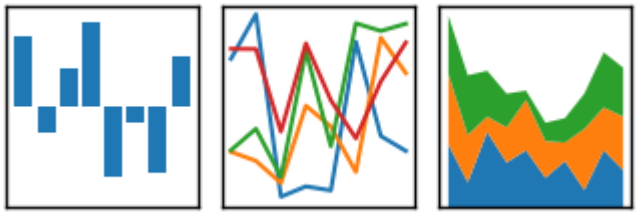
pandas

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



Gestión básica de datos y ficheros

El acceso a los datos es un paso necesario para la mayoría de los procesos y herramientas de análisis. La entrada y la salida generalmente se clasifican en algunas categorías principales: leer archivos de texto y otros formatos en disco más eficientes, cargar datos de bases de datos e interactuar con fuentes de red como las API web.

Leer y escribir datos en formato de texto

Pandas presenta una serie de funciones para leer datos tabulares como un objeto DataFrame. La sigueinte tabla resume algunos de ellos, aunque los más utilizados sean `read_csv` y `read_table` (`deprecate`).

Función	Descripción
read_csv	Carga datos delimitados desde un archivo o URL, la coma (',') es el delimitador predeterminado
read_table	Carga datos delimitados desde un archivo o URL, la tabulación ('\ t') es el delimitador predeterminado
read_fwf	Lee los datos en formato de columna de ancho fijo (sin delimitadores)
read_clipboard	Versión de `read_table` que lee datos del portapapeles
read_excel	Lee datos tabulares de un archivo Excel XLS o XLSX
read_hdf	Lee archivos HDF5 escritos por pandas
read_html	Lee todas las tablas encontradas en el documento HTML dado
read_json	Leer datos con formato JSON
read_msgpack	Leer datos de pandas codificados usando el formato binario de MessagePack
read_pickle	Leer un objeto arbitrario almacenado en formato de pickle de Python
read_sas	Lee un conjunto de datos SAS almacenado con los formatos de almacenamiento personalizados de SAS
read_sql	Lee los resultados de una consulta SQL (usando SQLAlchemy) como un DataFrame de pandas
read_stata	Lee un conjunto de datos de un archivo con formato Stata
read_feather	Lee un conjunto de datos de un archivo con formato binario Feather

De forma general de la mecánica de estas funciones permiten convertir datos de texto en un DataFrame. Los argumentos opcionales para estas funciones pueden caer en algunas categorías:

- Indexación: Puede tratar una o más columnas como el DataFrame devuelto, y si se deben obtener los nombres de las columnas del archivo, el usuario o no.
- Inferencia de tipos y conversión de datos: Esto incluye las conversiones de valor definidas por el usuario y la lista personalizada de marcadores de valor faltantes.
- Análisis de fecha y hora: Incluye la capacidad de combinación, incluida la combinación de información de fecha y hora en varias columnas en una sola columna en el resultado.
- Iteración: Soporte para iterar sobre trozos de archivos muy grandes.

- Limpieza de datos: Omitir filas, comentarios u otras cosas menores como datos numéricos con miles separados por comas.

Debido a lo desordenados que pueden ser los datos en el mundo real, algunas de las funciones de carga de datos (especialmente `read_csv`) se han vuelto muy complejas en sus opciones, por ejemplo `read_csv` tiene más de 50 parámetros diferentes. La documentación de los pandas en línea tiene muchos ejemplos sobre cómo funciona cada uno de ellos, por lo que si tiene dificultades para leer un archivo en particular, puede haber un ejemplo lo suficientemente similar para ayudarlo a encontrar los parámetros correctos.

Algunas de estas funciones, como `read_csv`, realizan inferencia de tipo, porque los tipos de datos forman parte del formato de datos. Eso significa que no necesariamente se tiene que especificar qué columnas son numéricas, enteras, booleanas o de cadena. Otros formatos de datos, como `HDF5`, `Feather` y `MessagePack`, tienen los tipos de datos almacenados en el formato.

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

Out[1]:
```
'0.25.1'
```

In [48]:
```
# cat muestra los contenidos del fichero en pantalla
# windows !type .\data\teoria\ex1.csv
!cat ./data/teoria/ex1.csv
```

```
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

In [49]:
```
# fichero con cabeceras
pd.read_csv('./data/teoria/ex1.csv')
```

Out[49]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [50]:
```
# fichero sin cabeceras
pd.read_csv('./data/teoria/ex2.csv', header=None)
```

Out[50]:

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [51]:
```
headers = ['a1', 'b1', 'c1', 'd1', 'message']
pd.read_csv('./data/teoria/ex2.csv', names=headers)
```

Out[51]:

	a1	b1	c1	d1	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [52]:
```
pd.read_csv('./data/teoria/ex2.csv', names=headers, index_col
='message')
```

Out[52]:

	a1	b1	c1	d1
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

In [53]:
```
# índice jerárquico de múltiples columnas
!cat ./data/teoria/csv_mindex.csv
```

```
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16
```

In [54]:
```
pd.read_csv('./data/teoria/csv_mindex.csv', index_col=['key1', 'key2'])
```

Out[54]:

		value1	value2
key1	**key2**		
one	**a**	1	2
	b	3	4
	c	5	6
	d	7	8
two	**a**	9	10
	b	11	12
	c	13	14
	d	15	16

In [55]:
```
# Sin delimitador fijo (espacios en blanco, retornos de carro...)
list(open('./data/teoria/ex3.txt'))
```

Out[55]:
```
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

In [56]:
```
pd.read_csv('./data/teoria/ex3.txt', sep='\s+')
```

Out[56]:

	A	B	C
aaa	-0.264438	-1.026059	-0.619500
bbb	0.927272	0.302904	-0.032399
ccc	-0.264273	-0.386314	-0.217601
ddd	-0.871858	-0.348382	1.100491

Debido a que hay un nombre de columna menos que el número de filas de datos, `read_csv` deduce que la primera columna debe ser el índice de DataFrame.

In [57]:
```
# eliminando filas no válidas del fichero
!cat ./data/teoria/ex4.csv
```

```
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

In [58]:
```
pd.read_csv('./data/teoria/ex4.csv', skiprows=[0, 2, 3])
```

Out[58]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [59]:
```
# Manejando datos no presentes o nulos
!cat ./data/teoria/ex5.csv
```

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,x,10,x,12,foo
```

In [60]:
```
pd.read_csv('./data/teoria/ex5.csv')
```

Out[60]:

	something	a	b	c	d	message
0	one	1	2	3	4	NaN
1	two	5	6	NaN	8	world
2	three	x	10	x	12	foo

In [61]:
```
pd.read_csv('./data/teoria/ex5.csv', na_values=['x'])
```

Out[61]:

	something	a	b	c	d	message
0	one	1.0	2	3.0	4	NaN
1	two	5.0	6	NaN	8	world
2	three	NaN	10	NaN	12	foo

Lectura de archivos de texto por partes

Cuando se procesan archivos muy grandes es posible que solo se desee leer un fragmento pequeño de un archivo o iterar a través de fragmentos más pequeños del archivo.

In [62]:
```
# configuración de visualización de pandas más compacta
pd.options.display.max_rows = 10
```

In [63]:
```
pd.read_csv('./data/teoria/ex6.csv')
```

Out[63]:

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q
...	...	...	...	...	...
9995	2.311896	-0.417070	-1.409599	-0.515821	L
9996	-0.479893	-0.650419	0.745152	-0.646038	E
9997	0.523331	0.787112	0.486066	1.093156	K
9998	-0.362559	0.598894	-1.843201	0.887292	G
9999	-0.096376	-1.012999	-0.657431	-0.573315	0

10000 rows × 5 columns

In [64]:
```
# leer un número específico de filas
pd.read_csv('./data/teoria/ex6.csv', nrows=5)
```

Out[64]:

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q

In [65]:
```
# leer un número específico de filas en base a un tamaño dado
pd.read_csv('./data/teoria/ex6.csv', chunksize=1000)
```

Out[65]:
```
<pandas.io.parsers.TextFileReader at 0x115c78588>
```

El objeto `TextParser` devuelto por `read_csv` permite iterar sobre las partes del archivo de acuerdo con el tamaño fijado en `chunksize` .

