In [50]:
```
import numpy as np
import pandas as pd
PREVIOUS_MAX_ROWS = pd.options.display.max_rows
pd.options.display.max_columns = 20
pd.options.display.max_rows = 20
pd.options.display.max_colwidth = 80
np.random.seed(12345)
import matplotlib.pyplot as plt
plt.rc("figure", figsize=(10, 6))
np.set_printoptions(precision=4, suppress=True)
```

4 Agregación de datos y operaciones de grupo

Categorizar un conjunto de datos y aplicar una función a cada grupo, ya sea una agregación o transformación, puede ser un componente crítico de un flujo de trabajo de análisis de datos. Después de cargar, fusionar y preparar un conjunto de datos, es posible que necesite calcular estadísticas de grupo o posiblemente tablas dinámicas (pivot tables) para fines de generación de informes o visualización. pandas proporciona una interfaz groupby versátil, lo que le permite cortar (slice), dividir (dice) y resumir (summarize) conjuntos de datos (datasets) de una manera natural.

Una de las razones de la popularidad de las bases de datos relacionales y SQL es la facilidad con la que los datos se pueden unir (join), filtrar, transformar y agregar. Sin embargo, los lenguajes de consulta como SQL imponen ciertas limitaciones a los tipos de operaciones de grupo que pueden realizarse. Como verás, con la expresividad de Python y pandas, podemos realizar operaciones de grupo bastante complejas expresándolas como funciones personalizadas de Python que manipulan los datos asociados a cada grupo. Entre los cuales se destaca:

- Dividir (split) un objeto pandas en trozos utilizando una o más claves (en forma de funciones, arrays o nombres de columnas DataFrame).
- Calcular estadísticas de resumen de grupo, como recuento (count), media (mean) o desviación estándar, o una función definida por el usuario
- Aplicar transformaciones dentro del grupo u otras manipulaciones, como normalización, regresión lineal, rango o selección de subconjuntos.
- Calcular tablas dinámicas (Pivot tables) y tabulaciones cruzadas (cross-tabulations: crosstab)
- Realizar análisis de cuantiles y otros análisis estadísticos de grupos

In [51]:
```
import numpy as np
import pandas as pd
```

4.1 Cómo pensar en las operaciones de grupo

Hadley Wickham, autor de muchos paquetes populares para el lenguaje de programación R, acuñó el término dividir-aplicar-combinar (**split-apply-combine**) para describir las operaciones de grupo.

En la primera etapa del proceso, los datos contenidos en un objeto pandas, ya sea una Serie, DataFrame u otro, se dividen en grupos basados en una o más claves (keys) que el analista proporciona. La división (split)

se realiza en un eje particular de un objeto. Por ejemplo, un DataFrame se puede agrupar en sus filas (axis="index") o en sus columnas (axis="columns").

Una vez hecho esto, se aplica (apply) una función a cada grupo, produciendo un nuevo valor.

Por último, los resultados de todas esas aplicaciones de funciones se combinan (combine) en un objeto resultado. La forma del objeto resultante dependerá normalmente de lo que se haga con los datos. En la Figura 4.1 se muestra un modelo de agregación de grupos simple.

Cada clave de agrupación puede adoptar muchas formas, y las claves no tienen por qué ser todas del mismo tipo:

- Una lista o array de valores de la misma longitud que el eje que se está agrupando.
- Un valor que indica el nombre de una columna en un DataFrame
- Un diccionario o serie que indique la correspondencia entre los valores del eje que se está agrupando y los nombres de los grupos
- Una función que se invocará en el índice del eje o en las etiquetas individuales del índice

Tenga en cuenta que los tres últimos métodos son atajos para producir un array de valores que se utilizará para dividir el objeto. Para empezar, supongamos un pequeño conjunto de datos tabulares como un DataFrame:

In [52]:
```
df = pd.DataFrame({"key1" : ["a", "a", None, "b", "b", "a", None],
                   "key2" : pd.Series([1, 2, 1, 2, 1, None, 1],
                                      dtype="Int64"),
                   "data1" : np.random.standard_normal(7),
                   "data2" : np.random.standard_normal(7)})
df
```

Out[52]:

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

Supongamos que desea calcular la media de la columna `data1` utilizando las etiquetas de `key1`. Hay varias formas de hacerlo. Una es acceder a data1 y llamar a `groupby` con la columna (una Serie) en key1:

In [53]:
```
grouped = df["data1"].groupby(df["key1"])
grouped
```

Out[53]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001D5604C8160>
```

Esta variable agrupada es ahora un objeto especial "GroupBy". En realidad, aún no ha calculado nada, salvo algunos datos intermedios sobre la clave de grupo df["key1"]. La idea es que este objeto tenga toda la información necesaria para aplicar alguna operación a cada uno de los grupos. Por ejemplo, para calcular las medias de los grupos podemos llamar al método de la media de GroupBy:

In [54]:
```
grouped.mean()
```

Out[54]:
```
key1
a    0.555881
b    0.705025
Name: data1, dtype: float64
```

Lo importante aquí es que los datos (una Serie) han sido agregados dividiendo los datos en la key de grupo, produciendo una nueva Serie que ahora está indexada por los valores únicos de la columna key1. El índice resultante tiene el nombre "key1" porque la columna DataFrame df["key"] lo tenía.

Si en su lugar hubiéramos pasado múltiples arrays como una lista, obtendríamos algo diferente:

In [55]:
```
means = df["data1"].groupby([df["key1"], df["key2"]]).mean()
means
```

Out[55]:
```
key1  key2
a     1      -0.204708
      2       0.478943
b     1       1.965781
      2      -0.555730
Name: data1, dtype: float64
```

Aquí agrupamos los datos utilizando dos keys, y la Serie resultante tiene ahora un índice jerárquico formado por los pares únicos de claves observados:

In [56]:
```
means.unstack()
```

Out[56]:

key2	**1**	**2**
key1		
a	-0.204708	0.478943
b	1.965781	-0.555730

En este ejemplo, las claves (keys) de grupo son todas Series, aunque podrían ser cualquier array de la longitud adecuada:

In [57]:
```
states = np.array(["OH", "CA", "CA", "OH", "OH", "CA", "OH"])

years = [2005, 2005, 2006, 2005, 2006, 2005, 2006]

df["data1"].groupby([states, years]).mean()
```

Out[57]:
```
CA  2005    0.936175
    2006   -0.519439
OH  2005   -0.380219
    2006    1.029344
Name: data1, dtype: float64
```

Con frecuencia, la información de agrupación se encuentra en el mismo DataFrame que los datos con los que se desea trabajar. En ese caso, puedes pasar nombres de columnas (ya sean cadenas, números u otros objetos Python) como claves de grupo:

In [58]:
```
df.groupby("key1").mean()
```

Out[58]:

key1	key2	data1	data2
a	1.5	0.555881	0.441920
b	1.5	0.705025	-0.144516

In [59]:
```
df.groupby("key2").mean(numeric_only=True)
```

Out[59]:

key2	data1	data2
1	0.333636	0.115218
2	-0.038393	0.888106

In [60]:
```
df.groupby(["key1", "key2"]).mean()
```

Out[60]:

key1	key2	data1	data2
a	1	-0.204708	0.281746
	2	0.478943	0.769023
b	1	1.965781	-1.296221
	2	-0.555730	1.007189

Puede observar que en el segundo caso, es necesario pasar numeric_only=True porque la columna key1 no es numérica y por lo tanto no se puede agregar con mean().

Independientemente del objetivo que se persiga al utilizar groupby, un método GroupBy generalmente útil es `size`, que devuelve una Serie que contiene los tamaños de los grupos:

In [61]:
```
df.groupby(["key1", "key2"]).size()
```

Out[61]:
```
key1  key2
a     1       1
      2       1
b     1       1
      2       1
dtype: int64
```

Tenga en cuenta que los valores que faltan en una clave de grupo se excluyen del resultado por defecto. Este comportamiento puede desactivarse pasando `dropna=False` a `groupby`:

In [62]:
```
df.groupby("key1", dropna=False).size()
```

Out[62]:
```
key1
a      3
b      2
NaN    2
dtype: int64
```

In [63]:
```
df.groupby(["key1", "key2"], dropna=False).size()
```

