los valores |
| skew | Muestra la asimetría (tercer momento) de los valores |
| kurt | Muestra de curtosis (cuarto momento) de valores |
| cumsum | Suma acumulativa de valores |
| cummin, cummax | Mínimo o máximo acumulado de valores, respectivamente |
| cumprod | Producto acumulado de valores |
| diff | Calcula la primera diferencia aritmética (útil para series de tiempo) |
| pct_change | Calcula los cambios porcentuales |

## 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 de acciones y volúmenes obtenidos de `Yahoo! Financial` usando el paquete `pandas-datareader` adicional. Si aún no lo tiene instalado, puede obtenerlo a través de `conda` o `pip` :

```
conda install pandas-datareader
```

Utilizo el módulo pandas_datareader para descargar algunos datos de unos pocos tickers de stock:

In [2]:
```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker) for ticker in
['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close'] for ticker, d
ata in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume'] for ticker, dat
a in all_data.items()})
```

In [18]:
```
all_data
```

Out[18]:
{'AAPL':

| Date | | High | Low | Open | Close |
|---|---|---|---|---|---|
| 2015-09-18 | 28.575001 | 27.967501 | 28.052500 | 28.362499 | 297141200.0 |
| 2015-09-21 | 28.842501 | 28.415001 | 28.417500 | 28.802500 | 200888000.0 |
| 2015-09-22 | 28.545000 | 28.129999 | 28.344999 | 28.350000 | 201384800.0 |
| 2015-09-23 | 28.680000 | 28.325001 | 28.407499 | 28.580000 | 143026800.0 |
| 2015-09-24 | 28.875000 | 28.092501 | 28.312500 | 28.750000 | 200878000.0 |
| 2015-09-25 | 29.172501 | 28.504999 | 29.110001 | 28.677500 | 224607600.0 |
| 2015-09-28 | 28.642500 | 28.110001 | 28.462500 | 28.110001 | 208436000.0 |
| 2015-09-29 | 28.377501 | 26.965000 | 28.207500 | 27.264999 | 293461600.0 |
| 2015-09-30 | 27.885000 | 27.182501 | 27.542500 | 27.575001 | 265892000.0 |
| 2015-10-01 | 27.405001 | 26.827499 | 27.267500 | 27.395000 | 255716400.0 |
| 2015-10-02 | 27.752501 | 26.887501 | 27.002501 | 27.594999 | 232079200.0 |
| 2015-10-05 | 27.842501 | 27.267500 | 27.469999 | 27.695000 | 208258800.0 |
| 2015-10-06 | 27.934999 | 27.442499 | 27.657499 | 27.827499 | 192787200.0 |
| 2015-10-07 | 27.942499 | 27.352501 | 27.934999 | 27.695000 | 187062400.0 |
| 2015-10-08 | 27.547501 | 27.052500 | 27.547501 | 27.375000 | 247918400.0 |

In [19]:
```
price
```

Out[19]:

| Date | AAPL | IBM | MSFT | GOOG |
|---|---|---|---|---|
| **2015-** | 26.210186 | 115.723709 | 39.514427 | 629.250000 |
| **2015-** | 26.616798 | 117.301270 | 40.086960 | 635.440002 |
| **2015-** | 26.198639 | 115.659622 | 39.896111 | 622.690002 |
| **2015-** | 26.411180 | 115.042992 | 39.868851 | 622.359985 |
| **2015-** | 26.568277 | 115.643631 | 39.905201 | 625.799988 |
| **2015-** | 26.501280 | 116.452400 | 39.932461 | 611.969971 |
| **2015-** | 25.976849 | 114.130112 | 39.341743 | 594.890015 |
| **2015-** | 25.195972 | 114.090080 | 39.478069 | 594.969971 |
| **2015-** | 25.482449 | 116.092072 | 40.223278 | 608.419983 |
| **2015-** | 25.316107 | 114.986984 | 40.541351 | 611.289978 |
| **2015-** | 25.500929 | 115.779770 | 41.413811 | 626.909973 |
| **2015-** | 25.593342 | 119.351326 | 42.377129 | 641.469971 |
| **2015-** | 25.715784 | 119.143120 | 42.486176 | 645.440002 |
| **2015-** | 25.593342 | 120.192162 | 42.531616 | 642.359985 |
| **2015-** | 25.297623 | 121.945938 | 43.122341 | 639.159973 |
| **2015-** | 25.902922 | 122.034019 | 42.813358 | 643.609985 |
| **2015-** | 25.782785 | 121.033043 | 42.713379 | 646.669983 |
| **2015-** | 25.826679 | 119.815773 | 42.613407 | 652.299988 |