In [66]:
```
chunker = pd.read_csv('./data/teoria/ex6.csv', chunksize=100
0)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
tot[:10]
```

Out[66]:
```
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

`TextParser` también está equipado con un método `get_chunk` que permite leer piezas de un tamaño arbitrario.

Escribir datos en formato de texto

De forma similar a la que se pueden recuperar datos desde ficheros, los datos también se pueden exportar a un formato texto (delimitado por comas por defecto) usando el método `to_csv` de DataFrame y Series.

In [67]:
```
data = pd.read_csv('./data/teoria/ex5.csv', na_values=['x'])
data
```

Out[67]:

	something	a	b	c	d	message
0	one	1.0	2	3.0	4	NaN
1	two	5.0	6	NaN	8	world
2	three	NaN	10	NaN	12	foo

Podemos definir un separador distinto en el parámetro `sep` y tratar los campos faltantes o nulos a la hora de escribir el fichero usando el parámetro `na_rep` :

In [68]:
```
data.to_csv('./data/out/ex5_out.csv', sep=';', na_rep='-')
!cat ./data/out/ex5_out.csv
```

```
;something;a;b;c;d;message
0;one;1.0;2;3.0;4;-
1;two;5.0;6;-;8;world
2;three;-;10;-;12;foo
```

Sin otras opciones especificadas, se escriben las etiquetas de filas y columnas. Dichas opciones pueden ser deshabilitadas:

In [69]:
```
data.to_csv('./data/out/ex5b_out.csv', index=False, header=False, na_rep='NULL')
!cat ./data/out/ex5b_out.csv
```

```
one,1.0,2,3.0,4,NULL
two,5.0,6,NULL,8,world
three,NULL,10,NULL,12,foo
```

También se puede seleccionar un subconjunto de las columas disponibles:

In [70]:
```
data.to_csv('./data/out/ex5c_out.csv', index=False, columns=['a', 'b', 'c'])
!cat ./data/out/ex5c_out.csv
```

```
a,b,c
1.0,2,3.0
5.0,6,
,10,
```

Datos JSON

JSON (abreviatura de *JavaScript Object Notation*) se ha convertido en uno de los formatos estándar para enviar datos mediante solicitud HTTP entre navegadores web y otras aplicaciones. Es un formato de datos mucho más libre que un formato de texto tabular como CSV.

JSON es un código Python casi válido, con la excepción de su valor nulo 'null' y algunos otros matices (como no permitir comas al final de las listas). Los tipos básicos son objetos (dicts), matrices (lists), cadenas, números, booleanos y nulos. Todas las claves de un objeto deben ser cadenas.

Hay varias bibliotecas de Python para leer y escribir datos JSON. La biblioteca estándar de Python cuenta con `json`. Para convertir una cadena JSON a Python, se utiliza `json.loads`:

In [71]:
```
obj = """
    {"name": "Wes",
     "places_lived": ["United States", "Spain", "Germany"],
     "pet": null,
     "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeu
s", "Zuko"]},
                  {"name": "Katie", "age": 38,
                   "pets": ["Sixes", "Stache", "Cisco"]}]
} """

import json
result = json.loads(obj)
result
```

Out[71]:
```
{'name': 'Wes',
 'places_lived': ['United States', 'Spain', 'Germany'],
 'pet': None,
 'siblings': [{'name': 'Scott', 'age': 30, 'pets': ['Zeus',
'Zuko']},
  {'name': 'Katie', 'age': 38, 'pets': ['Sixes', 'Stache', '
Cisco']}]}
```

Por otra parte, `json.dumps` convierte un objeto Python en una cadena JSON:

In [72]:
```
asjson = json.dumps(result)
asjson
```

Out[72]:
```
'{"name": "Wes", "places_lived": ["United States", "Spain",
"Germany"], "pet": null, "siblings": [{"name": "Scott", "ag
e": 30, "pets": ["Zeus", "Zuko"]}, {"name": "Katie", "age":
38, "pets": ["Sixes", "Stache", "Cisco"]}]}'
```

Una forma habitual de pasar un objeto JSON a un DataFrame o alguna otra estructura de datos para su análisis es manejarlos como una lista de diccionarios y seleccionar un subconjunto de los campos de datos:

In [73]:
```
pd.DataFrame(result['siblings'], columns=['name', 'age'])
```

Out[73]:

	name	age
0	Scott	30
1	Katie	38

La función `read_json` pueden convertir automáticamente los conjuntos de datos JSON en arrays específicos en una serie o en un marco de datos. Las opciones predeterminadas asumen que cada objeto en la matriz JSON es una fila en la tabla resultante:

In [74]:
```
!cat ./data/teoria/ex7.json
```

```
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

In [75]:
```
data = pd.read_json('./data/teoria/ex7.json')
data
```

Out[75]:

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

In [76]:
```
# Respuesta JSON manipulable desde Pandas
data = pd.read_json('http://abraham.etsisi.upm.es/api-test/fichas/v1/alumnos', dtype=object)
data.drop(columns=['NUMREG'], inplace=True)
data.drop([0], inplace=True)
data
```

Out[76]:

	Apellidos	DNI	Direccion	Equipo	Nombre	Telefono
1	Sancho Manrique	04864868	Zurriaga, 25	20	Beatriz	93-232-12-12
2	Cañas Cañas	07884898	Vinateros, 121	40	Joaquín	93-258-25-22
3	Camargo Román	15756214	Armadores, 1	7	Miguel	949-48-85-88
4	Alcocer Jarabo	34225234	Leonor de Cortinas, 7	9	Alejandro	91-532-12-11
5	Ramírez Audige	38624852	Fuencarral, 33	10	Alvaro	91-245-11-68
6	Pérez del Olmo	45824852	Cervantes, 22	11	Rocío	91-233-21-38
7	Bobadilla Sancho	48488588	Gaztambique, 32	13	Jesús	91-314-11-11
8	Gutiérrez Rodríguez	55555555	Km. 7 Carretera de Valencia	13	Abraham	91-336-78-63

Para exportar datos datos desde pandas a JSON se puede utilizar el método `to_json` :

In [77]:
```
data[['DNI', 'Nombre', 'Apellidos']].to_json(force_ascii=False)
```

Out[77]:

```
'{"DNI":{"1":"04864868","2":"07884898","3":"15756214","4":"34225234","5":"38624852","6":"45824852","7":"48488588","8":"55555555"},"Nombre":{"1":"Beatriz","2":"Joaquín","3":"Miguel","4":"Alejandro","5":"Alvaro","6":"Rocío","7":"Jesús","8":"Abraham"},"Apellidos":{"1":"Sancho Manrique","2":"Cañas Cañas","3":"Camargo Román","4":"Alcocer Jarabo","5":"Ramírez Audige","6":"Pérez del Olmo","7":"Bobadilla Sancho","8":"Gutiérrez Rodríguez"}}'
```


In [78]:
```
data[['DNI', 'Nombre', 'Apellidos']].to_json(orient='records', force_ascii=False)
```

Out[78]:

```
'[{"DNI":"04864868","Nombre":"Beatriz","Apellidos":"Sancho Manrique"},{"DNI":"07884898","Nombre":"Joaquín","Apellidos":"Cañas Cañas"},{"DNI":"15756214","Nombre":"Miguel","Apellidos":"Camargo Román"},{"DNI":"34225234","Nombre":"Alejandro","Apellidos":"Alcocer Jarabo"},{"DNI":"38624852","Nombre":"Alvaro","Apellidos":"Ramírez Audige"},{"DNI":"45824852","Nombre":"Rocío","Apellidos":"Pérez del Olmo"},{"DNI":"48488588","Nombre":"Jesús","Apellidos":"Bobadilla Sancho"},{"DNI":"55555555","Nombre":"Abraham","Apellidos":"Gutiérrez Rodríguez"}]'
```


In [79]:
```
import urllib.request

url = 'http://abraham.etsisi.upm.es/api-test/fichas/v1/alumnos'
with urllib.request.urlopen(url) as url:
    json_data = json.loads(url.read().decode())

data = pd.DataFrame(json_data[1:])
data
```

Out[79]:

	Apellidos	DNI	Direccion	Equipo	Nombre	Telefono
0	Sancho Manrique	04864868	Zurriaga, 25	20	Beatriz	93-232-12-12
1	Cañas Cañas	07884898	Vinateros, 121	40	Joaquín	93-258-25-22
2	Camargo Román	15756214	Armadores, 1	7	Miguel	949-48-85-88
3	Alcocer Jarabo	34225234	Leonor de Cortinas, 7	9	Alejandro	91-532-12-11
4	Ramírez Audige	38624852	Fuencarral, 33	10	Alvaro	91-245-11-68
5	Pérez del Olmo	45824852	Cervantes, 22	11	Rocío	91-233-21-38
6	Bobadilla Sancho	48488588	Gaztambique, 32	13	Jesús	91-314-11-11
7	Gutiérrez Rodríguez	55555555	Km. 7 Carretera de Valencia	13	Abraham	91-336-78-63

In [80]:
```
# Respuesta JSON no manipulable desde Pandas
url = 'https://www.googleapis.com/books/v1/volumes?q=intitle:
python'
with urllib.request.urlopen(url) as url:
    json_data = json.loads(url.read().decode())

keys_selected = ['title', 'authors', 'publisher', 'publishedD
ate', 'description']
'''
# Creando dicccionarios
data = pd.DataFrame()
for item in json_data['items']:
    item_dict = {key:item['volumeInfo'].get(key, None) for ke
y in keys_selected}
    data = data.append(item_dict, ignore_index=True)
'''
# Ajustando columnas
volumes = [item['volumeInfo'] for item in json_data['items']]
data = pd.DataFrame(volumes, columns=keys_selected)

'''
# json_normalize
from pandas.io.json import json_normalize
data = json_normalize(data = json_data,
                      record_path = 'items',
                      meta = ['totalItems'],
                      errors ='ignore')
'''
data
```

Out[80]:

	title	authors	publisher	publishedDate	description
0	Python para humanos	[feNiX10ist]	feNiX10ist	2014-07-27	Python para humanos es un libro que tiene como...
1	Aprende a Programar en Python	[Ángel Arias]	IT Campus Academy	2015-06-19	sí que no habéis programado nunca... A medida ...
2	Programar con Python 3	[Alberto Cuevas çlvarez]	Lulu.com	2019-10-17	Libro sobre programación en Python 3, con más ...
3	Nociones Básicas de Python	[Miguel Iván Bobadilla]	Miguel Iván Bobadilla	NaN	Libro para aprender a programar en lenguaje de...
4	Python Paso a paso	[Ángel Pablo Hinojosa Gutiérrez]	Grupo Editorial RA-MA	NaN	En los últimos años, el lenguaje de programaci...
5	Python 3 al descubierto - 2a ed.	[Arturo FERNANDEZ]	Alfaomega Grupo Editor	2013-09-30	Se ofrece un repaso a las principales caracter...
6	Programación en Python I	[Celeste Guagliano]	RedUsers	2019-09-03	Python es un lenguaje de programación multipla...
7	Programación en Python II	[Celeste Guagliano]	RedUsers	2019-10-02	Python es un lenguaje de programación multipla...
8	micro:bit y Python (Edici—n en Blanco y Negro)	[J.C. Bautista]	Lulu.com	2018-03-28	Aprende a programar en Python divirtiéndote co...
9	Hacking ético con herramientas Python	[José Manuel Ortega Candel]	Grupo Editorial RA-MA	NaN	En los últimos años, Python se ha convertido e...