```
key1  key2
```

Out[63]:
```
a     1       1
      2       1
      <NA>    1
b     1       1
      2       1
NaN   1       2
dtype: int64
```

Una función de grupo similar a size es `count`, que calcula el número de valores no nulos en cada grupo:

In [64]:
```
df.groupby("key1").count()
```

Out[64]:

key1	key2	data1	data2
a	2	3	3
b	2	2	2

Iteración sobre grupos

El objeto devuelto por groupby admite la iteración, generando una secuencia de 2-tuplas que contienen el nombre del grupo junto con el trozo (chunk) de datos. Considere lo siguiente:

In [65]:
```
for name, group in df.groupby("key1"):
    print(name)
    print(group)
# , itera sobre cada grupo y muestra
# el nombre del grupo y los datos
# correspondientes a ese grupo.
```

```
a
  key1  key2     data1     data2
0    a     1 -0.204708  0.281746
1    a     2  0.478943  0.769023
5    a  <NA>  1.393406  0.274992
b
  key1  key2     data1     data2
3    b     2 -0.555730  1.007189
4    b     1  1.965781 -1.296221
```

En el caso de múltiples claves, el primer elemento de la tupla será una tupla de valores clave:

In [66]:
```
for (k1, k2), group in df.groupby(["key1", "key2"]):
    print((k1, k2))
    print(group)
```

```
('a', 1)
  key1  key2     data1     data2
0    a     1 -0.204708  0.281746
('a', 2)
  key1  key2     data1     data2
1    a     2  0.478943  0.769023
('b', 1)
  key1  key2     data1     data2
4    b     1  1.965781 -1.296221
('b', 2)
  key1  key2    data1     data2
3    b     2 -0.55573  1.007189
```

Por supuesto, puedes elegir hacer lo que quieras con las piezas de datos. Una receta que puede resultarte

útil es calcular un diccionario de las piezas de datos como una sola línea:

In [67]:
```
pieces = {name: group for name, group in df.groupby("key1")}
pieces
```

Out[67]:
```
{'a':   key1  key2     data1     data2
 0    a     1 -0.204708  0.281746
 1    a     2  0.478943  0.769023
 5    a  <NA>  1.393406  0.274992,
 'b':   key1  key2     data1     data2
 3    b     2 -0.555730  1.007189
 4    b     1  1.965781 -1.296221}
```

In [68]:
```
pieces["b"]
```

Out[68]:

	key1	key2	data1	data2
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221

Por defecto groupby agrupa en axis="index", pero puedes agrupar en cualquiera de los otros ejes. Por ejemplo, podríamos agrupar las columnas de nuestro ejemplo df aquí por si empiezan por "key" o "data":

In [69]:
```
grouped = df.groupby({"key1": "key", "key2": "key",
                      "data1": "data", "data2": "data"}, axis="columns")
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\1884927282.py:1: FutureWarning: DataFr
ame.groupby with axis=1 is deprecated. Do `frame.T.groupby(...)` without axis instead.
  grouped = df.groupby({"key1": "key", "key2": "key",
```

Podemos imprimir los grupos así:

In [70]:
```
for group_key, group_values in grouped:
    print(group_key)
    print(group_values)
```

```
data
      data1     data2
0 -0.204708  0.281746
1  0.478943  0.769023
2 -0.519439  1.246435
3 -0.555730  1.007189
4  1.965781 -1.296221
5  1.393406  0.274992
6  0.092908  0.228913
key
   key1  key2
0     a     1
1     a     2
2  None     1
3     b     2
4     b     1
5     a  <NA>
6  None     1
```

Seleccionar una columna o un subconjunto de columnas

Indexar un objeto GroupBy creado a partir de un DataFrame con un nombre de columna o un array de nombres de columna tiene el efecto de subconjunto de columnas para la agregación. Esto significa que:

In [71]:
```
df.groupby("key1")["data1"]
```

Out[71]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001D5603BFEB0>
```

In [72]:
```
df.groupby("key1")[["data2"]]
```

Out[72]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001D5604C11F0>
```

Especialmente para grandes conjuntos de datos (large datasets), puede ser deseable agregar sólo unas pocas columnas. Por ejemplo, en el conjunto de datos anterior, para calcular las medias de sólo la columna data2 y obtener el resultado como DataFrame, podríamos escribir:

In [73]:
```
df.groupby(["key1", "key2"])[["data2"]].mean()
```

Out[73]:

		data2
key1	**key2**	
a	**1**	0.281746
	2	0.769023
b	**1**	-1.296221
	2	1.007189

El objeto devuelto por esta operación de indexación es un DataFrame agrupado si se pasa una lista o array, o una Serie agrupada si sólo se pasa un nombre de columna como escalar:

In [74]:
```
s_grouped = df.groupby(["key1", "key2"])["data2"]
```

In [75]:
```
s_grouped
```

Out[75]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001D5603E2820>
```

In [76]:
```
s_grouped.mean()
```

Out[76]:
```
key1  key2
a     1        0.281746
      2        0.769023
b     1       -1.296221
      2        1.007189
Name: data2, dtype: float64
```

Agrupación con diccionarios y series

La información de agrupación puede existir de otra forma que no sea un array. Consideremos otro ejemplo:

In [77]:
```
people = pd.DataFrame(np.random.standard_normal((5, 5)),
                      columns=["a", "b", "c", "d", "e"],
                      index=["Joe", "Steve", "Wanda", "Jill", "Trey"])
people
```

Out[77]:

	a	b	c	d	e
Joe	1.352917	0.886429	-2.001637	-0.371843	1.669025
Steve	-0.438570	-0.539741	0.476985	3.248944	-1.021228
Wanda	-0.577087	0.124121	0.302614	0.523772	0.000940

Jill	1.343810	-0.713544	-0.831154	-2.370232	-1.860761
Trey	-0.860757	0.560145	-1.265934	0.119827	-1.063512

In [78]:
```
people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values
```

In [79]:
```
people
```

Out[79]:

	a	b	c	d	e
Joe	1.352917	0.886429	-2.001637	-0.371843	1.669025
Steve	-0.438570	-0.539741	0.476985	3.248944	-1.021228
Wanda	-0.577087	NaN	NaN	0.523772	0.000940
Jill	1.343810	-0.713544	-0.831154	-2.370232	-1.860761
Trey	-0.860757	0.560145	-1.265934	0.119827	-1.063512

Ahora, supongamos que tengo una correspondencia de grupo para las columnas y quiero sumar las columnas por grupo:

In [80]:
```
mapping = {"a": "red", "b": "red", "c": "blue",
           "d": "blue", "e": "red", "f" : "orange"}
```

Ahora bien, se podría construir un array a partir de este diccionario para pasarlo a groupby, pero en su lugar podemos simplemente pasar el diccionario (se ha incluido la clave "f" para resaltar que las claves de agrupación no utilizadas están bien):

In [81]:
```
by_column = people.groupby(mapping, axis="columns")
by_column
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\1506222816.py:1: FutureWarning: DataFr
ame.groupby with axis=1 is deprecated. Do `frame.T.groupby(...)` without axis instead.
  by_column = people.groupby(mapping, axis="columns")
```

Out[81]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001D560455AF0>
```

In [82]:
```
by_column.sum()
```

Out[82]:

	blue	red
Joe	-2.373480	3.908371
Steve	3.725929	-1.999539
Wanda	0.523772	-0.576147
Jill	-3.201385	-1.230495
Trey	-1.146107	-1.364125

La misma funcionalidad se aplica a las series, que pueden verse como una asignación de tamaño fijo (fixed size mapping):

In [83]:
```
map_series = pd.Series(mapping)
```

In [84]:
```
map_series
```