In [3]:
```
returns = price.pct_change()
returns.tail()
```

Out[3]:

| Date | AAPL | IBM | MSFT | GOOG |
|---|---|---|---|---|
| **2020-09-10** | -0.032646 | -0.013905 | -0.028018 | -0.016018 |
| **2020-09-11** | -0.013129 | 0.007465 | -0.006525 | -0.007376 |
| **2020-09-14** | 0.030000 | 0.005187 | 0.006764 | -0.000947 |
| **2020-09-15** | 0.001560 | 0.002867 | 0.016406 | 0.014586 |
| **2020-09-16** | -0.015190 | 0.018442 | -0.001150 | 0.001935 |

El método `corr` de Series calcula la correlación de los valores superpuestos, no `NaN`, alineados por índice en dos series. El método `cov` calcula la covarianza:

In [4]:
```
# returns.MSFT.corr(returns.IBM)
returns['MSFT'].corr(returns['IBM'])
```

Out[4]:
```
0.5773030677290131
```

In [5]:
```
# returns.MSFT.cov(returns.IBM)
returns['MSFT'].cov(returns['IBM'])
```

Out[5]:
```
0.00016163139660133407
```

El método `corr` tiene un parámetro `method` que permite indicar el método para calcular las correlaciones: `pearson` (por defecto), `kendall` o `spearman`.

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

In [6]:
```
returns.corr()
```

Out[6]:

| | AAPL | IBM | MSFT | GOOG |
|---|---|---|---|---|
| **AAPL** | 1.000000 | 0.499397 | 0.703332 | 0.654536 |
| **IBM** | 0.499397 | 1.000000 | 0.577303 | 0.535776 |
| **MSFT** | 0.703332 | 0.577303 | 1.000000 | 0.782889 |
| **GOOG** | 0.654536 | 0.535776 | 0.782889 | 1.000000 |

```
In [7]: returns.cov()
```

Out[7]:

| | AAPL | IBM | MSFT | GOOG |
|---|---|---|---|---|
| **AAPL** | 0.000346 | 0.000149 | 0.000229 | 0.000201 |
| **IBM** | 0.000149 | 0.000257 | 0.000162 | 0.000142 |
| **MSFT** | 0.000229 | 0.000162 | 0.000305 | 0.000226 |
| **GOOG** | 0.000201 | 0.000142 | 0.000226 | 0.000272 |

Usando el método `corrwith` de DataFrame, se 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 [8]: returns.corrwith(returns.IBM)
```

```
Out[8]: AAPL    0.499397
        IBM     1.000000
        MSFT    0.577303
        GOOG    0.535776
        dtype: float64
```

Al pasar un DataFrame se calculan las correlaciones de los nombres de columna coincidentes.

```
In [9]: returns.corrwith(volume)
```

```
Out[9]: AAPL   -0.077392
        IBM    -0.098493
        MSFT   -0.053432
        GOOG   -0.150092
        dtype: float64
```

Al pasar `axis = 'columns'` aplica los métodos fila por fila. En todos los casos, los puntos de datos se alinean por etiqueta antes de calcular la correlación.

## Valores únicos, recuentos de valor y membresía

Otra clase de métodos extraen información sobre los valores contenidos en una Serie unidimensional. Por ejemplo, `unique` , devuelve una matriz con los valores únicos en una serie. Los valores únicos no necesariamente se devuelven ordenados, pero podrían ordenarse si fuera necesario ( `uniques.sort()` ).

In [10]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c
'])
obj
```

Out[10]:
```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [11]:
```
uniques = obj.unique()
uniques
```

Out[11]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

El método `value_counts` devuelve una serie que contiene frecuencias de valores:

In [12]:
```
obj.value_counts()
```

Out[12]:
```
a    3
c    3
b    2
d    1
dtype: int64
```

Las series están ordenadas por valor en orden descendente por defecto, `value_counts` también está disponible como un método de pandas de nivel superior que se puede usar con cualquier matriz o secuencia:

In [13]:
```
pd.value_counts(obj.values, sort=False)
```

Out[13]:
```
b    2
d    1
c    3
a    3
dtype: int64
```

El método `isin` realiza una verificación de membresía del 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 [14]:
```
obj
```

Out[14]:
```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [15]:
```
mask = obj.isin(['b', 'c'])
mask
```

Out[15]:
```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

In [16]:
```
obj[mask]
```

Out[16]:
```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Relacionado con `isin` está el método `Index.get_indexer` , que proporciona una matriz de índices resultado de aplicar elementos comunes en las series:

In [17]:
```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])
pd.Index(unique_vals).get_indexer(to_match)
```

Out[17]:
```
array([0, 2, 1, 1, 0, 2])
```