HTML: Web Scraping

Python tiene muchas bibliotecas para leer y escribir datos en los formatos ubicuos de HTML y XML. Los ejemplos incluyen `lxml` , `Beautiful Soup` y `html5lib` . Pandas tiene una función incorporada, `read_html` , que utiliza bibliotecas como las anteriores para analizar automáticamente las tablas de los archivos HTML como objetos DataFrame. La función `read_html` tiene varias opciones, pero por defecto busca e intenta analizar todos los datos tabulares contenidos en las etiquetas `<table>` . El resultado es una lista de objetos DataFrame:

In [81]:
```
tables = pd.read_html('./data/teoria/fdic_failed_bank_list.html')
```

In [82]:
```
tables[0].head()
```

Out[82]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0	Allied Bank	Mulberry	AR	91	Today's Bank	September 23, 2016	November 17, 2016
1	The Woodbury Banking Company	Woodbury	GA	11297	United Bank	August 19, 2016	November 17, 2016
2	First CornerStone Bank	King of Prussia	PA	35312	First-Citizens Bank & Trust Company	May 6, 2016	September 6, 2016
3	Trust Company Bank	Memphis	TN	9956	The Bank of Fayette County	April 29, 2016	September 6, 2016
4	North Milwaukee State Bank	Milwaukee	WI	20364	First-Citizens Bank & Trust Company	March 11, 2016	June 16, 2016

Leyendo archivos de Microsoft Excel

Pandas también admite la lectura y escritura de datos tabulares en archivos de Excel 2003 (y superiores) utilizando las clases `ExcelFile` y `ExcelWriter` o las funciones `read_excel` y `to_excel` . Internamente, estas herramientas utilizan los paquetes complementarios `xlrd` y `openpyxl` para leer archivos XLS y XLSX, respectivamente (puede que se necesite instalarlos manualmente con pip o conda).

In [83]:
```
# Crear un objeto ExcelFile es más óptimo cuando se están rea
lizando múltiples lecturas
xlsx = pd.ExcelFile('./data/teoria/ex8.xlsx')
frame = pd.read_excel(xlsx, 'Sheet1', index_col=0)
frame
```

Out[83]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [84]:
```
frame = pd.read_excel('./data/teoria/ex8.xlsx', 'Sheet1', ind
ex_col=0)
frame
```

Out[84]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Para escribir datos de pandas en formato Excel, primero debe crear un `ExcelWriter` , luego escribir datos en él utilizando el método `to_excel` de los objetos de pandas:

In [85]:
```
# Crear un objeto ExcelWriter es más óptimo cuando se están r
ealizando múltiples escrituras
# writer = pd.ExcelWriter('./data/out/ex2.xlsx')
# frame.to_excel(writer, 'Sheet1')
# writer.save()
frame.to_excel('./data/out/ex8_out.xlsx')
```

Interactuando con Bases de Datos

En una configuración empresarial, la mayoría de los datos no se pueden almacenar en archivos de texto o Excel. Las bases de datos relacionales basadas en SQL (como SQL Server, PostgreSQL y MySQL) se usan ampliamente. Cargar datos de SQL en un DataFrame es bastante sencillo, y `pandas` tienen algunas funciones para simplificar el proceso.

In [86]:
```
# base de datos SQLite utilizando el controlador sqlite3 incorporado de Python
import sqlite3
con = sqlite3.connect('./data/out/mydata.sqlite')

# Creamos la tabla
query = """
    CREATE TABLE IF NOT EXISTS test
       (a VARCHAR(20),
        b VARCHAR(20),
        c REAL,
        d INTEGER
       );
"""
con.execute(query)
con.commit()

# Insertamos datos
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
con.executemany(stmt, data)
con.commit()
```

La mayoría de los controladores SQL de Python (PyODBC, psycopg2, MySQLdb, pymssql, etc.) devuelven una lista de tuplas al seleccionar datos de una tabla:

In [87]:
```
# recuperamos los datos
cursor = con.execute('select * from test')
rows = cursor.fetchall()
rows
```

Out[87]:
```
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5),
 ('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

Se puede pasar la lista de tuplas al constructor del DataFrame, pero también se necesitan los nombres de las columnas, contenidos en la descripción del cursor:

In [88]:
```
cursor.description
```

Out[88]:
```
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))
```

In [89]:
```
pd.DataFrame(rows, columns=[x[0] for x in cursor.descriptio
n])
```

Out[89]:

	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5
3	Atlanta	Georgia	1.25	6
4	Tallahassee	Florida	2.60	3
5	Sacramento	California	1.70	5

In [90]:
```
con.close()
```

El proyecto `SQLAlchemy` es un popular kit de herramientas SQL de Python que extrae muchas de las diferencias comunes entre las bases de datos SQL. pandas tiene una función `read_sql` que permite leer datos fácilmente desde una conexión general de SQLAlchemy.

In [91]:
```
import sqlalchemy as sqla
db = sqla.create_engine('sqlite:///data/out/mydata.sqlite')
pd.read_sql('select * from test', db)
```

Out[91]:

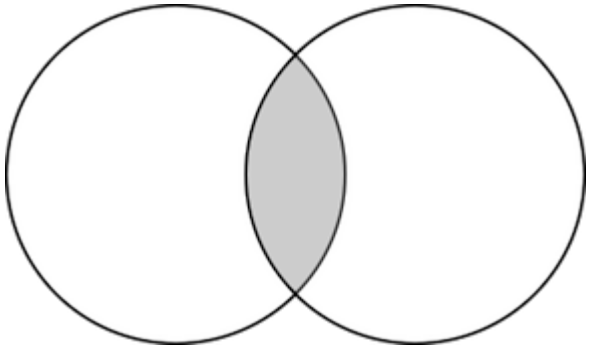
	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5
3	Atlanta	Georgia	1.25	6
4	Tallahassee	Florida	2.60	3
5	Sacramento	California	1.70	5

In [92]:
```
db.dispose()
```

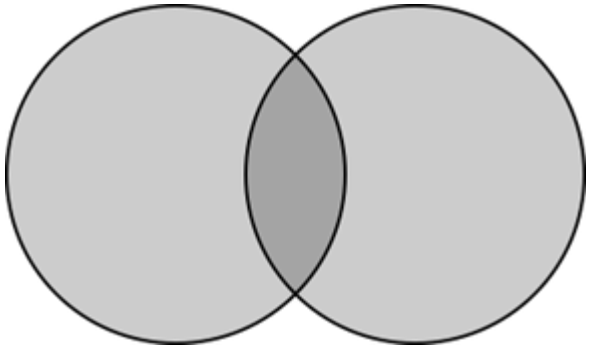
Fusionar, unir y concatenar

pandas ofrece varias facilidades para combinar fácilmente Series y DataFrame con varios tipos de lógica de conjuntos para los índices y la funcionalidad de álgebra relacional en el caso de operaciones de unión/concatenación. La lógica de la unión permite diferentes resultados, de forma general las opciones más habituales son:

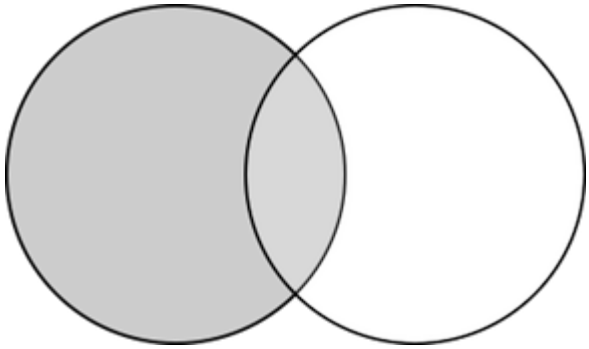
- **Inner Join**: Intersección



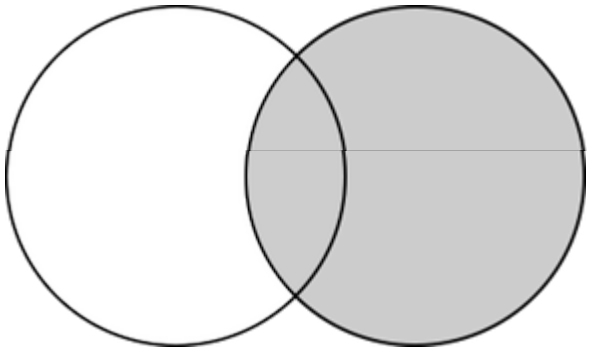
- **Outer Join** (*full outer join*): Unión de todas las filas. El DataFrame resultante puede tener valores faltantes.



- **Left Join**: Se mantienen todas las filas de la izquierda y se incoporan los datos de la derecha para las claves seleccionadas (intersección). El DataFrame resultante puede tener valores faltantes en el lado derecho.



- **Right Join**: Se mantienen todas las filas de la derecha y se incoporan los datos de la izquierda para las claves seleccionadas (intersección). El DataFrame resultante puede tener valores faltantes en el lado izquierdo.



Concatenar objetos

La función `concat` hace todo el trabajo pesado de realizar operaciones de concatenación a lo largo de un eje mientras realiza la lógica de conjunto opcional (unión o intersección) de los índices (si los hay) en los otros ejes. Similar a la función `numpy.concatenate` en ndarrays, `pandas.concat` toma una lista o diccionario de objetos tipificados homogéneamente y los concatena con un manejo configurable de "qué hacer con los otros ejes":