```
a       red
```

Out[84]:
```
b       red
c      blue
d      blue
e       red
f    orange
dtype: object
```

In [85]:
```
people.groupby(map_series, axis="columns").count()
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\2727320543.py:1: FutureWarning: DataFr
ame.groupby with axis=1 is deprecated. Do `frame.T.groupby(...)` without axis instead.
  people.groupby(map_series, axis="columns").count()
```

Out[85]:

	blue	red
Joe	2	3
Steve	2	3
Wanda	1	2
Jill	2	3
Trey	2	3

Agrupación con funciones

El uso de funciones Python es una forma más genérica de definir una asignación de grupo (group mapping) en comparación con un diccionario o una serie. Cualquier función que se pase como clave de grupo se llamará una vez por cada valor de índice (o una vez por cada valor de columna si se utiliza axis="columns"), y los valores devueltos se utilizarán como nombres de grupo. Más concretamente, considere el ejemplo DataFrame de la sección anterior, que tiene los nombres de pila de las personas como valores de índice. Supongamos que desea agrupar por la longitud del nombre. Aunque podría calcular un array de longitudes de cadena, es más sencillo pasar la función `len`:

In [86]:
```
people.groupby(len).sum()
```

Out[86]:

	a	b	c	d	e
3	1.352917	0.886429	-2.001637	-0.371843	1.669025
4	0.483052	-0.153399	-2.097088	-2.250405	-2.924273
5	-1.015657	-0.539741	0.476985	3.772716	-1.020287

Mezclar funciones con arrays, diccionarios o Series no es un problema, ya que todo se convierte en arrays internamente:

In [87]:
```
key_list = ["one", "one", "one", "two", "two"]
```

In [88]:
```
people.groupby([len, key_list]).min()
```

Out[88]:

		a	b	c	d	e
3	**one**	1.352917	0.886429	-2.001637	-0.371843	1.669025
4	**two**	-0.860757	-0.713544	-1.265934	-2.370232	-1.860761
5	**one**	-0.577087	-0.539741	0.476985	0.523772	-1.021228

Agrupación por niveles de índice

Una última posibilidad para los conjuntos de datos indexados jerárquicamente es la posibilidad de agregar utilizando uno de los niveles de un índice de eje. Veamos un ejemplo:

In [89]:
```
columns = pd.MultiIndex.from_arrays([["US", "US", "US", "JP", "JP"],
                                     [1, 3, 5, 1, 3]],
                                     names=["cty", "tenor"])
columns
```

Out[89]:
```
MultiIndex([('US', 1),
            ('US', 3),
            ('US', 5),
            ('JP', 1),
            ('JP', 3)],
           names=['cty', 'tenor'])
```

In [90]:
```
hier_df = pd.DataFrame(np.random.standard_normal((4, 5)), columns=columns)
```

In [91]:
```
hier_df
```

Out[91]:

cty			US		JP
tenor	**1**	**3**	**5**	**1**	**3**
0	0.332883	-2.359419	-0.199543	-1.541996	-0.970736
1	-1.307030	0.286350	0.377984	-0.753887	0.331286
2	1.349742	0.069877	0.246674	-0.011862	1.004812
3	1.327195	-0.919262	-1.549106	0.022185	0.758363

Para agrupar por nivel, pase el número o el nombre del nivel utilizando la palabra clave `level`:

In [92]:
```
hier_df.groupby(level="cty", axis="columns").count()
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\2795167867.py:1: FutureWarning: DataFr
ame.groupby with axis=1 is deprecated. Do `frame.T.groupby(...)` without axis instead.
  hier_df.groupby(level="cty", axis="columns").count()
```

Out[92]:

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

4.2 Agregación de datos

Las agregaciones se refieren a cualquier transformación de datos que produzca valores escalares a partir de arrays. En los ejemplos anteriores se han utilizado varias de ellas, como la media, el recuento (count), el mínimo y la suma. Puede que se pregunte qué ocurre cuando invoca mean() en un objeto GroupBy. Muchas agregaciones comunes, como las que se encuentran en la siguiente Tabla , tienen implementaciones optimizadas. Sin embargo, no está limitado sólo a este conjunto de métodos.

`any, all` : Devuelve True si alguno (uno o más valores) o todos los valores no-NA son "truthy"

`count` : Número de valores no NA

`cummin, cummax` : Mínimo y máximo acumulados de los valores no NA

`cumsum` : Suma acumulada de los valores no NA

`cumprod` : Producto acumulado de los valores no NA

`first, last` : First and last non-NA values

`mean` : Media de los valores no NA

`median` : Mediana aritmética de los valores no NA

`min, max` : Mínimo y máximo de los valores no NA

`nth` : Recuperar el valor que aparecería en la posición n con los datos ordenados

`ohlc` : Calcular cuatro estadísticas de "apertura-alta-baja-cierre" para datos de tipo serie temporal

`prod` : Producto de valores no NA

`cuantil` : Calcular el cuantil de la muestra

`rango` : Rangos ordinales de valores no NA, como llamar a `Series.rank`

`size` : Calcular el tamaño de los grupos, devolviendo el resultado como una Serie.

`sum` : Suma de los valores no NA

`std, var` : Desviación típica y varianza de la muestra

Puede utilizar agregaciones de su propia autoría y, además, llamar a cualquier método que también esté definido en el objeto que se está agrupando. Por ejemplo, el método `nsmallest` Series selecciona el menor número solicitado de valores de los datos. Aunque `nsmallest` no está implementado explícitamente para GroupBy, podemos utilizarlo con una implementación no optimizada. Internamente, GroupBy trocea(slices) la serie, llama a `piece.nsmallest(n)` para cada trozo y, a continuación, reúne los resultados en el objeto resultante:

In [93]:
```
df
```

Out[93]:

	key1	key2	data1	data2
0	a	1	-0.204708	0.281746
1	a	2	0.478943	0.769023
2	None	1	-0.519439	1.246435
3	b	2	-0.555730	1.007189
4	b	1	1.965781	-1.296221
5	a	<NA>	1.393406	0.274992
6	None	1	0.092908	0.228913

In [94]:
```
grouped = df.groupby("key1")
grouped
```

Out[94]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001D560505D00>
```

In [95]:
```
grouped["data1"].nsmallest(2)
```

Out[95]:
```
key1
a     0   -0.204708
      1    0.478943
b     3   -0.555730
      4    1.965781
Name: data1, dtype: float64
```

Para utilizar sus propias funciones de agregación, pase cualquier función que agregue un array al método `aggregate` o a su alias corto `agg`:

In [96]:
```
def peak_to_peak(arr):
    return arr.max() - arr.min()
```

In [97]:
```
grouped.agg(peak_to_peak)
```

Out[97]:

key1	key2	data1	data2
a	1	1.598113	0.494031
b	1	2.521511	2.303410

Puede observar que algunos métodos, como `describe`, también funcionan, aunque no sean agregaciones, estrictamente hablando:

In [98]:
```
grouped.describe()
```

Out[98]:

	key2								data1					
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	r
key1															
a	2.0	1.5	0.707107	1.0	1.25	1.5	1.75	2.0	3.0	0.555881	...	0.936175	1.393406	3.0	0.44
b	2.0	1.5	0.707107	1.0	1.25	1.5	1.75	2.0	2.0	0.705025	...	1.335403	1.965781	2.0	-0.14

2 rows × 24 columns

Aplicación por columnas y funciones múltiples

Vamos a utilizar el conjunto de datos tips.csv (propinas). Tras cargarlo con pandas.read_csv, añadimos una columna de porcentaje de propina:

In [99]:
```
tips = pd.read_csv("tips.csv")
```

In [100…
```
tips.head()
```

Out[100]:

	total_bill	tip	smoker	day	time	size
0	16.99	1.01	No	Sun	Dinner	2
1	10.34	1.66	No	Sun	Dinner	3
2	21.01	3.50	No	Sun	Dinner	3

3	23.68	3.31	No	Sun	Dinner	2
4	24.59	3.61	No	Sun	Dinner	4

Ahora se añadirá una columna `tip_pct` con el porcentaje de propina de la factura total:

In [101…
```
tips["tip_pct"] = tips["tip"] / tips["total_bill"]
```

In [102…
```
tips.head()
```

Out[102]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Como ya has visto, agregar una Serie o todas las columnas de un DataFrame es cuestión de usar `aggregate` (o agg) con la función deseada o llamar a un método como `mean` o `std`. Sin embargo, puede que quieras agregar usando una función diferente, dependiendo de la columna, o múltiples funciones a la vez. Afortunadamente, esto es posible de hacer, en primer lugar, se agruparán las propinas por día y fumador:

In [103…
```
grouped = tips.groupby(["day", "smoker"])
```

Tenga en cuenta que para estadísticas descriptivas como las de la anterior, puede pasar el nombre de la función como una cadena:

In [104…
```
grouped_pct = grouped["tip_pct"]
```

In [105…
```
grouped_pct.agg("mean")
```

Out[105]:
```
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

Si en su lugar pasa una lista de funciones o nombres de funciones, obtendrá un DataFrame con nombres de columnas tomados de las funciones:

In [106…
```
grouped_pct.agg(["mean", "std", peak_to_peak])
```

Out[106]:

		mean	std	peak_to_peak
day	**smoker**			
Fri	**No**	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925

Sat	**No**	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	**No**	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	**No**	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

Aquí pasamos una lista de funciones de agregación a agg para que las evalúe independientemente en los grupos de datos.

No es necesario que aceptes los nombres que GroupBy da a las columnas; en particular, las funciones `lambda` tienen el nombre "", lo que las hace difíciles de identificar . Por lo tanto, si pasa una lista de tuplas (nombre, función), el primer elemento de cada tupla se utilizará como los nombres de columna del DataFrame (puede pensar en una lista de 2 tuplas como un mapeo (mapping) ordenado):

In [107…
```
grouped_pct.agg([("average", "mean"), ("stdev", np.std)])
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\1734782830.py:1: FutureWarning: The pr
ovided callable <function std at 0x000001D55E4D9280> is currently using SeriesGroupBy.st
d. In a future version of pandas, the provided callable will be used directly. To keep c
urrent behavior pass 'std' instead.
  grouped_pct.agg([("average", "mean"), ("stdev", np.std)])
```

Out[107]:

		average	stdev
day	**smoker**		
Fri	**No**	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	**No**	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	**No**	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	**No**	0.160298	0.038774
	Yes	0.163863	0.039389

Con un DataFrame se tienen más opciones, ya que puede especificar una lista de funciones para aplicar a todas las columnas o diferentes funciones por columna. Para empezar, supongamos que queremos calcular las mismas tres estadísticas para las columnas `tip_pct` y `total_bill` :

In [108…
```
functions = ["count", "mean", "max"]
```

In [109…
```
result = grouped[["tip_pct", "total_bill"]].agg(functions)
```

In [110…
```
result
```

Out[110]:

		tip_pct			total_bill		
		count	**mean**	**max**	**count**	**mean**	**max**
day	**smoker**						

Fri	**No**	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	**No**	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	**No**	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	**No**	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

Como puede ver, el DataFrame resultante tiene columnas jerárquicas, lo mismo que obtendría agregando cada columna por separado y utilizando concat para unir los resultados utilizando los nombres de las columnas como argumento clave:

In [111…
```
result["tip_pct"]
```

Out[111]:

		count	mean	max
day	**smoker**			
Fri	**No**	4	0.151650	0.187735
	Yes	15	0.174783	0.263480
Sat	**No**	45	0.158048	0.291990
	Yes	42	0.147906	0.325733
Sun	**No**	57	0.160113	0.252672
	Yes	19	0.187250	0.710345
Thur	**No**	45	0.160298	0.266312
	Yes	17	0.163863	0.241255

Como antes, se puede pasar una lista de tuplas con nombres personalizados:

In [112…
```
ftuples = [("Average", "mean"), ("Variance", np.var)]
```

In [113…
```
grouped[["tip_pct", "total_bill"]].agg(ftuples)
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\365474927.py:1: FutureWarning: The pro
vided callable <function var at 0x000001D55E4D93A0> is currently using SeriesGroupBy.va
r. In a future version of pandas, the provided callable will be used directly. To keep c
urrent behavior pass 'var' instead.
  grouped[["tip_pct", "total_bill"]].agg(ftuples)
```

Out[113]:

		tip_pct		total_bill	
		Average	Variance	Average	Variance
day	**smoker**				
Fri	**No**	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	**No**	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535

Sun	**No**	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	**No**	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

Ahora, supongamos que desea aplicar funciones potencialmente diferentes a una o más de las columnas. Para ello, pase un diccionario a agg que contenga un mapeo de nombres de columnas a cualquiera de las especificaciones de función enumeradas hasta ahora:

In [114…
```
grouped.agg({"tip" : np.max, "size" : "sum"})
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\2044707828.py:1: FutureWarning: The pr
ovided callable <function max at 0x000001D55E4D5820> is currently using SeriesGroupBy.ma
x. In a future version of pandas, the provided callable will be used directly. To keep c
urrent behavior pass 'max' instead.
  grouped.agg({"tip" : np.max, "size" : "sum"})
```

Out[114]:

		tip	size
day	**smoker**		
Fri	**No**	3.50	9
	Yes	4.73	31
Sat	**No**	9.00	115
	Yes	10.00	104
Sun	**No**	6.00	167
	Yes	6.50	49
Thur	**No**	6.70	112
	Yes	5.00	40

In [115…
```
grouped.agg({"tip_pct" : ["min", "max", "mean", "std"],
             "size" : "sum"})
```

Out[115]:

		tip_pct				size
		min	max	mean	std	sum
day	**smoker**					
Fri	**No**	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	**No**	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	**No**	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	**No**	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

Un DataFrame tendrá columnas jerárquicas sólo si se aplican múltiples funciones al menos a una columna.

Devolución de datos agregados sin índices de filas

En todos los ejemplos hasta ahora, los datos agregados vuelven con un índice, potencialmente jerárquico, compuesto a partir de las combinaciones únicas de claves de grupo. Dado que esto no siempre es deseable, puede desactivar este comportamiento en la mayoría de los casos pasando as_index=False a groupby:

In [116…
```
grouped = tips.groupby(["day", "smoker"], as_index=False)
```

In [117…
```
grouped.mean(numeric_only=True)
```

Out[117]:

	day	smoker	total_bill	tip	size	tip_pct
0	Fri	No	18.420000	2.812500	2.250000	0.151650
1	Fri	Yes	16.813333	2.714000	2.066667	0.174783
2	Sat	No	19.661778	3.102889	2.555556	0.158048
3	Sat	Yes	21.276667	2.875476	2.476190	0.147906
4	Sun	No	20.506667	3.167895	2.929825	0.160113
5	Sun	Yes	24.120000	3.516842	2.578947	0.187250
6	Thur	No	17.113111	2.673778	2.488889	0.160298
7	Thur	Yes	19.190588	3.030000	2.352941	0.163863

Por supuesto, siempre es posible obtener el resultado en este formato llamando a reset_index sobre el resultado. El uso del argumento as_index=False evita algunos cálculos innecesarios.

4.3 Aplicar: General dividir-aplicar-combinar (split-apply-combine)

El método GroupBy de propósito más general es `apply`. Este metodo `apply` divide el objeto que se está manipulando en trozos, invoca la función pasada en cada trozo, y luego intenta concatenar los trozos.

Volviendo al conjunto de datos de propinas de antes, supongamos que desea seleccionar los cinco valores principales de `tip_pct` por grupo. Primero, se escribe una función que seleccione las filas con los mayores valores en una columna en particular:

In [128…
```
def top(df, n=5, column="tip_pct"):
    return df.sort_values(column, ascending=False)[:n]
```

In [129…
```
top(tips, n=6)
```

Out[129]:

	total_bill	tip	smoker	day	time	size	tip_pct
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
232	11.61	3.39	No	Sat	Dinner	2	0.291990
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Ahora, si agrupamos por (smoker) fumador, digamos, y llamamos a `apply` con esta función, obtenemos lo siguiente:

In [120…
```
tips.groupby("smoker").apply(top)
```

Out[120]:

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	**232**	11.61	3.39	No	Sat	Dinner	2	0.291990
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	88	24.71	5.85	No	Thur	Lunch	2	0.236746
Yes	**172**	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

¿Qué ha ocurrido aquí? En primer lugar, la DataFrame `tips` se divide en grupos basados en el valor de `smoker`. Después se llama a la función `top` en cada grupo, y los resultados de cada llamada a la función se pegan usando `pandas.concat`, etiquetando(labelling) las piezas con los nombres de los grupos. Por lo tanto, el resultado tiene un índice jerárquico con un nivel interno que contiene valores de índice del DataFrame original.

Si se pasa una función a `apply` que toma otros argumentos o palabras clave, puede pasarlos después de la función:

In [121…
```
tips.groupby(["smoker", "day"]).apply(top, n=1, column="total_bill")
```

Out[121]:

			total_bill	tip	smoker	day	time	size	tip_pct
smoker	**day**								
No	**Fri**	**94**	22.75	3.25	No	Fri	Dinner	2	0.142857
	Sat	**212**	48.33	9.00	No	Sat	Dinner	4	0.186220
	Sun	**156**	48.17	5.00	No	Sun	Dinner	6	0.103799
	Thur	**142**	41.19	5.00	No	Thur	Lunch	5	0.121389
Yes	**Fri**	**95**	40.17	4.73	Yes	Fri	Dinner	4	0.117750
	Sat	**170**	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	Sun	**182**	45.35	3.50	Yes	Sun	Dinner	3	0.077178
	Thur	**197**	43.11	5.00	Yes	Thur	Lunch	4	0.115982

Más allá de esta mecánica básica de uso, sacar el máximo partido de apply puede requerir algo de creatividad. Lo que ocurra dentro de la función pasada depende del analista; debe devolver un objeto

pandas o un valor escalar. El resto de este capítulo consistirá principalmente en ejemplos que muestran cómo resolver varios problemas utilizando groupby.

Por ejemplo, recordamos que antes llamó a `describe` sobre un objeto GroupBy:

In [122…
```
result = tips.groupby("smoker")["tip_pct"].describe()
```

In [123…
```
result
```

Out[123]:

smoker	count	mean	std	min	25%	50%	75%	max
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014	0.291990
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059	0.710345

In [124…
```
result.unstack("smoker")
```

Out[124]:
```
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

