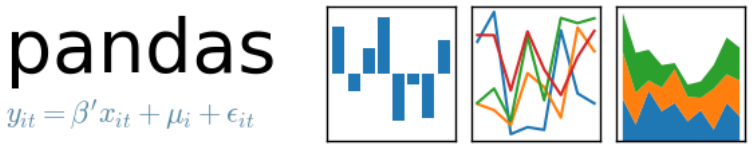

Pandas

Pandas

- *pandas* is a high-performance, open source library for data analysis in Python
  - Developed by Wes McKinney in 2008
- Key features of pandas:
  - It can process a variety of data sets in different formats: time series, tabular heterogeneous, and matrix data
  - It facilitates loading/importing data from varied sources such as CSV and DB/SQL
  - It can handle a myriad of operations on data sets: subsetting, slicing, filtering, merging, groupBy, re-ordering, and re-shaping
  - It can be used for parsing and munging (conversion) of data as well as modeling and statistical analysis
  - It integrates well with other Python libraries such as SciPy, and scikit-learn
  - It delivers fast performance



Importing pandas

- Typically the pandas library is imported under the alias pd:

```
import pandas as pd
```

- Typically, pandas will be imported together with NumPy:

```
import numpy as np
import pandas as pd
```

Pandas Objects

- NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks
- Its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.)
- Pandas objects build on the NumPy array structure and provide efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time
- There are three main data structures in pandas:
  - Series
  - DataFrame
  - Panel

Series

- A Series is a one-dimensional array of indexed data (of any NumPy data type)
  - You can think of it as a fixed-length, ordered dict, mapping index values to data values
- The syntax for creating a Series data structure is:

```
pd.Series(data, index=idx)
```

key -> row (with one element only)

- The simplest Series is formed from only an array of data:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

  - The Series wraps both a sequence of values and a sequence of indices
  - If an index is not specified, a default index [0,... n-1], is created

Series Attributes

- We can access the sequence of values and the index object of the Series via its **values** and **index** attributes, respectively:

```
data.values
```

```
array([0.25, 0.5 , 0.75, 1.  ])
```

```
data.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

  - The values are simply a familiar NumPy array
  - The index is an array-like object of type pd.Index

try it
how to see values of the index?

Indexing and Slicing

- Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
data[1]
```

```
0.5
```

```
data[1:3]
```

```
1    0.50
2    0.75
dtype: float64
```

```
data[data > 0.5]
```

```
2    0.75
3    1.00
dtype: float64
```

U: return values 0.5 and 0.75

Series as a Generalized NumPy Array

- The essential difference between a NumPy array and a Series object is the presence of the index
- While the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values
- This explicit index definition gives the Series object additional capabilities
- The index need not be an integer, but can consist of values of any desired type
- For example, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

'a b c'.split

Series as a Generalized NumPy Array

- And the item access works as expected:

```
data['b']
```

```
0.5
```

```
data['c'] = 0.8
data
```

```
a    0.25
b    0.50
c    0.80
d    1.00
dtype: float64
```

- We can even use non-contiguous or non-sequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Series as a Generalized NumPy Array

- And the item access works as expected:

```
data['b']
```

```
0.5
```

```
data['c'] = 0.8
data
```

```
a    0.25
b    0.50
c    0.80
d    1.00
dtype: float64
```

- We can even use non-contiguous or non-sequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

data[2]=?
how to get the third row? iloc

Series as a Generalized NumPy Array

- A Series's index can be altered in place by assignment:

```
data.index = [1, 3, 5, 10]
data
```

```
1     0.25
3     0.50
5     0.75
10    1.00
dtype: float64
```

~~Series as a Generalized NumPy Array~~

Experis Academy
ManpowerGroup

- A Series's index can be altered in place by assignment:

```
data.index = [1, 3, 5, 10]
data
```

```
1     0.25
3     0.50
5     0.75
10    1.00
dtype: float64
```

Series as a Specialized Dictionary

- You can also think of a Pandas Series as a specialization of a Python dictionary
- A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, while a Series is a structure which maps *typed* keys to a set of *typed* values
  - The type information makes Pandas Series more efficient than Python dictionaries for certain operations (just as NumPy arrays are more efficient than Python lists for certain operations)
- A Series object can be constructed directly from a Python dictionary:

```
population_dict = {'California': 39776830,
                   'Texas': 28704330,
                   'Florida': 21312211,
                   'New York': 19862512,
                   'Pennsylvania': 12823989}
population = pd.Series(population_dict)
population
```

```
California      39776830
Florida         21312211
New York        19862512
Pennsylvania    12823989
Texas           28704330
dtype: int64
```

U - create with pop in millions.xx

Series as a Specialized Dictionary

- By default, a Series will be created where the index is drawn from the sorted keys
- From here, typical dictionary-style item access can be performed:

```
population['California']
```