```
pd.concat(objs, axis=0, join='outer', join_axes=None,
          ignore_inde x=False,

          keys=None, levels=None, names=None,

          verify_integrity=False copy=True)
```

In [93]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                   index=[4, 5, 6, 7])

df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])

result = pd.concat([df1, df2, df3])
result
```

Out[93]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
...	...	...	...	...
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

12 rows × 4 columns

Supongamos que quisiéramos asociar claves específicas con cada una de las piezas del DataFrame cortado. Podemos hacer esto usando el argumento claves:

In [94]:
```
# concatenación creando nuevas claves
# equivalente pd.concat({'x': df1, 'y': df2, 'z': df3})
result = pd.concat([df1, df2, df3], keys=['x', 'y', 'z'])
result
```

Out[94]:

		A	B	C	D
x	0	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
	...	...	...	...	...
	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

12 rows × 4 columns

In [95]:
```
result.index.levels
```

Out[95]:
```
FrozenList([['x', 'y', 'z'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11]])
```

In [96]:
```
result.loc['y']
```

Out[96]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Un atajo útil para `concat` es el método de instancia de `append` en Series y DataFrame que concatena a lo largo del `eje = 0` , es decir, usando el índice de la Serie o el DataFrame.

> **Nota**: Vale la pena señalar que `concat` (y, por lo tanto, `append`) hace una copia completa de los datos, y que la reutilización constante de esta función puede crear un impacto significativo en el rendimiento. Si necesita usar la operación sobre varios conjuntos de datos, use una lista de comprensión.

Establecer la lógica en los otros ejes

Al pegar varios DataFrames, se puede elegir cómo manejar los otros ejes (aparte del que está concatenado). Esto se puede hacer de las siguientes tres maneras:

- La unión de todos ellos, `join='outer'` . Esta es la opción predeterminada, ya que da como resultado una pérdida de información cero.
- La intersección, `join='inner'` .
- La utilización de un índice específico, pasado al argumento `join_axes` .

In [97]:
```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])

result = pd.concat([df1, df4], axis=1, sort=False)
result
```

Out[97]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

In [98]:
```
result = pd.concat([df1, df4], axis=1, join='inner')
result
```

Out[98]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

In [99]:
```
result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
result
```

Out[99]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Ignorar los índices en el eje de concatenación

Para los objetos DataFrame que no tienen un índice significativo, es posible que se desee agregarlos e ignorar el hecho de que pueden tener índices superpuestos. Para hacer esto, usa el argumento ignore_index:

In [100]:
```
result = pd.concat([df1, df4], ignore_index=True, sort=False)
result
```

Out[100]:

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	NaN	B2	NaN	D2	F2
5	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Concatenando con dimensiones diferentes

Se puede concatenar una mezcla de objetos Series y DataFrame.

In [101]:
```
# en una serie con nombre el nombre de la Serie se transforma
en el nombre de la columna
s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')
result = pd.concat([df1, s1], axis=1)
result
```

Out[101]:

	A	B	C	D	X
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

In [102]:
```
# las series sin nombre hacen que se numeren consecutivamente las nuevas columnas
s2 = pd.Series(['_0', '_1', '_2', '_3'])
result = pd.concat([df1, s2, s2, s2], axis=1)
result
```

Out[102]:

	A	B	C	D	0	1	2
0	A0	B0	C0	D0	_0	_0	_0
1	A1	B1	C1	D1	_1	_1	_1
2	A2	B2	C2	D2	_2	_2	_2
3	A3	B3	C3	D3	_3	_3	_3

In [103]:
```
# si se ingnoran los índices, los nombres de las columnas no se tiene en cuenta
result = pd.concat([df1, s1], axis=1, ignore_index=True)
result
```

Out[103]:

	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

Concatenando con claves de grupo

Un uso bastante común del argumento `keys` es anular los nombres de columna al crear un nuevo DataFrame basado en una serie existente.

In [104]:
```
s3 = pd.Series([0, 1, 2, 3], name='foo')
s4 = pd.Series([0, 1, 2, 3])
s5 = pd.Series([0, 1, 4, 5])
pd.concat([s3, s4, s5], axis=1)
```

Out[104]:

	foo	0	1
0	0	0	0
1	1	1	1
2	2	2	4
3	3	3	5

In [105]:
```
pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow
'])
```

Out[105]:

	red	blue	yellow
0	0	0	0
1	1	1	1
2	2	2	4
3	3	3	5

In [106]:
```
pieces = {'x': df1, 'y': df2, 'z': df3}
result = pd.concat(pieces)
result
```

Out[106]:

		A	B	C	D
x	**0**	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	**4**	A4	B4	C4	D4
	...	...	...	...	...
	7	A7	B7	C7	D7
z	**8**	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

12 rows × 4 columns

In [107]:
```
result.index.levels
```

Out[107]:
```
FrozenList([['x', 'y', 'z'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11]])
```

In [108]:
```
pd.concat(pieces, keys=['z', 'y'])
```

Out[108]:

		A	B	C	D
z	**8**	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11
y	**4**	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7

In [109]:
```
result.loc[['z', 'y']]
```

Out[109]:

		A	B	C	D
y	**4**	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7
z	**8**	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

Unión/fusión de Series con nombre y DataFrames como bases de datos

pandas tiene operaciones de unión en memoria de alto rendimiento con todas las funciones idiomáticamente muy similares a las bases de datos relacionales como SQL. Estos métodos tienen un rendimiento significativamente mejor (en algunos casos, más de un orden de magnitud mejor) que otras implementaciones de código abierto (como `base::merge.data.frame` en `R`). La razón de esto es el cuidadoso diseño algorítmico y el diseño interno de los datos en DataFrame.

pandas proporciona una función única, `merge`, como punto de entrada para todas las operaciones de unión de base de datos estándar entre DataFrame o Series con nombre:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_o
n=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
```

Breve introducción sobre los métodos de fusión (álgebra relacional)

Los usuarios experimentados de bases de datos relacionales como SQL estarán familiarizados con la terminología utilizada para describir las operaciones de unión entre dos estructuras similares a tablas SQL (objetos DataFrame). Hay varios casos a considerar que son muy importantes de entender:

- uniones **uno a uno**: por ejemplo, al unir dos objetos DataFrame sobre sus índices (que deben contener valores únicos).
- uniones **muchos a uno**: por ejemplo, al unir un índice (único) a una o más columnas en un DataFrame diferente.
- uniones **muchos a muchos**: por ejemplo, al unir columnas con columnas.

In [110]:
```
# ejemplo básico con una combinación de claves únicas
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})

result = pd.merge(left, right, on='key')
result
```

Out[110]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

In [111]:
```
# combinación con múltiples clave (intersección de claves por defecto, how='inner')
```

In [112]:
```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                      'key2': ['K0', 'K0', 'K0', 'K0'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})

result = pd.merge(left, right, on=['key1', 'key2'])
result
```

Out[112]:

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

El argumento `how` especifica cómo determinar qué claves se incluirán en la tabla resultante. Si una combinación de claves que no aparece en las tablas izquierda o derecha, los valores en la tabla combinada serán `NA` . La siguiente tabla resume las opciones y equivalentes de SQL:

Método	SQL	Decripción
left	LEFT OUTER JOIN	Utiliza sólo las claves de la izquierda
right	RIGHT OUTER JOIN	Utiliza sólo las claves de la derecha
outer	FULL OUTER JOIN	Utiliza la unión de las claves
inner	INNER JOIN	Utiliza la intersección de las claves

Buscando claves duplicadas

Los usuarios pueden usar el argumento `validate` para verificar automáticamente si hay duplicados inesperados en sus claves de combinación. La unicidad de la clave se verifica antes de las operaciones de combinación y, por lo tanto, protege contra los desbordamientos de memoria. La verificación de la singularidad de las claves también es una buena forma de garantizar que las estructuras de datos de los usuarios sean las esperadas.

In [113]:
```
# valores duplicados de B en el DataFrame derecho
left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})
right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2, 2]})
result = pd.merge(left, right, on='B', how='outer', validate
="one_to_one")
```

```
---------------------------------------------------------------
---------------
MergeError                                Traceback (most re
cent call last)
<ipython-input-113-d3725e30670c> in <module>
      2 left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})
      3 right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2,
2]})
----> 4 result = pd.merge(left, right, on='B', how='outer',
validate="one_to_one")

~/anaconda3/lib/python3.7/site-packages/pandas/core/reshape/
merge.py in merge(left, right, how, on, left_on, right_on, l
eft_index, right_index, sort, suffixes, copy, indicator, val
idate)
     45                          right_index=right_index, so
rt=sort, suffixes=suffixes,
     46                          copy=copy, indicator=indica
tor,
---> 47                          validate=validate)
     48     return op.get_result()
     49

~/anaconda3/lib/python3.7/site-packages/pandas/core/reshape/
merge.py in __init__(self, left, right, how, on, left_on, ri
ght_on, axis, left_index, right_index, sort, suffixes, copy,
indicator, validate)
    537         # are in fact unique.
    538         if validate is not None:
--> 539             self._validate(validate)
    540
    541     def get_result(self):

~/anaconda3/lib/python3.7/site-packages/pandas/core/reshape/
merge.py in _validate(self, validate)
   1082                                     " not a one-to-one
merge")
   1083             elif not right_unique:
-> 1084                 raise MergeError("Merge keys are not
unique in right dataset;"
   1085                                     " not a one-to-one
merge")
   1086