Dentro de GroupBy, cuando se invoca un método como `describe`, en realidad es sólo un atajo para:

In [125…
```
def f(group):
    return group.describe()

grouped.apply(f)
```

Out[125]:

		total_bill	tip	size	tip_pct
0	**count**	4.000000	4.000000	4.00	4.000000
	mean	18.420000	2.812500	2.25	0.151650
	std	5.059282	0.898494	0.50	0.028123
	min	12.460000	1.500000	2.00	0.120385
	25%	15.100000	2.625000	2.00	0.137239
…	**…**	…	…	…	…
7	**min**	10.340000	2.000000	2.00	0.090014
	25%	13.510000	2.000000	2.00	0.148038
	50%	16.470000	2.560000	2.00	0.153846

75%	19.810000	4.000000	2.00	0.194837
max	43.110000	5.000000	4.00	0.241255

64 rows × 4 columns

Suprimir las claves de grupo

En los ejemplos anteriores, se ve que el objeto resultante tiene un índice jerárquico formado a partir de las claves de grupo, junto con los índices de cada pieza del objeto original. Puede desactivar esto pasando group_keys=False a groupby:

In [127…
```
tips.groupby("smoker", group_keys=False).apply(top)
```

Out[127]:

	total_bill	tip	smoker	day	time	size	tip_pct
232	11.61	3.39	No	Sat	Dinner	2	0.291990
149	7.51	2.00	No	Thur	Lunch	2	0.266312
51	10.29	2.60	No	Sun	Dinner	2	0.252672
185	20.69	5.00	No	Sun	Dinner	5	0.241663
88	24.71	5.85	No	Thur	Lunch	2	0.236746
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

In [130…
```
tips.groupby("smoker").apply(top)
```

Out[130]:

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	**232**	11.61	3.39	No	Sat	Dinner	2	0.291990
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	88	24.71	5.85	No	Thur	Lunch	2	0.236746
Yes	**172**	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

Análisis de cuantiles y Bucket ("Contenedor")

Como se vió en el tema de Data Wrangling (Join, Combine, and Reshape), pandas tiene algunas herramientas, en particular `pandas.cut` y `pandas.qcut`, para rebanar los datos en buckets con bins de su elección, o por cuantiles de muestra. Combinando estas funciones con groupby es conveniente realizar análisis de ' buckets' o cuantiles en un conjunto de datos. Considere un conjunto de datos aleatorio simple y una categorización de buckets de igual longitud utilizando `pandas.cut`:

In [131…
```
frame = pd.DataFrame({"data1": np.random.standard_normal(1000),
                      "data2": np.random.standard_normal(1000)})


frame
```

Out[131]:

	data1	data2
0	-0.660524	-0.612905
1	0.862580	0.316447
2	-0.010032	0.838295
3	0.050009	-1.034423
4	0.670216	0.434304
...	...	...
995	-1.261344	1.170900
996	1.165148	0.678661
997	-0.621249	-0.125921
998	-0.799318	0.150581
999	0.777233	-0.884475

1000 rows × 2 columns

In [132…
```
frame.head()
```

Out[132]:

	data1	data2
0	-0.660524	-0.612905
1	0.862580	0.316447
2	-0.010032	0.838295
3	0.050009	-1.034423
4	0.670216	0.434304

In [133…
```
quartiles = pd.cut(frame["data1"], 4)
```

In [136…
```
quartiles.head(10)
```

Out[136]:
```
0      (-1.23, 0.489]
1      (0.489, 2.208]
2      (-1.23, 0.489]
3      (-1.23, 0.489]
4      (0.489, 2.208]
5      (0.489, 2.208]
6      (-1.23, 0.489]
7      (-1.23, 0.489]
```

```
8     (-2.956, -1.23]
9      (-1.23, 0.489]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.
208] < (2.208, 3.928]]
```

El objeto `Categorical` devuelto por `cut` puede pasarse directamente a groupby. Así que podríamos calcular un conjunto de estadísticas de grupo para los cuartiles, así:

In [137…
```
def get_stats(group):
    return pd.DataFrame(
        {"min": group.min(), "max": group.max(),
        "count": group.count(), "mean": group.mean()}
    )
```

In [138…
```
grouped = frame.groupby(quartiles)
```

```
C:\Users\juanj\AppData\Local\Temp\ipykernel_16200\4041437760.py:1: FutureWarning: The de
fault of observed=False is deprecated and will be changed to True in a future version of
pandas. Pass observed=False to retain current behavior or observed=True to adopt the fut
ure default and silence this warning.
  grouped = frame.groupby(quartiles)
```

In [139…
```
grouped.apply(get_stats)
```

Out[139]:

		min	max	count	mean
data1					
(-2.956, -1.23]	**data1**	-2.949343	-1.230179	94	-1.658818
	data2	-3.399312	1.670835	94	-0.033333
(-1.23, 0.489]	**data1**	-1.228918	0.488675	598	-0.329524
	data2	-2.989741	3.260383	598	-0.002622
(0.489, 2.208]	**data1**	0.489965	2.200997	298	1.065727
	data2	-3.745356	2.954439	298	0.078249
(2.208, 3.928]	**data1**	2.212303	3.927528	10	2.644253
	data2	-1.929776	1.765640	10	0.024750

Ten en cuenta que el mismo resultado podría haberse calculado de forma más sencilla con:

In [140…
```
grouped.agg(["min", "max", "count", "mean"])
```

Out[140]:

	data1				data2			
	min	max	count	mean	min	max	count	mean
data1								
(-2.956, -1.23]	-2.949343	-1.230179	94	-1.658818	-3.399312	1.670835	94	-0.033333
(-1.23, 0.489]	-1.228918	0.488675	598	-0.329524	-2.989741	3.260383	598	-0.002622
(0.489, 2.208]	0.489965	2.200997	298	1.065727	-3.745356	2.954439	298	0.078249
(2.208, 3.928]	2.212303	3.927528	10	2.644253	-1.929776	1.765640	10	0.024750

Estos eran cubos de igual longitud; para calcular cubos de igual tamaño basados en cuantiles muestrales, utilice pandas.qcut. Podemos pasar 4 como el número de cubos para calcular los cuartiles de la muestra, y

pasar labels=False para obtener sólo los índices de los cuartiles en lugar de los intervalos:

In [87]:
```
quartiles_samp = pd.qcut(frame["data1"], 4, labels=False)
quartiles_samp
```