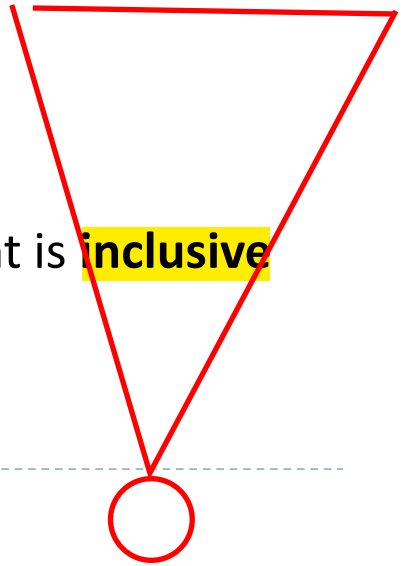
```
39776830
```

- Unlike a dictionary, the Series also supports array-style operations such as slicing:

```
population['California': 'New York']
```

```
California    39776830
Florida       21312211
New York      19862512
dtype: int64
```

  - Slicing with labels behaves differently than normal Python slicing in that the endpoint is **inclusive**



~~Series as a Specialized Dictionary~~

- By default, a Series will be created where the index is drawn from the sorted keys
- From here, typical dictionary-style item access can be performed:

```
population['California']
```

```
39776830
```

- Unlike a dictionary, the Series also supports array-style operations such as slicing:

```
population['California': 'New York']
```

```
California    39776830
Florida       21312211
New York      19862512
dtype: int64
```

  - Slicing with labels behaves differently than normal Python slicing in that the endpoint is **inclusive**

Creating a Series from a Dictionary

- The index can be explicitly set if a different result is preferred:

```
population = pd.Series(population_dict,
                       index=['California', 'New York', 'Florida'])
population
```

```
California    39776830
New York      19862512
Florida       21312211
dtype: int64
```

- In this case, the Series is populated only with the explicitly identified keys
- Just as in the case of dict, KeyError is raised if you try to retrieve a missing label:

```
try:
    population['Illinois']
except KeyError:
    print("Country's data is not available")
```

```
Country's data is not available
```

Exercise

- Create a Pandas series from each of the items below: a list, a NumPy array and a dictionary:

```
my_list = list('abcedfghijklmnopqrstuvwxyz')
my_arr = np.arange(26)
my_dict = dict(zip(my_list, my_arr))
```

Operations on Series

- Just like NumPy arrays, you can apply math functions on Series:

```
ser2 * 2
```

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

same broadcasting as numpy

```
np.exp(ser2)
```

```
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

```
np.mean(ser2)
```

```
2.75
```

```
np.std(ser2)
```

```
4.710360920354193
```

Detecting Missing Data

- The **isnull()** and **notnull()** functions in pandas can be used to detect missing data:

```
pd.isnull(ser4)
```

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
pd.notnull(ser4)
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

assign np.nan to Cal' and test it

- We'll discuss handling missing data later in the course

Alignment

- An important feature of Series is that the data is automatically differently-indexed data on the basis of the label:

pros and cons?

```
ser3
```

```
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
ser4
```

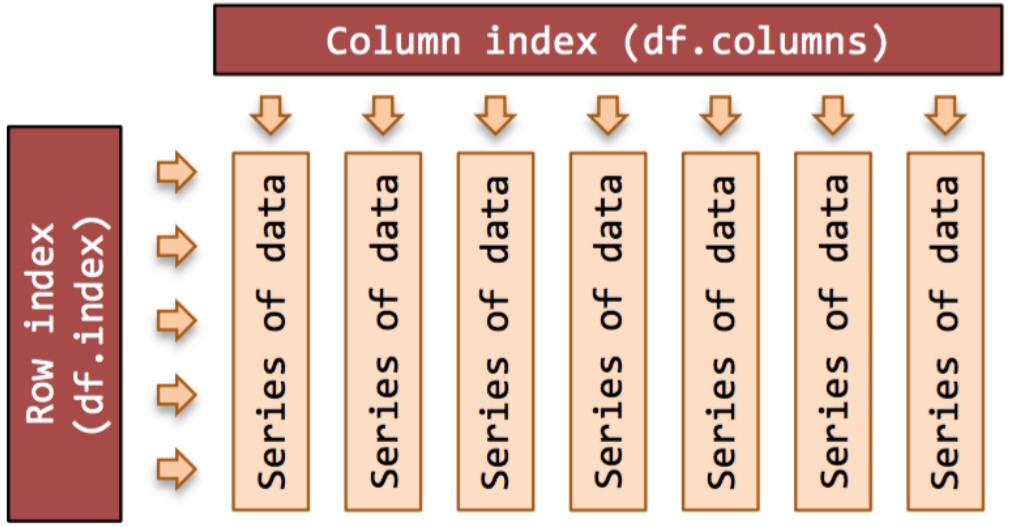
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
ser3 + ser4
```

```
California        NaN
Ohio          70000.0
Oregon        32000.0
Texas        142000.0
Utah              NaN
dtype: float64
```