MergeError: Merge keys are not unique in right dataset; not
a one-to-one merge
```

Si el usuario conoce los duplicados en el DataFrame derecho pero quiere asegurarse de que no haya duplicados en el DataFrame izquierdo, se puede usar el argumento `validate = 'one_to_many'` en su lugar, lo que no generará una excepción.

In [114]:
```
# valores duplicados de B en el DataFrame derecho
pd.merge(left, right, on='B', how='outer', validate="one_to_many")
```

Out[114]:

	A_x	B	A_y
0	1	1	NaN
1	2	2	4.0
2	2	2	5.0
3	2	2	6.0

Si el argumento `indicator` es `True`, se agregará una columna de tipo categórico llamada `_merge` al objeto de salida que informa sobre el tipo de operación realizada. También admite argumentos de cadena para poner nombre a la columna del indicador.

La fusión conservará el `dtype` de las claves de unión. Si la columna tiene valores `NA` el dtype resultante será upcast.

Uniones con índices y columnas

El método `join` permite combinar las columnas de dos DataFrames indexados potencialmente diferentes en un DataFrame de un solo resultado.

In [115]:
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                    index=['K0', 'K1', 'K2'])

right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                     index=['K0', 'K2', 'K3'])

result = left.join(right)
result
```

Out[115]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

La alineación de los datos aquí está en los índices (etiquetas de fila). Este mismo comportamiento se puede lograr mediante `merge` más argumentos adicionales que le indiquen que use los índices:

In [116]:
```
# result = pd.merge(left, right, left_index=True, right_index
=True, how='inner')
result = left.join(right, how='inner')
result
```

Out[116]:

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

In [117]:
```
# result = pd.merge(left, right, left_index=True, right_index
=True, how='outer')
result = left.join(right, how='outer')
result
```

Out[117]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

El método `join` tiene el argumento opcional `on` que puede ser una columna o varios nombres de columna, y que especifica que el DataFrame pasado se alineará en esa columna en el DataFrame.

In [118]:
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3'],
                     'key': ['K0', 'K1', 'K0', 'K1']})

right = pd.DataFrame({'C': ['C0', 'C1'],
                      'D': ['D0', 'D1']},
                     index=['K0', 'K1'])

# result = pd.merge(left, right, left_on='key', right_index=True, how='left', sort=False)
result = left.join(right, on='key')
result
```

Out[118]:

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

Agrupar: dividir-aplicar-combinar

Por "agrupar por" (*group by*) nos referimos a un proceso que involucra uno o más de los siguientes pasos:

- Dividir los datos en grupos según algunos criterios.
- Aplicar una función a cada grupo de forma independiente.
- Combinar los resultados en una estructura de datos.

De éstos, el paso de división es el más sencillo. De hecho, en muchas situaciones podemos desear dividir el conjunto de datos en grupos y hacer algo con esos grupos. En el paso de aplicación, podríamos desear uno de los siguientes:

- Agregación: calcular una estadística de resumen (o estadísticas) para cada grupo. Algunos ejemplos:
  - Calcular sumas o medios grupales.
  - Calcular los tamaños / conteos de los grupos.
- Transformación: realizar algunos cálculos específicos del grupo y devolver un objeto indexado. Algunos ejemplos:
  - Estandarizar los datos (`zscore`) dentro de un grupo.
  - Rellenar `NA` dentro de grupos con un valor derivado de cada grupo.
- Filtración: descartar algunos grupos, de acuerdo con un cálculo grupal que evalúa `True` o `False` . Algunos ejemplos:
  - Descartar los datos que pertenecen a grupos con solo unos pocos miembros.
  - Filtrar los datos según la suma o la media del grupo.
- Alguna combinación de lo anterior: GroupBy examinará los resultados del paso de aplicación e intentará obtener un resultado sensiblemente combinado si no encaja en ninguna de las dos categorías anteriores.

Dividir un objeto en grupos

Los objetos pandas se pueden dividir en cualquiera de sus ejes. La definición abstracta de agrupación es proporcionar una asignación de etiquetas a nombres de grupo.

El mapeo a grupos se puede especificar de muchas maneras diferentes:

- Una función de Python, para ser llamada en cada una de las etiquetas de eje.
- Una lista o matriz NumPy de la misma longitud que el eje seleccionado.
- Un diccionario de Series, proporcionando un mapeo `etiqueta -> grupo` .
- Para los objetos DataFrame, una cadena que indica una columna que se utilizará para agrupar. Por supuesto, `df.groupby ('A')` es solo azúcar sintáctica para `df.groupby (df ['A'])` , pero hace la vida más sencilla.
- Para los objetos DataFrame, una cadena que indica un nivel de índice que se utilizará para agrupar.
- Una lista de cualquiera de las cosas anteriores.

In [25]:
```
from numpy import nan as NA
data = {'animal': ['cat', 'dog', 'bat', 'penguin'],
        'num_legs': [4, 4, 2, 2],
        'num_wings': [0, 0, 2, 2],
        'class': ['mammal', 'mammal', 'mammal', 'bird'],
        'locomotion': ['walks', 'walks', 'flies', 'walks']}
df = pd.DataFrame(data)
df
```

Out[25]:

	animal	num_legs	num_wings	class	locomotion
0	cat	4	0	mammal	walks
1	dog	4	0	mammal	walks
2	bat	2	2	mammal	flies
3	penguin	2	2	bird	walks

In [7]:
```
# por defecto axis=0
grouped = df.groupby('class')
grouped
```

Out[7]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11
4128ad0>
```

In [8]:
```
grouped.groups
```

Out[8]:
```
{'bird': Int64Index([3], dtype='int64'),
 'mammal': Int64Index([0, 1, 2], dtype='int64')}
```

In [9]:
```
len(grouped)
```

Out[9]:
```
2
```

In [10]:
```
for name, group in grouped:
    print('Grupo: ' + str(name))
    print("-" * 40)
    print(group, end="\n\n")
```

```
Grupo: bird
----------------------------------------
    animal  num_legs  num_wings class locomotion
3  penguin         2          2  bird      walks

Grupo: mammal
----------------------------------------
  animal  num_legs  num_wings   class locomotion
0    cat         4          0  mammal      walks
1    dog         4          0  mammal      walks
2    bat         2          2  mammal      flies
```

```
In [12]: grouped.get_group('mammal')
```

Out[12]:

	animal	num_legs	num_wings	class	locomotion
0	cat	4	0	mammal	walks
1	dog	4	0	mammal	walks
2	bat	2	2	mammal	flies

Una vez que se ha creado el grupo (objeto GroupBy), hay múltiples métodos disponibles para realizar un cálculo en los datos agrupados:

grouped. agg() boxplot() cummax() expanding() head() max() ngroups pipe() sem() take aggregate() class cummin() ffill() hist max_speed nth() plot shift() transform() all() corr cumprod() fillna idxmax mean() nunique() prod() size() tshift any() corrwith cumsum() filter() idxmin median() ohlc() quantile skew var() apply() count() describe() first() indices min() order rank() std() backfill() cov diff get_group() last() ndim pad() resample() sum() bfill() cumcount() dtypes groups mad ngroup() pct_change() rolling() tail()

```
In [13]: grouped.size()
```

Out[13]:

```
class
bird      1
mammal    3
dtype: int64
```

```
In [17]: grouped.max()
```

Out[17]:

	animal	num_legs	num_wings	locomotion
class				
bird	penguin	2	2	walks
mammal	dog	4	2	walks

```
In [18]: grouped.describe()
```

Out[18]:

	num_legs								num_wings	
	count	mean	std	min	25%	50%	75%	max	count	mean
class										
bird	1.0	2.000000	NaN	2.0	2.0	2.0	2.0	2.0	1.0	2.000000
mammal	3.0	3.333333	1.154701	2.0	3.0	4.0	4.0	4.0	3.0	0.666667

In [19]:
```
grouped['num_legs'].agg(['sum', 'mean', 'std'])
```

Out[19]:

class	sum	mean	std
bird	2	2.000000	NaN
mammal	10	3.333333	1.154701

In [22]:
```
grouped.agg({'num_legs': 'sum', 'animal': lambda x: ", ".join(x)})
```

Out[22]:

class	num_legs	animal
bird	2	penguin
mammal	10	cat, dog, bat

Pivotar una agregación

Pivotar es una forma muy poderosa de transformar las agrupaciones de los DataFrames. La función `pivot_table` nos permite agrupar, ordenar, calcular y manejar datos de una forma muy similar a la que se hace con las hojas de cálculo.

```
DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True)
```

Con los argumentos de `index` y `columns` puede especificar qué variables usar para etiquetar las filas y columnas. Durante la operación, se pueden agregar los valores utilizando la función estadística indicada en el parámetro `aggfunc`, que por defecto, calculará la media de todas aquellas columnas que sean de tipo numérico. Los niveles en la tabla resultante se almacenarán en objetos MultiIndex (índices jerárquicos) sobre el índice y las columnas del DataFrame resultante.