Out[87]:
```
0      1
1      3
2      2
3      2
4      3
      ..
995    0
996    3
997    1
998    0
999    3
Name: data1, Length: 1000, dtype: int64
```

In [88]:
```
quartiles_samp.head()
```

Out[88]:
```
0    1
1    3
2    2
3    2
4    3
Name: data1, dtype: int64
```

In [89]:
```
grouped = frame.groupby(quartiles_samp)
```

In [90]:
```
grouped.apply(get_stats)
```

Out[90]:

		min	max	count	mean
data1					
0	**data1**	-2.949343	-0.685484	250	-1.212173
	data2	-3.399312	2.628441	250	-0.027045
1	**data1**	-0.683066	-0.030280	250	-0.368334
	data2	-2.630247	3.260383	250	-0.027845
2	**data1**	-0.027734	0.618965	250	0.295812
	data2	-3.056990	2.458842	250	0.014450
3	**data1**	0.623587	3.927528	250	1.248875
	data2	-3.745356	2.954439	250	0.115899

Ejemplo: Rellenar valores faltantes con valores específicos de grupo

Al limpiar los datos que faltan, en algunos casos eliminará las observaciones de datos utilizando dropna, pero en otros es posible que desee rellenar los valores nulos (NA) utilizando un valor fijo o algún valor derivado de los datos. fillna es la herramienta adecuada para utilizar; por ejemplo, aquí se rellenan los valores nulos con la media:

In [91]:
```
s = pd.Series(np.random.standard_normal(6))
```

In [92]:
```
s[::2] = np.nan
```

In [93]:
```
s
```

Out[93]:
```
0         NaN
1    0.227290
2         NaN
3   -2.153545
4         NaN
5   -0.375842
dtype: float64
```

In [94]:
```
s.fillna(s.mean())
```

Out[94]:
```
0   -0.767366
1    0.227290
2   -0.767366
3   -2.153545
4   -0.767366
5   -0.375842
dtype: float64
```

Supongamos que necesita que el valor de relleno varíe según el grupo. Una forma de hacerlo es agrupar los datos y utilizar apply con una función que llame a fillna en cada trozo de datos. Aquí hay algunos datos de muestra sobre los estados de EE.UU. divididos en regiones orientales y occidentales:

In [95]:
```
states = ["Ohio", "New York", "Vermont", "Florida",
          "Oregon", "Nevada", "California", "Idaho"]
group_key = ["East", "East", "East", "East",
             "West", "West", "West", "West"]
```

In [96]:
```
data = pd.Series(np.random.standard_normal(8), index=states)
```

In [97]:
```
data
```

Out[97]:
```
Ohio          0.329939
New York      0.981994
Vermont       1.105913
Florida      -1.613716
Oregon        1.561587
Nevada        0.406510
California    0.359244
Idaho        -0.614436
dtype: float64
```

Establezcamos que faltan algunos valores en los datos:

In [99]:
```
data[["Vermont", "Nevada", "Idaho"]] = np.nan
data
```

Out[99]:
```
Ohio          0.329939
New York      0.981994
Vermont            NaN
Florida      -1.613716
Oregon        1.561587
Nevada             NaN
California    0.359244
Idaho              NaN
dtype: float64
```

In [100…
```
data.groupby(group_key).size()
```

Out[100]:
```
East    4
West    4
dtype: int64
```

In [101…
```
data.groupby(group_key).count()
```

Out[101]:
```
East    3
West    2
dtype: int64
```

In [103…
```
data.groupby(group_key).mean()
```

Out[103]:
```
East   -0.100594
West    0.960416
dtype: float64
```

Podemos rellenar los valores NA utilizando las medias de grupo, así:

In [102…
```
def fill_mean(group):
    return group.fillna(group.mean())

data.groupby(group_key).apply(fill_mean)
```

Out[102]:
```
East  Ohio          0.329939
      New York      0.981994
      Vermont      -0.100594
      Florida      -1.613716
West  Oregon        1.561587
      Nevada        0.960416
      California    0.359244
      Idaho         0.960416
dtype: float64
```

En otro caso, es posible que tenga valores de relleno predefinidos en su código que varían según el grupo. Dado que los grupos tienen un atributo de nombre establecido internamente, podemos utilizarlo:

In [105…
```
fill_values = {"East": 0.5, "West": -1}
def fill_func(group):
    return group.fillna(fill_values[group.name])

data.groupby(group_key).apply(fill_func)
```

Out[105]:
```
East  Ohio          0.329939
      New York      0.981994
      Vermont       0.500000
      Florida      -1.613716
West  Oregon        1.561587
      Nevada       -1.000000
      California    0.359244
      Idaho        -1.000000
dtype: float64
```

Ejemplo: Muestreo aleatorio y permutación

Supongamos que desea extraer una muestra aleatoria (con o sin reemplazo) de un gran conjunto de datos con fines de simulación Monte Carlo o alguna otra aplicación. Hay varias formas de realizar los "sorteos"; aquí utilizamos el método de muestreo para Series.

Para demostrarlo, aquí tienes una forma de construir una baraja de naipes al estilo inglés:

In [106…
```
suits = ["H", "S", "C", "D"]  # Hearts, Spades, Clubs, Diamonds
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ["A"] + list(range(2, 11)) + ["J", "K", "Q"]
cards = []
for suit in suits:
    cards.extend(str(num) + suit for num in base_names)
```

In [109…
```
deck = pd.Series(card_val, index=cards)
```

Ahora tenemos una Serie de longitud 52 cuyo índice contiene nombres de cartas, y los valores son los que se usan en el blackjack y otros juegos (para simplificar las cosas, dejo que el as "A" sea 1):

In [108…
```
deck.head(13)
```

Out[108]:
```
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

Ahora, basándome en lo que he dicho antes, robar una mano de cinco cartas de la baraja podría escribirse como:

In [110…
```
def draw(deck, n=5):
    return deck.sample(n)
draw(deck)
```

Out[110]:
```
4D     4
QH    10
8S     8
7D     7
9C     9
dtype: int64
```

Supongamos que queremos dos cartas al azar de cada palo. Dado que el palo es el último carácter del nombre de cada carta, podemos agruparlas en función de éste y utilizar aplicar:

In [111…
```
def get_suit(card):
    # last letter is suit
    return card[-1]

deck.groupby(get_suit).apply(draw, n=2)
```

Out[111]:
```
C  6C     6
   KC    10
D  7D     7
   3D     3
H  7H     7
   9H     9
S  2S     2
   QS    10
dtype: int64
```

Alternativamente, podríamos pasar group_keys=False para eliminar el índice de palos externo, dejando sólo las cartas seleccionadas:

In [112…
```
deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
```

Out[112]:
```
AC      1
3C      3
5D      5
```

```
4D      4
10H    10
7H      7
QS     10
7S      7
dtype: int64
```

Ejemplo: Media ponderada por grupo y correlación

Bajo el paradigma dividir-aplicar-combinar (splitapply-combine) de groupby, son posibles las operaciones entre columnas de un DataFrame o dos Series, como una media ponderada de grupo. Como ejemplo, tomemos este conjunto de datos que contiene claves de grupo, valores y algunos pesos:

In [113…
```
df = pd.DataFrame({"category": ["a", "a", "a", "a",
                                "b", "b", "b", "b"],
                   "data": np.random.standard_normal(8),
                   "weights": np.random.uniform(size=8)})
df
```

Out[113]:

	category	data	weights
0	a	-1.691656	0.955905
1	a	0.511622	0.012745
2	a	-0.401675	0.137009
3	a	0.968578	0.763037
4	b	-1.818215	0.492472
5	b	0.279963	0.832908
6	b	-0.200819	0.658331
7	b	-0.217221	0.612009

La media ponderada por categoría sería entonces:

In [114…
```
grouped = df.groupby("category")
def get_wavg(group):
    return np.average(group["data"], weights=group["weights"])

grouped.apply(get_wavg)
```

Out[114]:
```
category
a   -0.495807
b   -0.357273
dtype: float64
```

Como ejemplo, consideremos un conjunto de datos financieros obtenidos originalmente de Yahoo! Finance que contiene los precios al final del día de algunas acciones y del índice S&P 500 (el símbolo SPX):

In [115…
```
close_px = pd.read_csv("stock_px.csv", parse_dates=True,
                       index_col=0)
close_px.info()
close_px.tail(4)
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
```