DataFrame

- DataFrame is the most commonly used data structure in pandas
- A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different type (int, bool, etc.)
- DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names
- It can be thought of as a dictionary of aligned Series objects, sharing the same index
  - A DataFrame column is a Series structure
- It is similar to structured arrays in NumPy with
  - Columns can be inserted and deleted



DataFrame Creation

- There are numerous ways to construct a DataFrame
- The constructor accepts many different types of arguments:
  - Dictionary of 1D NumPy arrays, lists, dictionaries, or Series structures
  - 2D NumPy array
  - Structured NumPy array
  - Another DataFrame structure
  - and more
- Row label indexes and column labels can be specified along with the data
- If they're not specified, they will be generated from the input data in an intuitive way
  - For example, from the keys of dict (for column labels) or by using range(n) for row labels, where n corresponds to the number of rows

backstage

Using a Dictionary of Series

- Let's first construct two Series listing the population and area of the five most populated US states:

```
population_dict = {'California': 39776830,
                   'Texas': 28704330,
                   'Florida': 21312211,
                   'New York': 19862512,
                   'Pennsylvania': 12823989}
population = pd.Series(population_dict)
```

```
area_dict = {'California': 423967,
             'Texas': 695662,
             'Florida': 170312,
             'New York': 141297,
             'Pennsylvania': 119282}
area = pd.Series(area_dict)
```

Using a Dictionary of Series

- Now we can use a dictionary of these two series to construct a DataFrame object:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

	area	population
California	423967	39776830
Florida	170312	21312211
New York	141297	19862512
Pennsylvania	119282	12823989
Texas	695662	28704330

try now with removing Texas from area series

DataFrame Attributes

- The row index labels and column labels can be accessed via the **index** and **columns** attributes:

```
states.index
```

```
Index(['California', 'Florida', 'New York', 'Pennsylvania',
'Texas'], dtype='object')
```

```
states.columns
```

```
Index(['area', 'population'], dtype='object')
```

- The **values** attribute returns the data contained in the DataFrame as a 2D ndarray

```
states.values
```

```
array([[  423967, 39776830],
       [  170312, 21312211],
       [  141297, 19862512],
       [  119282, 12823989],
       [  695662, 28704330]], dtype=int64)
```

DataFrame As Specialized Dictionary

- Similarly, we can also think of a DataFrame as a specialization of a dictionary
- Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data
- For example, asking for the 'area' attribute returns the areas Series object:

```
states['area']
```

```
California      423967
Florida         170312
New York        141297
Pennsylvania    119282
Texas           695662
Name: area, dtype: int64
```

verify this type is Series

  - Notice the potential confusion here: in a 2D NumPy array, data[0] will return the first *row*, while in a DataFrame, data['col0'] will return the first *column*

DataFrame from a Single Series Object

- A single-column DataFrame can be constructed from a single Series:

```
pd.DataFrame(population, columns=['population'])
```

and if I will not specify columns?

	population
California	39776830
Florida	21312211
New York	19862512
Pennsylvania	12823989
Texas	28704330

DataFrame from a Dictionary of Lists

- One of the most common ways to create a DataFrame is from a dictionary of equal-length lists or NumPy arrays:

```
data = {'population': [39776830, 28704330, 21312211, 19862512, 12823989],
        'area': [423967, 695662, 170312, 141297, 119282]}
index = ['California', 'Texas', 'Florida', 'New York', 'Pennsylvania']
states = pd.DataFrame(data, index=index)
states
```

	area	population
California	423967	39776830
Texas	695662	28704330
Florida	170312	21312211
New York	141297	19862512
Pennsylvania	119282	12823989

- Note that the columns are placed in sorted order

try to change the names after the df creation
think of it as a property of an object

df = df[['pop2', 'pop']]

DataFrame from a Dictionary of Lists

- You can specify the sequence of columns, and they will appear in the specified order:

```
data = {'population': [39776830, 28704330, 21312211, 19862512, 12823989],
        'area': [423967, 695662, 170312, 141297, 119282]}
index = ['California', 'Texas', 'Florida', 'New York', 'Pennsylvania']
states = pd.DataFrame(data, index=index,
                      columns=['population', 'area'])
states
```

	population	area
California	39776830	423967
Texas	28704330	695662
Florida	21312211	170312
New York	19862512	141297
Pennsylvania	12823989	119282

DataFrame from a Dictionary of Dictionaries

- DataFrames can also be created by using a nested dictionary of dictionaries
  - The outer dictionary keys are interpreted as the columns
  - The keys in the inner dictionaries are unioned and sorted to form the row indices

```
data = {'population': {'California': 39776830,
                       'Texas': 28704330,
                       'Florida': 21312211,
                       'New York': 19862512,
                       'Pennsylvania': 12823989},
        'area': {'California': 423967,
                 'Texas': 695662,
                 'Florida': 170312,
                 'New York': 141297,
                 'Pennsylvania': 119282}
       }
states = pd.DataFrame(data)
states
```

	area	population
California	423967	39776830
Florida	170312	21312211
New York	141297	19862512
Pennsylvania	119282	12823989
Texas	695662	28704330