In [26]:
```
# df = df.reset_index().rename({'index': 'animal'}, axis='columns')
df
```

Out[26]:

	animal	num_legs	num_wings	class	locomotion
0	cat	4	0	mammal	walks
1	dog	4	0	mammal	walks
2	bat	2	2	mammal	flies
3	penguin	2	2	bird	walks

In [27]:
```
# equivalente a df.groupby('class').mean()
df.pivot_table(index='class')
```

Out[27]:

class	num_legs	num_wings
bird	2.000000	2.000000
mammal	3.333333	0.666667

In [30]:
```
# equivalente a df.groupby('class').num_legs.mean()
df.pivot_table(index='class', values='num_legs')
```

Out[30]:

class	num_legs
bird	2.000000
mammal	3.333333

In [54]:
```
import numpy as np
# equivalente a df.groupby(['class','order']).num_legs.agg(['sum', 'mean'])
df.pivot_table(index=['class', 'animal'], values=['num_legs', 'num_wings'], aggfunc=[np.sum])
```

Out[54]:

		sum	
		num_legs	num_wings
class	animal		
bird	penguin	2	2
	bat	2	2
mammal	cat	4	0
	dog	4	0

In [47]:
```
df.pivot_table(index=['class'], columns=['num_legs', 'locomotion'], values='animal', aggfunc=lambda x:' ,'.join(x), fill_value='-')
```

Out[47]:

num_legs	2	4	
locomotion	flies	walks	walks
class			
bird	-	penguin	-
mammal	bat	-	cat ,dog

También existe la función `pivot` similar a la anterior que permite cambiar la forma de los datos (producir una tabla "pivote") basada en los valores de las columnas. Utiliza valores únicos de índices/columnas especificados para formar ejes del DataFrame resultante. Esta función no admite la agregación de datos, los valores múltiples darán como resultado un MultiIndex en las columnas.

```
DataFrame.pivot(self, index=None, columns=None, values=None)
```

Series temporales

Las series de datos temporales son una forma importante de datos estructurados en muchos campos diferentes, como finanzas, economía, ecología, neurociencia y física. Todo lo que se observa o mide en diferentes puntos en el tiempo forma una serie temporal.

La biblioteca estándar de Python incluye tipos de datos para datos de fecha y hora, así como funciones relacionadas con el calendario. Los módulos `date` , `time` y `calendar` son sus elementos principales. El tipo `datetime.datetime` , o simplemente `datetime` , se usa ampliamente:

```
In [130]: from datetime import datetime
          now = datetime.now()
          now
Out[130]: datetime.datetime(2020, 2, 26, 13, 12, 18, 434783)

In [131]: now.year, now.month, now.day
Out[131]: (2020, 2, 26)
```

`datetime` almacena tanto la fecha como la hora en microsegundos, `timedelta` representa la diferencia temporal entre dos objetos de fecha y hora:

```
In [132]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)
          delta
Out[132]: datetime.timedelta(days=926, seconds=56700)

In [133]: delta.days, delta.seconds
Out[133]: (926, 56700)
```

Como en otros lenguajes existen difernetes funciones que permiten la conversión entre cadenas de texto y objetos fecha-hora y viceversa como `datetime.strftime` o `datetime.strptime` .

Indexación y Selección

Las series temporales se comportan como cualquier `Serie` o `DataFrame` de `pandas` cuando se está indexando y seleccionando datos en función de las etiquetas:

In [134]:
```
AAPL = pd.read_csv('./data/Yahoo/AAPL.csv', index_col='Date',
parse_dates=['Date'])
AAPL.head()
```

Out[134]:

Date	High	Low	Open	Close	Volume	Adj Close
2017-01-03	116.330002	114.760002	115.800003	116.150002	28781900.0	111.70983
2017-01-04	116.510002	115.750000	115.849998	116.019997	21118100.0	111.58477
2017-01-05	116.860001	115.809998	115.919998	116.610001	22193600.0	112.15222
2017-01-06	118.160004	116.470001	116.779999	117.910004	31751900.0	113.40254
2017-01-09	119.430000	117.940002	117.949997	118.989998	33561900.0	114.44124

In [135]:
```
price = AAPL['Adj Close']
price.head()
```

Out[135]:
```
Date
2017-01-03    111.709831
2017-01-04    111.584778
2017-01-05    112.152229
2017-01-06    113.402542
2017-01-09    114.441246
Name: Adj Close, dtype: float64
```

In [136]:
```
price['2017-01-04']  # equivalente a price['20170104']
```

Out[136]:
```
111.58477783203124
```

In [137]:
```
AAPL.loc['2017-01-04']  # equivalente a AAPL.loc['20170104']
```

Out[137]:
```
High          1.165100e+02
Low           1.157500e+02
Open          1.158500e+02
Close         1.160200e+02
Volume        2.111810e+07
Adj Close     1.115848e+02
Name: 2017-01-04 00:00:00, dtype: float64
```

Se puede pasar un año o solo un año y un mes para seleccionar fácilmente segmentos de datos:

In [138]:
```
price['2017-01']
```

Out[138]:
```
Date
2017-01-03    111.709831
2017-01-04    111.584778
2017-01-05    112.152229
2017-01-06    113.402542
2017-01-09    114.441246
                  ...
2017-01-25    117.220779
2017-01-26    117.278488
2017-01-27    117.288101
2017-01-30    116.980324
2017-01-31    116.711029
Name: Adj Close, Length: 20, dtype: float64
```

In [139]:
```
AAPL.loc['2017-01']
```

Out[139]:

Date	High	Low	Open	Close	Volume	Adj Close
2017-01-03	116.330002	114.760002	115.800003	116.150002	28781900.0	111.70983
2017-01-04	116.510002	115.750000	115.849998	116.019997	21118100.0	111.58477
2017-01-05	116.860001	115.809998	115.919998	116.610001	22193600.0	112.15222
2017-01-06	118.160004	116.470001	116.779999	117.910004	31751900.0	113.40254
2017-01-09	119.430000	117.940002	117.949997	118.989998	33561900.0	114.44124
...	...	...	...	...	...	.
2017-01-25	122.099998	120.279999	120.419998	121.879997	32377600.0	117.22077
2017-01-26	122.440002	121.599998	121.669998	121.940002	26337600.0	117.27848
2017-01-27	122.349998	121.599998	122.139999	121.949997	20562900.0	117.28810
2017-01-30	121.629997	120.660004	120.930000	121.629997	30377500.0	116.98032
2017-01-31	121.389999	120.620003	121.150002	121.349998	49201000.0	116.71102

20 rows × 6 columns

In [140]:
```
price[:'2017-01-06']
```

Out[140]:
```
Date
2017-01-03    111.709831
2017-01-04    111.584778
2017-01-05    112.152229
2017-01-06    113.402542
Name: Adj Close, dtype: float64
```

In [141]:
```
AAPL[:'2017-01-06']
```

Out[141]:

Date	High	Low	Open	Close	Volume	Adj Close
2017-01-03	116.330002	114.760002	115.800003	116.150002	28781900.0	111.70983
2017-01-04	116.510002	115.750000	115.849998	116.019997	21118100.0	111.58477
2017-01-05	116.860001	115.809998	115.919998	116.610001	22193600.0	112.15222
2017-01-06	118.160004	116.470001	116.779999	117.910004	31751900.0	113.40254

Debido a que la mayoría de las series temporales se ordenan cronológicamente, las consulta de rango pueden utilizar sellos de tiempo no contenidos en la serie:

In [142]:
```
# equivalente a price.truncate(before='2017-01-01', after='2017-01-07')
price['2017-01-01':'2017-01-07']
```

Out[142]:
```
Date
2017-01-03    111.709831
2017-01-04    111.584778
2017-01-05    112.152229
2017-01-06    113.402542
Name: Adj Close, dtype: float64
```

In [143]:
```
# equivalente a AAPL.truncate(before='2017-01-01', after='2017-01-07')
AAPL['2017-01-01':'2017-01-07']
```

Out[143]:

Date	High	Low	Open	Close	Volume	Adj Close
2017-01-03	116.330002	114.760002	115.800003	116.150002	28781900.0	111.70983
2017-01-04	116.510002	115.750000	115.849998	116.019997	21118100.0	111.58477
2017-01-05	116.860001	115.809998	115.919998	116.610001	22193600.0	112.15222
2017-01-06	118.160004	116.470001	116.779999	117.910004	31751900.0	113.40254

Rangos de fechas, frecuencias y desplazamiento

Se supone que las series temporales genéricas en pandas son irregulares, es decir, no tienen una frecuencia fija. Para muchas aplicaciones esto es suficiente. Sin embargo, a menudo es conveniente trabajar en relación con una frecuencia fija, como diaria, mensual o cada 15 minutos, incluso si eso significa introducir valores perdidos en una serie de tiempo. Afortunadamente, pandas tienen un conjunto completo de frecuencias de serie estándar y herramientas para remuestrear, inferir frecuencias y generar rangos de fechas de frecuencia fija.