```
 0   AAPL    2214 non-null   float64
 1   MSFT    2214 non-null   float64
 2   XOM     2214 non-null   float64
 3   SPX     2214 non-null   float64
dtypes: float64(4)
memory usage: 86.5 KB
```

Out[115]:

	AAPL	MSFT	XOM	SPX
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

El método DataFrame info() es una forma práctica de obtener una visión general del contenido de un DataFrame.

Una tarea de interés podría ser calcular un DataFrame consistente en las correlaciones anuales de los rendimientos diarios (calculados a partir de los cambios porcentuales) con el SPX. Para ello, primero creamos una función que calcula la correlación por pares de cada columna con la columna "SPX":

In [116…

```
def spx_corr(group):
    return group.corrwith(group["SPX"])
```

A continuación, calculamos el cambio porcentual en close_px utilizando pct_change:

In [117…

```
rets = close_px.pct_change().dropna()
```

Por último, agrupamos estos cambios porcentuales por año, que puede extraerse de cada etiqueta de fila con una función de una línea que devuelve el atributo year de cada etiqueta datetime:

In [118…

```
def get_year(x):
    return x.year

by_year = rets.groupby(get_year)
by_year.apply(spx_corr)
```

Out[118]:

	AAPL	MSFT	XOM	SPX
2003	0.541124	0.745174	0.661265	1.0
2004	0.374283	0.588531	0.557742	1.0
2005	0.467540	0.562374	0.631010	1.0
2006	0.428267	0.406126	0.518514	1.0
2007	0.508118	0.658770	0.786264	1.0
2008	0.681434	0.804626	0.828303	1.0
2009	0.707103	0.654902	0.797921	1.0
2010	0.710105	0.730118	0.839057	1.0
2011	0.691931	0.800996	0.859975	1.0

También se pueden calcular correlaciones entre columnas. Aquí calculamos la correlación anual entre Apple y Microsoft:

In [119…
```
def corr_aapl_msft(group):
    return group["AAPL"].corr(group["MSFT"])
by_year.apply(corr_aapl_msft)
```

Out[119]:
```
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

Ejemplo: Regresión lineal por grupos

En la misma línea que el ejemplo anterior, puede utilizar groupby para realizar análisis estadísticos más complejos por grupos, siempre que la función devuelva un objeto pandas o un valor escalar. Por ejemplo, se puede definir la siguiente función `regress` (usando la librería econométrica statsmodels), que ejecuta una regresión por mínimos cuadrados ordinarios (MCO) en cada trozo de datos:

In [121…
```
import statsmodels.api as sm
def regress(data, yvar=None, xvars=None):
    Y = data[yvar]
    X = data[xvars]
    X["intercept"] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Puedes instalar `statsmodels` con conda si no lo tienes ya:

```
conda install statsmodels
```

Ahora, para ejecutar una regresión lineal anual de AAPL sobre los rendimientos del SPX, ejecute:

In [123…
```
by_year.apply(regress, yvar="AAPL", xvars=["SPX"])
```

Out[123]:

	SPX	intercept
2003	1.195406	0.000710
2004	1.363463	0.004201
2005	1.766415	0.003246
2006	1.645496	0.000080
2007	1.198761	0.003438
2008	0.968016	-0.001110
2009	0.879103	0.002954
2010	1.052608	0.001261
2011	0.806605	0.001514

4.4 Transformadas de grupo y GroupBys "desenrollados"

En la sección split-apply-combine, vimos el método apply en operaciones agrupadas para realizar transformaciones. Hay otro método incorporado llamado `transform`, que es similar a `apply`, pero impone más restricciones sobre el tipo de función que puede utilizar:

- Puede producir un valor escalar que se emitirá a la forma del grupo.
- Puede producir un objeto de la misma forma que el grupo de entrada.
- No debe mutar su entrada.
- Veamos un ejemplo sencillo para ilustrarlo:

In [124…
```
df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
                   'value': np.arange(12.)})
df
```

Out[124]:

	key	value
0	a	0.0
1	b	1.0
2	c	2.0
3	a	3.0
4	b	4.0
5	c	5.0
6	a	6.0
7	b	7.0
8	c	8.0
9	a	9.0
10	b	10.0
11	c	11.0

Aquí están los medios de grupo por clave:

In [125…
```
g = df.groupby('key')['value']
g.mean()
```

Out[125]:
```
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

Supongamos en cambio que queremos producir una Serie de la misma forma que df['valor'] pero con los valores sustituidos por la media agrupados por 'key'. Podemos pasar una función que calcule la media de un solo grupo a `transform`:

In [126…
```
def get_mean(group):
    return group.mean()
g.transform(get_mean)
```

Out[126]:
```
0     4.5
```

```
1      5.5
2      6.5
3      4.5
4      5.5
5      6.5
6      4.5
7      5.5
8      6.5
9      4.5
10     5.5
11     6.5
Name: value, dtype: float64
```

Para las funciones de agregación incorporadas, podemos pasar un alias de cadena como con el método GroupBy agg:

In [127…
```
g.transform('mean')
```

Out[127]:
```
0      4.5
1      5.5
2      6.5
3      4.5
4      5.5
5      6.5
6      4.5
7      5.5
8      6.5
9      4.5
10     5.5
11     6.5
Name: value, dtype: float64
```

Al igual que `apply`, transform funciona con funciones que devuelven series, pero el resultado debe tener el mismo tamaño que la entrada. Por ejemplo, podemos multiplicar cada grupo por 2 utilizando una función auxiliar:

In [128…
```
def times_two(group):
    return group * 2
g.transform(times_two)
```

Out[128]:
```
0       0.0
1       2.0
2       4.0
3       6.0
4       8.0
5      10.0
6      12.0
7      14.0
8      16.0
9      18.0
10     20.0
11     22.0
Name: value, dtype: float64
```

Como ejemplo más complicado, podemos calcular los rangos en orden descendente para cada grupo:

In [129…
```
def get_ranks(group):
    return group.rank(ascending=False)
g.transform(get_ranks)
```

Out[129]:
```
0      4.0
1      4.0
2      4.0
3      3.0
```

```
4     3.0
5     3.0
6     2.0
7     2.0
8     2.0
9     1.0
10    1.0
11    1.0
Name: value, dtype: float64
```

Consideremos una función de transformación de grupo compuesta a partir de agregaciones simples:

In [130…
```
def normalize(x):
    return (x - x.mean()) / x.std()
```

Podemos obtener resultados equivalentes en este caso utilizando la `transform` o `apply`

In [131…
```
g.transform(normalize)
```

Out[131]:
```
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

In [132…
```
g.apply(normalize)
```

Out[132]:
```
key
a    0    -1.161895
     3    -0.387298
     6     0.387298
     9     1.161895
b    1    -1.161895
     4    -0.387298
     7     0.387298
     10    1.161895
c    2    -1.161895
     5    -0.387298
     8     0.387298
     11    1.161895
Name: value, dtype: float64
```

Las funciones de agregación incorporadas, como "media" o "suma", suelen ser mucho más rápidas que una función de aplicación general. Éstas también tienen una "ruta rápida" cuando se utilizan con `transform`. Esto nos permite realizar lo que se denomina una operación de grupo sin envolver (unwrapped):

In [133…
```
g.transform('mean')
```

Out[133]:
```
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
```

```
8      6.5
9      4.5
10     5.5
11     6.5
Name: value, dtype: float64
```

In [134…
```
normalized = (df['value'] - g.transform('mean')) / g.transform('std')
```

In [135…
```
normalized
```

Out[135]:
```
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

Aquí, estamos haciendo aritmética entre las salidas de múltiples operaciones GroupBy en lugar de escribir una función y pasarla a groupby(...).apply. Esto es lo que se entiende por "unwrapped".

Aunque una operación de grupo "unwrapped" puede implicar múltiples agregaciones de grupo, el beneficio general de las operaciones vectorizadas a menudo supera esto.