DataFrame from a Dictionary of Dictionaries

- If some keys in one of the inner dictionaries are missing, Pandas will fill them in with NaN values:

```
data = {'population': {'California': 39776830,
                       'Texas': 28704330,
                       'Florida': 21312211,
                       'New York': 19862512},
        'area': {'California': 423967,
                 'Texas': 695662,
                 'New York': 141297,
                 'Pennsylvania': 119282}
       }
states = pd.DataFrame(data)
states
```

	area	population
California	423967.0	39776830.0
Florida	NaN	21312211.0
New York	141297.0	19862512.0
Pennsylvania	119282.0	NaN
Texas	695662.0	28704330.0

DataFrame from a List of Dictionaries

- Any list of dictionaries can be made into a DataFrame
- The dictionary keys are interpreted as the columns

```
data = [population_dict, area_dict]
states = pd.DataFrame(data, index=['population', 'area'])
states
```

	California	Florida	New York	Pennsylvania	Texas
population	39776830	21312211	19862512	12823989	28704330
area	423967	170312	141297	119282	695662

- Of course you can always transpose the result:

```
states.T
```

	population	area
California	39776830	423967
Florida	21312211	170312
New York	19862512	141297
Pennsylvania	12823989	119282
Texas	28704330	695662

DataFrame from a 2D NumPy array

- Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['Column A', 'Column B'],
             index=['a', 'b', 'c'])
```

	Column A	Column B
a	0.188171	0.090570
b	0.308763	0.254368
c	0.630323	0.761876

- If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 3))
```

	0	1	2
0	0.237744	0.876218	0.583536
1	0.059770	0.177468	0.564984
2	0.510760	0.789046	0.486915

DataFrame from a NumPy Structured Array

- A Pandas DataFrame operates much like a structured array, and can be created directly from one:

```
dtype = [('name', 'U10'), ('height', float), ('age', int)]
values = [('Arthur', 1.83, 41), ('Lancelot', 1.91, 38), ('Galahad', 1.75, 38)]
a = np.array(values, dtype=dtype)
pd.DataFrame(a)
```

	name	height	age
0	Arthur	1.83	41
1	Lancelot	1.91	38
2	Galahad	1.75	38

DataFrame Construction Summary

Experis Academy
ManpowerGroup

- The following table shows the possible data inputs to DataFrame constructor:

Type	Description
dict of Series	Each Series becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels.
2D ndarray	A matrix of data, passing optional row and column labels
NumPy structured array	Treated as the "dict of arrays" case
list of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed

Exercise

- Create the following DataFrame in at least 3 different ways:

	AMZN	FB	GOOG
Closing price	2039.51	171.16	1197.00
EPS	12.63	6.46	23.16
Shares Outstanding(M)	487.74	2398.61	348.95
Beta	1.61	0.66	1.40
P/E	157.98	25.87	51.38
Market Cap(B)	972.96	482.68	829.37

Data Indexing and Selection

Experis Academy
ManpowerGroup

- In NumPy we looked in detail at methods to access, set, and modify values in arrays
- These included:
  - Indexing (e.g., arr[2, 1])
  - Slicing (e.g., arr[:, 1:5])
  - Masking (e.g., arr[arr > 0])
  - Fancy indexing (e.g., arr[0, [1, 5]])
  - And combinations thereof (e.g., arr[:, [1, 5]])
- Pandas has similar means of accessing and modifying values in Pandas Series and DataFrame objects
- We'll start with the simple case of the one-dimensional Series object, and then move on to the more complicated two-dimensional DataFrame object

Selecting Columns

- A column in a DataFrame can be retrieved as a Series by a dictionary-like notation

```
pop_df['city']
```

```
0         NYC
1         NYC
2          LA
3          LA
4     Chicago
Name: city, dtype: object
```

- or by attribute:
  - this only works if the index element is a valid Python identifier

```
pop_df.year
```

```
0    2010
1    2015
2    2010
3    2015
4    2015
Name: year, dtype: int64
```

Selecting Columns

- We can pass a list of columns to the [] operator in order to select the columns in a particular order:

```
pop_df[['city', 'pop']]
```

	city	pop
0	NYC	8.19
1	NYC	8.51
2	LA	3.80
3	LA	3.95
4	Chicago	2.71

first row: df[0] or df[:1]

- Note that rows cannot be selected with the bracket operator [] in a DataFrame
  - This was a design decision made by the creators in order to avoid ambiguity

Adding Columns

- A new column can be added via assignment, as follows:

```
pop_df['state'] = ['NY', 'NY', 'CA', 'CA', 'IL']
pop_df['males%'] = 0.492
pop_df
```

	city	pop	year	state	males%
0	NYC	8.19	2010	NY	0.492
1	NYC	8.51	2015	NY	0.492
2	LA	3.80	2010	CA	0.492
3	LA	3.95	2015	CA	0.492
4	Chicago	2.71	2015	IL	0.492