Generando rangos de fechas

`pandas.date_range` es responsable de generar un `DatetimeIndex` con una longitud indicada de acuerdo a una frecuencia particular:

```
In [144]: index = pd.date_range('2019-01-01', '2019-01-10')
          index
```

```
Out[144]: DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '20
          19-01-04',
                           '2019-01-05', '2019-01-06', '2019-01-07', '20
          19-01-08',
                           '2019-01-09', '2019-01-10'],
                          dtype='datetime64[ns]', freq='D')
```

Por defecto, `date_range` genera marcas de tiempo diarias. Si solo pasa una fecha de inicio o finalización, se debe pasar una cantidad de períodos para generar. También se puede indicar la frecuencia a utilizar entre cada entrada del índice.

```
In [145]: pd.date_range(start='2019-01-01', periods=10)
```

```
Out[145]: DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '20
          19-01-04',
                           '2019-01-05', '2019-01-06', '2019-01-07', '20
          19-01-08',
                           '2019-01-09', '2019-01-10'],
                          dtype='datetime64[ns]', freq='D')
```

```
In [146]: pd.date_range(start='2019-01-01', periods=10, freq='B')
```

```
Out[146]: DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '20
          19-01-04',
                           '2019-01-07', '2019-01-08', '2019-01-09', '20
          19-01-10',
                           '2019-01-11', '2019-01-14'],
                          dtype='datetime64[ns]', freq='B')
```

In [147]:
```
pd.date_range(start='2019-01-01 08:00:00', periods=9, freq='2H')
```

Out[147]:
```
DatetimeIndex(['2019-01-01 08:00:00', '2019-01-01 10:00:00',
               '2019-01-01 12:00:00', '2019-01-01 14:00:00',
               '2019-01-01 16:00:00', '2019-01-01 18:00:00',
               '2019-01-01 20:00:00', '2019-01-01 22:00:00',
               '2019-01-02 00:00:00'],
              dtype='datetime64[ns]', freq='2H')
```

In [148]:
```
pd.date_range(start='2019-01', periods=8, freq='WOM-3FRI')
```

Out[148]:
```
DatetimeIndex(['2019-01-18', '2019-02-15', '2019-03-15', '2019-04-19',
               '2019-05-17', '2019-06-21', '2019-07-19', '2019-08-16'],
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

Desplazamiento de datos (avance y retroceso)

"Desplazar" se refiere a mover datos hacia atrás y hacia adelante a través del tiempo. Tanto Series como DataFrame tienen un método `shift` de desplazamiento para ir hacia adelante o hacia atrás por los datos, dejando el índice sin modificar:

In [149]:
```
price['2017-01-03':'2017-01-07'].shift(3, freq='D')
```

Out[149]:
```
Date
2017-01-06    111.709831
2017-01-07    111.584778
2017-01-08    112.152229
2017-01-09    113.402542
Freq: D, Name: Adj Close, dtype: float64
```

In [150]:
```
price['2017-01-03':'2017-01-07'].shift(-4, freq='D')
```

Out[150]:
```
Date
2016-12-30    111.709831
2016-12-31    111.584778
2017-01-01    112.152229
2017-01-02    113.402542
Freq: D, Name: Adj Close, dtype: float64
```

Remuestreo y conversión de frecuencia

El remuestreo se refiere al proceso de convertir una serie de tiempo de una frecuencia a otra. La agregación de datos de frecuencias más altas a frecuencias más bajas se denomina *downsampling*, mientras que la conversión de frecuencias más bajas a frecuencias más altas se llama *upsampling*.

Los objetos pandas están equipados con el método `resample`, que es la función central para todas las conversiones de frecuencia. Tiene una API similar a la de `groupby`, se llama a `resample` para agrupar los datos y aplicar una función de agregación.

Downsampling

Cuando se pasa de una frecuencia más alta a una más baja es necesario aplicar un proceso de agregación para no perder información.

```
In [151]: price['2017'].resample('M').mean()
```

```
Out[151]: Date
          2017-01-31    114.999078
          2017-02-28    128.988916
          2017-03-31    135.828642
          2017-04-30    138.020382
          2017-05-31    147.433160
                           ...
          2017-08-31    154.663035
          2017-09-30    153.469347
          2017-10-31    153.672636
          2017-11-30    168.280443
          2017-12-31    167.978944
          Freq: M, Name: Adj Close, Length: 12, dtype: float64
```

```
In [152]: price['2017'].resample('M', kind='period').mean()
```

```
Out[152]: Date
          2017-01    114.999078
          2017-02    128.988916
          2017-03    135.828642
          2017-04    138.020382
          2017-05    147.433160
                        ...
          2017-08    154.663035
          2017-09    153.469347
          2017-10    153.672636
          2017-11    168.280443
          2017-12    167.978944
          Freq: M, Name: Adj Close, Length: 12, dtype: float64
```

In [153]:
```
price['2017'].resample('M').ohlc()
```

Out[153]:

Date	open	high	low	close
2017-01-31	111.709831	117.288101	111.584778	116.711029
2017-02-28	123.828148	132.440292	123.616562	132.324371
2017-03-31	135.028976	139.211563	133.956802	138.767197
2017-04-30	138.805832	139.839371	135.888687	138.757538
2017-05-31	141.587769	151.405930	141.539459	148.166336
...	...	...	...	...
2017-08-31	145.537842	159.692978	145.537842	159.692978
2017-09-30	159.741684	159.741684	146.596222	150.072464
2017-10-31	149.770599	164.600632	149.449265	164.600632
2017-11-30	162.507095	171.611557	162.507095	167.938385
2017-12-31	167.156601	172.404373	165.163025	165.378021

12 rows × 4 columns

Upsampling

Cuando se connvierte de una frecuencia baja a otra mayor no es necesario aplicar una agregación.

In [154]:
```
price['2017-01-03':'2017-01-05']
```

Out[154]:
```
Date
2017-01-03    111.709831
2017-01-04    111.584778
2017-01-05    112.152229
Name: Adj Close, dtype: float64
```

In [155]:
```
price12 = price['2017-01-03':'2017-01-05'].resample('12H').asfreq()
price12
```

Out[155]:
```
Date
2017-01-03 00:00:00    111.709831
2017-01-03 12:00:00           NaN
2017-01-04 00:00:00    111.584778
2017-01-04 12:00:00           NaN
2017-01-05 00:00:00    112.152229
Freq: 12H, Name: Adj Close, dtype: float64
```

In [156]:
```
price['2017-01-03':'2017-01-05'].resample('12H').ffill()
```

Out[156]:
```
Date
2017-01-03 00:00:00    111.709831
2017-01-03 12:00:00    111.709831
2017-01-04 00:00:00    111.584778
2017-01-04 12:00:00    111.584778
2017-01-05 00:00:00    112.152229
Freq: 12H, Name: Adj Close, dtype: float64
```

Gráficas directas

Pandas proporciona opciones integradas para realizar visualizaciones de los datos que maneja, tanto a nivel de serie como de DataFrame:

In [3]:
```
dataset = pd.read_csv("./data/teoria/salaries.csv")
rank = dataset['rank']
discipline = dataset['discipline']
phd = dataset['phd']
service = dataset['service']
sex = dataset['sex']
salary = dataset['salary']
dataset.head()
```

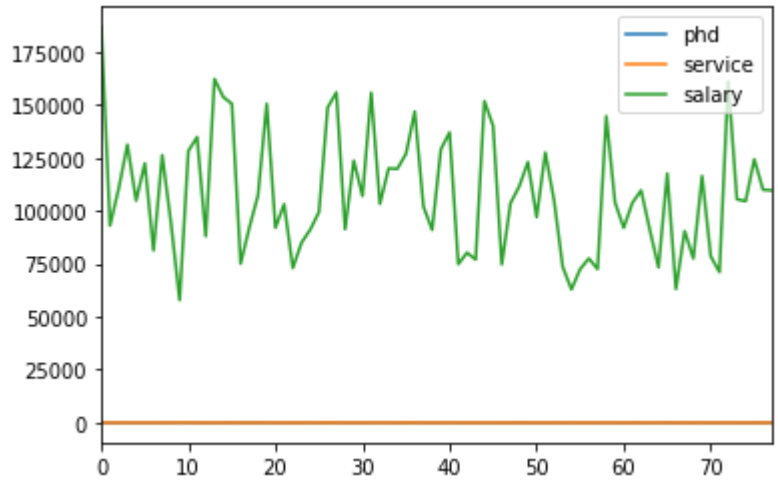
Out[3]:

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Líneas

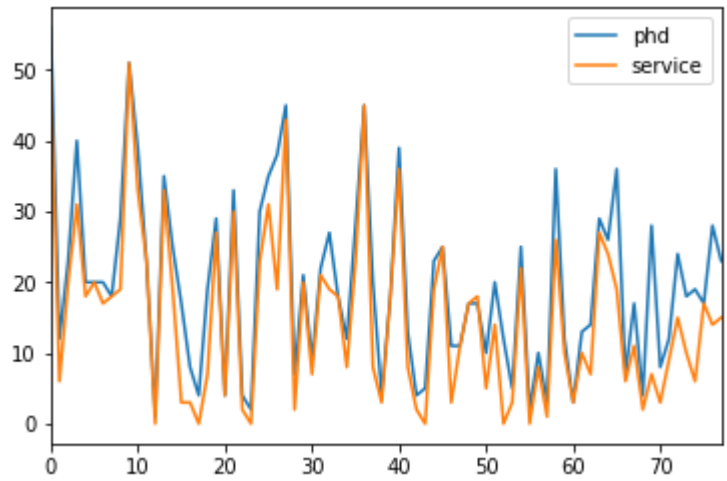
Muestran la evolución de un conjunto de variables.

```
In [5]: dataset[["rank", "discipline","phd","service", "sex", "salary"]].plot();
```



Es importante tener en cuenta la naturaleza de la información que queremos visualizar y en caso de valores numéricos la maginitud de las unidades utilizadas.