4.5 Tablas dinámicas (Pivot Tables) y tabulaciones cruzadas (Cross-Tabulation)

Una tabla dinámica es una herramienta de resumen de datos que se encuentra con frecuencia en los programas de hojas de cálculo y otros programas de análisis de datos. Agrega una tabla de datos por una o más claves, organizando los datos en un rectángulo con algunas de las claves de grupo a lo largo de las filas y otras a lo largo de las columnas. Las tablas pivotantes en Python con pandas son posibles a través de la facilidad groupby descrita en este capítulo, combinada con operaciones de remodelación (reshape) utilizando indexación jerárquica. DataFrame también tiene un método `pivot_table`, y también hay una función de alto nivel `pandas.pivot_table`. Además de proporcionar una interfaz conveniente para groupby, pivot_table puede añadir totales parciales, también conocidos como márgenes (margins).

Volviendo al conjunto de datos de propinas (tips.csv), supongamos que desea calcular una tabla de medias de grupo (el tipo de agregación por defecto de pivot_table) ordenadas por `day` y `smoker` en las filas:

In [136…
```
tips.head()
```

Out[136]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

In [137…
```
tips.pivot_table(index=["day", "smoker"],
                 values=["size", "tip", "tip_pct", "total_bill"])
```

Out[137]:

day	smoker	size	tip	tip_pct	total_bill
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

Esto se podría haber producido con groupby directamente, utilizando `tips.groupby(["day", "smoker"]).mean()`. Ahora, supongamos que queremos tomar la media de sólo `tip_pct` y `size`, y adicionalmente agrupar por `time`. Aqui se pondrá `smoker` en las columnas de la tabla y `time` y `day` en las filas:

In [138…
```
tips.pivot_table(index=["time", "day"], columns="smoker",
                 values=["tip_pct", "size"])
```

Out[138]:

		size		tip_pct	
smoker		No	Yes	No	Yes
time	day				
Dinner	Fri	2.000000	2.222222	0.139622	0.165347
	Sat	2.555556	2.476190	0.158048	0.147906
	Sun	2.929825	2.578947	0.160113	0.187250
	Thur	2.000000	NaN	0.159744	NaN
Lunch	Fri	3.000000	1.833333	0.187735	0.188937
	Thur	2.500000	2.352941	0.160311	0.163863

Podríamos aumentar esta tabla para incluir totales parciales pasando `margins=True`. Esto tiene el efecto de añadir todas las etiquetas de fila y columna, con los valores correspondientes a las estadísticas de grupo para todos los datos dentro de un solo nivel:

In [139…
```
tips.pivot_table(index=["time", "day"], columns="smoker",
                 values=["tip_pct", "size"], margins=True)
```

Out[139]:

		size			tip_pct		
smoker		No	Yes	All	No	Yes	All
time	day						
Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
	Sat	2.555556	2.476190	2.517241	0.158048	0.147906	0.153152

	Sun	2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
	Thur	2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	**Fri**	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

Aquí, los valores `All` son medias sin tener en cuenta al smoker frente a non-smoker (las columnas `All`) ni ninguno de los dos niveles de agrupación en las filas (la fila `All`).

Para utilizar una función de agregación distinta de la media, pásela al argumento de la palabra clave `aggfunc`. Por ejemplo, "count" o len le proporcionarán una tabulación cruzada (cross-tabulation) (recuento o frecuencia) de los tamaños de los grupos (aunque "count" excluirá los valores nulos del recuento dentro de los grupos de datos, mientras que len no lo hará):

In [140…
```
tips.pivot_table(index=["time", "smoker"], columns="day",
                 values="tip_pct", aggfunc=len, margins=True)
```

Out[140]:

	day	**Fri**	**Sat**	**Sun**	**Thur**	**All**
time	**smoker**					
Dinner	**No**	3.0	45.0	57.0	1.0	106
	Yes	9.0	42.0	19.0	NaN	70
Lunch	**No**	1.0	NaN	NaN	44.0	45
	Yes	6.0	NaN	NaN	17.0	23
All		19.0	87.0	76.0	62.0	244

Si algunas combinaciones están vacías (o de otro modo NA), es posible que desee pasar un `fill_value`:

In [141…
```
tips.pivot_table(index=["time", "size", "smoker"], columns="day",
                 values="tip_pct", fill_value=0)
```

Out[141]:

			day	**Fri**	**Sat**	**Sun**	**Thur**
time	**size**	**smoker**					
Dinner	**1**	**No**	0.000000	0.137931	0.000000	0.000000
		Yes	0.000000	0.325733	0.000000	0.000000
	2	**No**	0.139622	0.162705	0.168859	0.159744
		Yes	0.171297	0.148668	0.207893	0.000000
	3	**No**	0.000000	0.154661	0.152663	0.000000
…	**…**	**…**	…	…	…	…
Lunch	**3**	**Yes**	0.000000	0.000000	0.000000	0.204952
	4	**No**	0.000000	0.000000	0.000000	0.138919
		Yes	0.000000	0.000000	0.000000	0.155410
	5	**No**	0.000000	0.000000	0.000000	0.121389
	6	**No**	0.000000	0.000000	0.000000	0.173706

21 rows × 4 columns

La siguiente tabla muestra un resumen de las opciones de pivot_table

`values` : Nombre o nombres de columna a agregar; por defecto, agrega todas las columnas numéricas

`index` : Nombres de columnas u otras claves de grupo para agrupar en las filas de la tabla dinámica resultante

`columns` : Nombres de columnas u otras claves de grupo para agrupar en las columnas de la tabla dinámica resultante

`aggfunc` : Función de agregación o lista de funciones ("media" por defecto); puede ser cualquier función válida en un contexto groupby

`fill_value` : Sustituir los valores que faltan en la tabla de resultados

`dropna` : Si es True, no incluir columnas cuyas entradas sean todas NA

`margins` : Añadir subtotales de fila/columna y total general (False por defecto)

`margins_name` : Nombre que se utilizará para las etiquetas de fila/columna de margen al pasar margins=True; por defecto es "All".

`observed` : Con claves de grupo categóricas, si es True, mostrar sólo los valores de categoría observados en las claves en lugar de todas las categorías.

Tabulaciones cruzadas: Crosstab

Una tabulación cruzada (cross-tabulation or crosstab) es un caso especial de tabla dinámica (pivot table) que calcula frecuencias de grupo. He aquí un ejemplo:

In [143…
```
from io import StringIO
data = """Sample  Nationality  Handedness
1   USA  Right-handed
2   Japan    Left-handed
3   USA  Right-handed
4   Japan    Right-handed
5   Japan    Left-handed
6   Japan    Right-handed
7   USA  Right-handed
8   USA  Left-handed
9   Japan    Right-handed
10  USA  Right-handed"""
data = pd.read_table(StringIO(data), sep="\s+")
```

In [144…
```
data
```

Out[144]:

	Sample	Nationality	Handedness
0	1	USA	Right-handed
1	2	Japan	Left-handed
2	3	USA	Right-handed
3	4	Japan	Right-handed

4	5	Japan	Left-handed
5	6	Japan	Right-handed
6	7	USA	Right-handed
7	8	USA	Left-handed
8	9	Japan	Right-handed
9	10	USA	Right-handed

Como parte de algún análisis de encuestas, podríamos querer resumir estos datos por nacionalidad y lateralidad. Se podría utilizar pivot_table para hacer esto, pero la función `pandas.crosstab` puede ser más conveniente:

In [145…
```
pd.crosstab(data["Nationality"], data["Handedness"], margins=True)
```

Out[145]:

Handedness	Left-handed	Right-handed	All
Nationality			
Japan	2	3	5
USA	1	4	5
All	3	7	10

Los dos primeros argumentos de `crosstab` pueden ser un array o una serie o una lista de arrays. Como en los datos de tips:

In [146…
```
pd.crosstab([tips["time"], tips["day"]], tips["smoker"], margins=True)
```

Out[146]:

	smoker	No	Yes	All
time	**day**			
Dinner	**Fri**	3	9	12
	Sat	45	42	87
	Sun	57	19	76
	Thur	1	0	1
Lunch	**Fri**	1	6	7
	Thur	44	17	61
All		151	93	244