Adding Columns

- A DataFrame structure can be treated as if it were a dictionary of Series objects
- To insert a column at a specific location, you can use the **insert()** method:

```
pop_df.insert(4, 'females%', 1 - pop_df['males%'])
pop_df
```

	city	pop	year	state	females%	males%
0	NYC	8.19	2010	NY	0.508	0.492
1	NYC	8.51	2015	NY	0.508	0.492
2	LA	3.80	2010	CA	0.508	0.492
3	LA	3.95	2015	CA	0.508	0.492
4	Chicago	2.71	2015	IL	0.508	0.492

Alignment

- DataFrame objects align in a manner similar to Series objects, except that they align on both column and index labels
- The resulting object is the union of the column and row labels:

```
ore1_df = pd.DataFrame(np.array([[20, 35, 25, 20],
                                 [11, 28, 32, 29]]),
                       columns=['iron', 'magnesium', 'copper', 'silver'])
ore1_df
```

	iron	magnesium	copper	silver
0	20	35	25	20
1	11	28	32	29

```
ore2_df = pd.DataFrame(np.array([[14, 34, 26, 26],
                                 [33, 19, 25, 23]]),
                       columns=['iron', 'magnesium', 'gold', 'silver'])
ore2_df
```

	iron	magnesium	gold	silver
0	14	34	26	26
1	33	19	25	23

```
ore1_df + ore2_df
```

	copper	gold	iron	magnesium	silver
0	NaN	NaN	34	69	46
1	NaN	NaN	44	47	52

- When there are no row labels or column labels in common, the value is filled with NaN

Alignment

- If you combine a DataFrame object and a Series object, the default behavior is to broadcast the Series object across the rows:

```
ore1_df + pd.Series([25, 25, 25, 25],
                    index=['iron', 'magnesium', 'copper', 'silver'])
```

	iron	magnesium	copper	silver
0	45	60	50	45
1	36	53	57	54

Function Application

- NumPy ufuncs (element-wise array methods) work fine with pandas objects:

```
np.sqrt(ore1_df)
```

	iron	magnesium	copper	silver
0	4.472136	5.916080	5.000000	4.472136
1	3.316625	5.291503	5.656854	5.385165

- DataFrame's **apply()** method allows you to apply a function on 1D arrays to each column or row:

```
f = lambda x: x.max() - x.min()
ore1_df.apply(f)
```

```
iron          9
magnesium     7
copper        7
silver        9
dtype: int64
```

```
ore1_df.apply(f, axis=1)
```

```
0    15
1    21
dtype: int64
```

Exercise

- Consider the following Python dictionary data and Python list labels:

```
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog', 'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- Create a DataFrame animals_df from this dictionary **data** which has the index **labels**
- Select just the 'animal' and 'age' columns from animals_df

Slicing

- DataFrame rows can be selected using **slicing**, similarly to NumPy arrays
- For example, to obtain the first row in the DataFrame:

try df[0]

```
pop_df[0:1]
```

	city	pop	year
0	NYC	8.19	2010

- To obtain all rows starting from index 2:

```
pop_df[2:]
```

	city	pop	year
2	LA	3.80	2010
3	LA	3.95	2015
4	Chicago	2.71	2015

Slicing

- Obtain rows at intervals of two, starting from row 0:

```
pop_df[::2]
```

	city	pop	year
0	NYC	8.19	2010
2	LA	3.80	2010
4	Chicago	2.71	2015

- Reverse the order of rows in DataFrame:

```
pop_df[::-1]
```

	city	pop	year
4	Chicago	2.71	2015
3	LA	3.95	2015
2	LA	3.80	2010
1	NYC	8.51	2015
0	NYC	8.19	2010

Integer-Based Indexing

- The **.iloc** attribute supports integer-based positional indexing
- It accepts the following as inputs:
  - A single integer, for example, 7
  - A list or array of integers, for example, [2, 3]
    - Each integer corresponds to a different axis
  - A slice object with integers, for example, 1:4
  - Any combination of the above, for example [5, 2:4] will select row no. 5 and column 2, 3
- For example, selecting row no. 2:

```
pop_df.iloc[2]
```

```
city       LA
pop       3.8
year     2010
Name: 2, dtype: object
```

  - Indexing using a single integer results in a Series object

Integer-Based Indexing

- Selecting cell (1, 1):

```
pop_df.iloc[1, 1]
```

```
8.51
```

- Selecting rows 1 and 3, and the first 2 columns:

```
pop_df.iloc[[1, 3], 0:2]
```

	city	pop
1	NYC	8.51
3	LA	3.95

  - Indexing using a list of integers results in a DataFrame object

Integer-Based Indexing

- The **.iat** attribute can be used for a quick selection of scalar values

```
pop_df.iloc[3, 1]
```

```
3.95
```

can select subset also

```
pop_df.iat[3, 1]
```

```
3.95
```

cant do that ^^^

```
%timeit pop_df.iloc[3, 1]
```

```
9.76 µs ± 637 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit pop_df.iat[3, 1]
```