```
In [159]: dataset[["phd","service"]].plot();
```



Barras

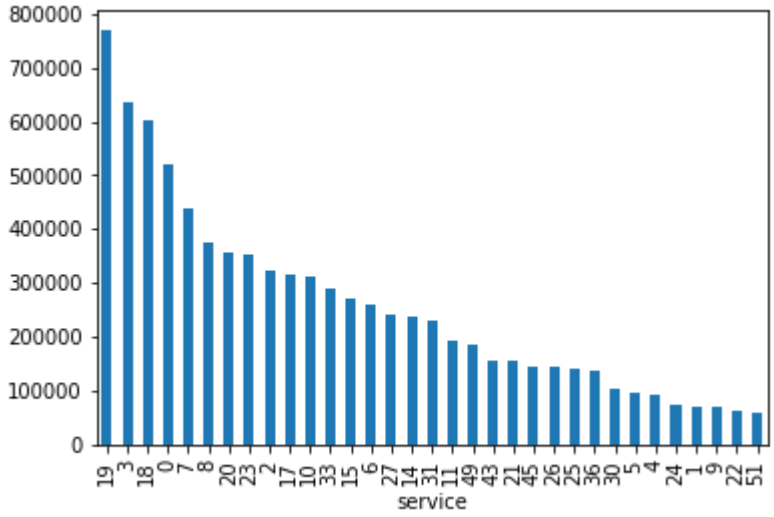
Los gráficos de barras se utilizan para comparar diferentes grupos o para rastrear cambios a lo largo del tiempo. Sin embargo, cuando se trata de medir un cambio en el tiempo, los gráficos de barras son mejores cuando los cambios son más grandes.

In [160]:
```
dataset1 = dataset.groupby(['service']).sum()
dataset1.sort_values("salary", ascending = False, inplace=True)
dataset1.head()
```

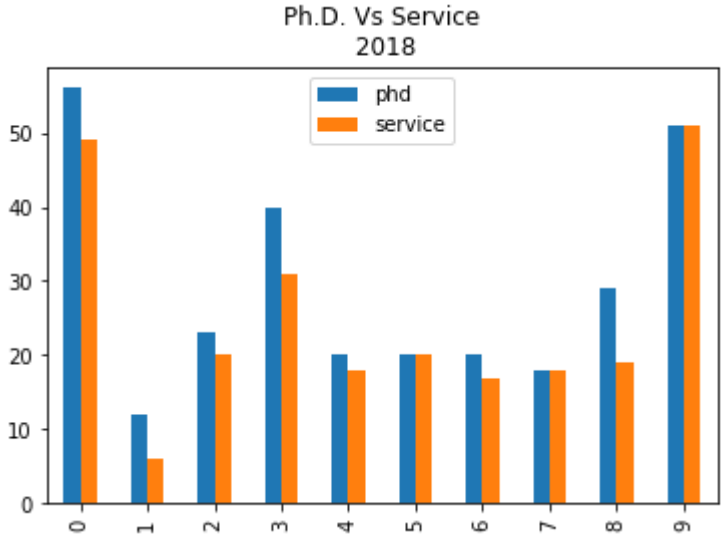
Out[160]:

service	phd	salary
19	178	769448
3	56	635216
18	91	603060
0	26	519500
7	70	440408

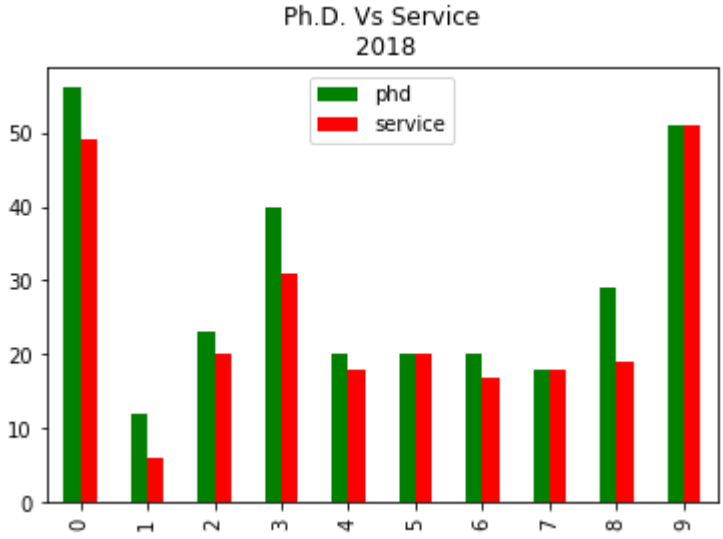
In [161]:
```
dataset1["salary"].plot.bar();
```



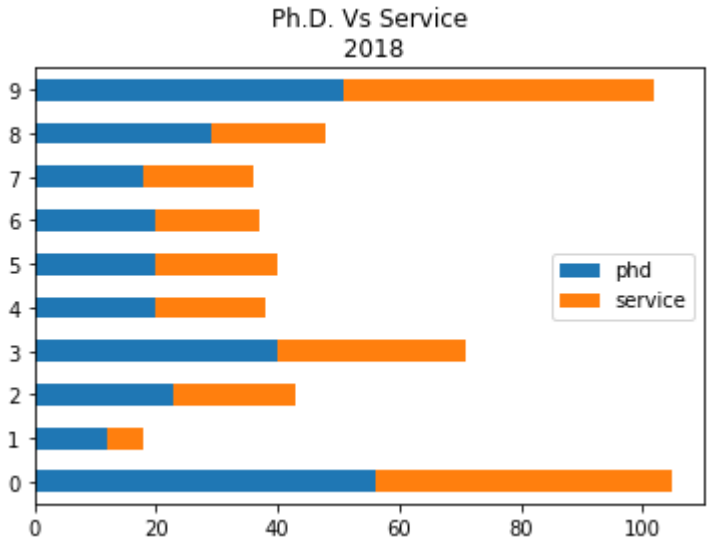
In [162]:
```
dataset[[ 'phd', 'service' ]].head(10).plot.bar(title="Ph.D. Vs Service\n 2018");
```



In [163]:
```
dataset[['phd', 'service']].head(10).plot.bar(title="Ph.D. Vs Service\n 2018" , color=['g','r']);
```



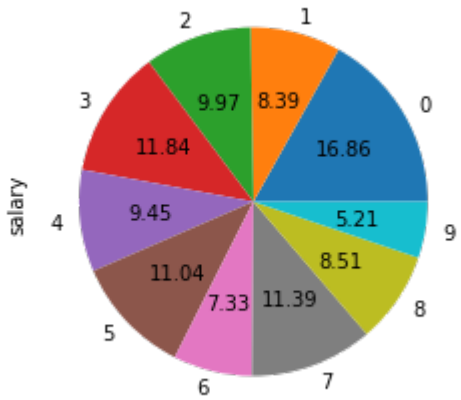
In [164]:
```
dataset[['phd', 'service']].head(10).plot.barh(title="Ph.D. Vs Service\n 2018", stacked=True);
```



Tartas

Los gráficos circulares o de tarta son útiles para comparar partes de un todo. No muestran cambios en el tiempo. El gráfico circular es útil para comparar un número de variables pequeño.

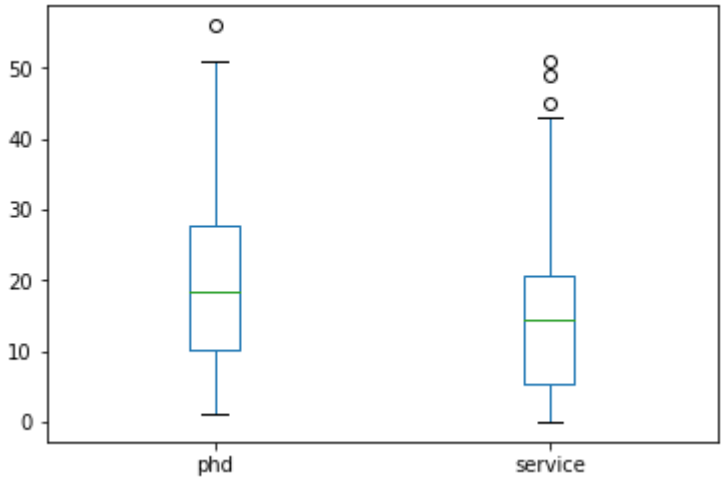
```
In [165]: dataset["salary"].head(10).plot.pie(autopct='%.2f');
```



Cajas

La gráfica de caja se usa para comparar variables usando algunos valores estadísticos (media y cuantiles generalmente). Las variables comparables deben ser de las mismas unidades de datos.

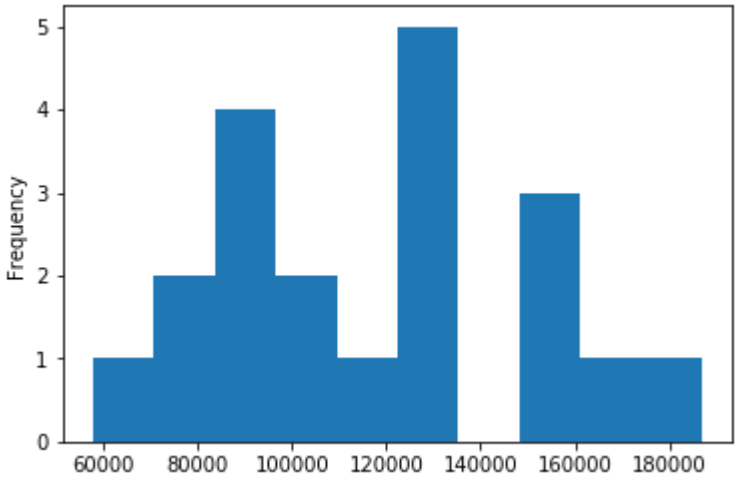
```
In [166]: dataset[["phd","service"]].plot.box();
```



Histiogramas

Se puede usar un histograma para representar una variable específica o un conjunto de variables.

In [167]:
```
dataset["salary"].head(20).plot.hist();
```



Gráficos de dispersión

Un diagrama de dispersión muestra la relación entre dos factores de un experimento. Se utiliza una línea de tendencia para determinar la correlación positiva, negativa o nula.

In [168]:
```
dataset.plot(kind='scatter', x='phd', y='service', title='Pop
uation vs area and density\n 2018', s=0.9);
```