```
6.06 µs ± 29.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

  - %timeit is an IPython magic function, which can be used to time a particular piece of code

Label Indexing

- The **.loc** attribute supports pure label-based indexing
- It accepts the following as valid inputs:
  - A single label such as ['March'] or ['Dubai']
  - List or array of labels, for example, ['Dubai', 'UK Brent']
  - A slice object with labels, for example, 'May':'Aug'
  - A boolean array
- For our illustrative dataset, we'll use the average snowy weather temperature data for New York city from the following sites:
  - http://www.currentresults.com/Weather/New-York/Places/new-york-city-snowfall-totals-snow-accumulation-averages.php
  - http://www.currentresults.com/Weather/New-York/Places/new-york-city-temperatures-by-month-average.php

Label Indexing

- Create the DataFrame:

```
nyc_snow_avgs_data = {'Months': ['January', 'February', 'March',
                                 'April', 'November', 'December'],
                      'Avg Snow Days': [4.0, 2.7, 1.7, 0.2, 0.2, 2.3],
                      'Avg Precip. (cm)': [17.8, 22.4, 9.1, 1.5, 0.8, 12.2],
                      'Avg Low Temp. (C)': [-3, -2, 2, 7, 5, 0] }
```

```
nyc_snow_avgs = pd.DataFrame(nyc_snow_avgs_data,
                             index = nyc_snow_avgs_data['Months'],
                             columns = ['Avg Snow Days','Avg Precip. (cm)',
                                        'Avg Low Temp. (C)'])
nyc_snow_avgs
```

	Avg Snow Days	Avg Precip. (cm)	Avg Low Temp. (C)
January	4.0	17.8	-3
February	2.7	22.4	-2
March	1.7	9.1	2
April	0.2	1.5	7
November	0.2	0.8	5
December	2.3	12.2	0

```
df = pd.DataFrame(
np.random.randint(100,200, (5,3)),
                  index=list('abcde'),
                  columns=list('xyz'))
```

Label Indexing

- Using a single label:

```
nyc_snow_avgs.loc['January']
```

```
Avg Snow Days          4.0
Avg Precip. (cm)      17.8
Avg Low Temp. (C)     -3.0
Name: January, dtype: float64
```

- Using a list of labels:

```
nyc_snow_avgs.loc[['January', 'April']]
```

	Avg Snow Days	Avg Precip. (cm)	Avg Low Temp. (C)
January	4.0	17.8	-3
April	0.2	1.5	7

Label Indexing

- Using a label range:

```
nyc_snow_avgs.loc['January': 'March']
```

	Avg SnowDays	Avg Precip. (cm)	Avg Low Temp. (C)
January	4.0	17.8	-3
February	2.7	22.4	-2
March	1.7	9.1	2

- Note that while using the .loc and .iloc operators, the row index must always be specified first
  - In contrast to the [] operator, where columns can be selected directly
- Hence, we get an error if we do the following:

```
nyc_snow_avgs.loc['Avg Snow Days']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

Label Indexing

- The correct way to do this is to specifically select all rows by using colon (:) operator:

```
nyc_snow_avgs.loc[:, 'Avg Snow Days']
```

```
January      4.0
February     2.7
March        1.7
April        0.2
November     0.2
December     2.3
Name: Avg Snow Days, dtype: float64
```

- You can select a specific cell value using a row label index and a column label index
- The following example shows the average number of snow days in March:

```
nyc_snow_avgs.loc['March', 'Avg Snow Days']
```

```
1.7
```

```
nyc_snow_avgs.loc['March']['Avg Snow Days']
```

```
1.7
```

Boolean Indexing

- You can also filter rows using a boolean array (similarly to NumPy arrays)
- For example, we can select which months have less than one snow day on average:

```
nyc_snow_avgs['Avg Snow Days'] < 1
```

```
January     False
February    False
March       False
April        True
November     True
December    False
Name: Avg Snow Days, dtype: bool
```

```
nyc_snow_avgs[nyc_snow_avgs['Avg Snow Days'] < 1]
```

	Avg Snow Days	Avg Precip. (cm)	Avg Low Temp. (C)
April	0.2	1.5	7
November	0.2	0.8	5

Boolean Indexing

- As for NumPy arrays, you can use the logical operators | (OR), & (AND), ~ (NOT) in boolean indexing
- The following example selects which months have more than 2 snow days on average and their average low temperature is less than 0:

```
nyc_snow_avgs[(nyc_snow_avgs['Avg Snow Days'] > 2) &
              (nyc_snow_avgs['Avg Low Temp. (C)'] < 0)]
```

	Avg Snow Days	Avg Precip. (cm)	Avg Low Temp. (C)
January	4.0	17.8	-3
February	2.7	22.4	-2

get the rows where x > 150 and y<120

Exercise

- Implement the following operations on the DataFrame animals_df:
  - Return the first 3 rows of the DataFrame
  - Select the number of visits of the animal in row 'd'
  - Select the data in rows ['b', 'e'] and in columns ['animal', 'age']
  - Select the data in rows [3, 4, 8] and in columns ['animal', 'age']
  - Select the rows where the number of visits is greater than 2
  - Select the rows where the animal is a cat and the age is less than 3
  - Select the rows where the age is missing, i.e. is NaN
  - Change the age in row 'f' to 1.5

Sorting

- Sorting a data set by some criterion is another important built-in operation
- To sort lexicographically by row or column index, use the **sort_index()** method, which returns a new, sorted object
- For example, sorting a Series object by its index:

```
ser = pd.Series(range(4), index=['d', 'b', 'a', 'c'])
ser.sort_index()
```

```
a    2
b    1
c    3
d    0
dtype: int64
```

- To sort a Series by its values, user the **sort_values()** method:

```
ser2 = pd.Series([4, 7, -3, 2])
ser2.sort_values()
```

```
2   -3
3    2
0    4
1    7
dtype: int64
```

Sorting

- With a DataFrame, you can sort by index on either axis:

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

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

```
df.sort_index()
```

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

```
df.sort_index(axis=1)
```

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

Sorting

- To sort a DataFrame by the values in the columns, use **sort_values()** with the column indexes:

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

create it

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

```
df.sort_values('b')
```

df is changed?
which parameter?

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

Sorting

- To sort by multiple columns, pass a list of names:

```
df.sort_values(['a', 'b'])
```

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

try
df.sort_values(['b', 'a'])

- The **ascending** argument can be used to change the sorting order (default is ascending)

```
df.sort_values('b', ascending=False)
```

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

Sorting

- If you want to sort each column by a different order, you can pass a list of booleans to the ascending argument:

```
df.sort_values(['a', 'b'], ascending=[False, True])
```

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

- You can use the **inplace** argument to sort the data in place:

```
df.sort_values('b', inplace=True)
df
```

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

Computing Descriptive Statistics

- pandas objects are equipped with a set of common statistical methods
- Most of these methods extract a single value (like the sum or mean) from a Series, or a series of values from the rows or columns of a DataFrame
- Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data

Computing Descriptive Statistics

- A list of summary statistics and related methods:

Method	Description
count	Number of non-NA values
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation from mean value
median	Arithmetic median (50% quantile) of values
quantile	Compute sample quantile ranging from 0 to 1
var	Sample variance of values
std	Sample standard deviation of values
cumsum	Cumulative sum of values
describe	Compute set of summary statistics for Series or each DataFrame column

Computing Descriptive Statistics

- Consider a small DataFrame:

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

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

axis=0

- Calling DataFrame's **sum()** method returns a Series containing column sums:

```
df.sum()
```

```
one    9.25
two   -5.80
dtype: float64
```

type of the result?

Computing Descriptive Statistics

- Passing **axis=1** sums over the rows instead:

```
df.sum(axis=1)
```

```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

- You can take into account NA values by using the **skipna** option:

```
df.sum(axis=1, skipna=False)
```

```
a     NaN
b    2.60
c     NaN
d   -0.55
dtype: float64
```

Computing Descriptive Statistics

- Some methods, like **idxmin** and **idxmax**, return indirect statistics like the index value where the minimum or maximum values are attained:

```
df.idxmax()
```

```
one    b
two    d
dtype: object
```

- Other methods are *accumulations*:

use cases to do that?

```
df.cumsum()
```

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

Computing Descriptive Statistics

- The method **describe()** produces multiple summary statistics in one shot:

```
df.describe()
```

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

recommendation : T

- On non-numeric data, describe produces alternate summary statistics:

```
ser = pd.Series(['a', 'a', 'b', 'c'] * 4)
ser.describe()
```

```
count     16
unique     3
top        a
freq       8
dtype: object
```

Correlation and Covariance

- Some summary statistics, like correlation and covariance, are computed from pairs of arguments
- Let's consider DataFrames of stock prices obtained from the web
- We first need to install the **pandas_datareader** module
  - Functions from pandas_datareader.data and pandas_datareader.wb extract data from various Internet sources into a pandas DataFrame
- ~~Open Anaconda command prompt and type:~~

```
pip install pandas-datareader
```

Correlation and Covariance

- The following code downloads all the stock data between 1/1/2010 and 1/1/2018 of the specified tickers:

```
import pandas_datareader.data as web
start = '1/1/2015'
end = '1/1/2018'

stocks_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    stocks_data[ticker] = web.DataReader(ticker, 'iex', start, end)
```

iex stopped the API support last June

```
stocks_data
```

```
{'AAPL':                open      high       low     close     volume
 date
 2015-01-02  104.6128  104.6597  100.8186  102.6781   53204626
 2015-01-05  101.7014  102.0395   98.9966   99.7855   64285491
 2015-01-06  100.0579  100.8937   98.2641   99.7949   65797116
 2015-01-07  100.6777  101.6169  100.2034  101.1942   40105934
 2015-01-08  102.5842  105.3265  102.0864  105.0823   59364547
 2015-01-09  105.8149  106.3596  103.5046  105.1950   53699527
```

- stocks_data is a dictionary containing a DataFrame for every specified ticker
- The index of the DataFrame is the date and its columns are open, high, low, close and volume

Correlation and Covariance

- We now create a DataFrame of the prices of all the tickers, indexed by the date:

```
prices_df = pd.DataFrame({tic: data['close']
                          for tic, data in stocks_data.items()})
prices_df.head()
```

	AAPL	GOOG	IBM	MSFT
date				
2015-01-02	102.6781	524.81	142.5291	42.9486
2015-01-05	99.7855	513.87	140.2865	42.5490
2015-01-06	99.7949	501.96	137.2610	41.9290
2015-01-07	101.1942	501.10	136.3640	42.4618
2015-01-08	105.0823	502.68	139.3278	43.7109

  - The method **head(n)** returns the first n rows of the DataFrame (the default is n = 5)

tail...

Correlation and Covariance

- We now compute percent changes of the prices:

```
price_change = prices_df.pct_change()
price_change.head()
```

pct_change()

date	AAPL	GOOG	IBM	MSFT
2015-01-02	NaN	NaN	NaN	NaN
2015-01-05	-0.028172	-0.020846	-0.015734	-0.009304
2015-01-06	0.000094	-0.023177	-0.021567	-0.014571
2015-01-07	0.014022	-0.001713	-0.006535	0.012707
2015-01-08	0.038422	0.003153	0.021734	0.029417

Correlation and Covariance

- The **corr()** method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series
- By default, it uses Pearson correlation measure

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

  - where n is the sample size, x_i and y_i are the sample points and $\bar{x}$ and $\bar{y}$ are the sample means
- For example, the correlation between the price changes of Microsoft and Google stocks is:

```
price_change.MSFT.corr(price_change.GOOG)
```

```
0.5872899839300288
```

Correlation and Covariance

- The **cov()** method of Series computes the covariance of the overlapping, non-NA, aligned-by-index values in two Series, normalized by n-1

$$\operatorname{cov}(X,Y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- For example, the covariance between the price changes of Microsoft and Google stocks is:

```
price_change.MSFT.cov(price_change.GOOG)
```

```
0.00011810635860666633
```

Correlation and Covariance

- DataFrame's **corr()** and **cov()** methods return a full correlation or covariance matrix as a DataFrame, respectively:

CC =
```
price_change.corr()
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.419992	0.330654	0.494976
GOOG	0.419992	1.000000	0.337053	0.587290
IBM	0.330654	0.337053	1.000000	0.416134
MSFT	0.494976	0.587290	0.416134	1.000000

```
price_change.cov()
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.000211	0.000086	0.000057	0.000102
GOOG	0.000086	0.000200	0.000057	0.000118
IBM	0.000057	0.000057	0.000144	0.000071
MSFT	0.000102	0.000118	0.000071	0.000202

Correlation and Covariance

- Using DataFrame's **corrwith()** method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame
- For example, to compute the correlation between IBM stock's price change and the other stocks price changes:

```
price_change.corrwith(price_change.IBM)
```

```
AAPL    0.330654
GOOG    0.337053
IBM     1.000000
MSFT    0.416134
dtype: float64
```

IBM -> to all the rest from CC ?

- Passing a DataFrame computes the correlations of matching column names

Exercise

- Calculate the sum of all visits (the total number of visits) in aminals_df
- Count the number of cats in the DataFrame
- Calculate the mean of the ages of all the animals with more than 2 visits
- Compute the correlation and covariance between the age and visits columns

what is the problem that we have?

Reading and Writing Text Files

Experis Academy
ManpowerGroup

- pandas features a number of functions for reading tabular data as a DataFrame:

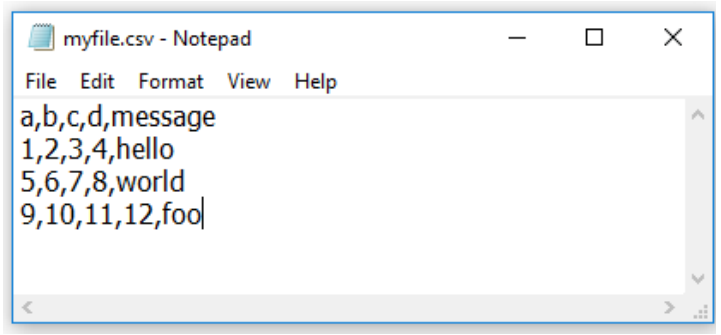
Function	Description
read_csv	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter.
read_table	Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter.
read_fwf	Read data in fixed-width column format (that is, no delimiters)

- The options for these functions fall into a few categories:
  - **Indexing**: can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all
  - **Type inference and data conversion**: this includes user-defined value conversions and custom list of missing value markers
  - **Datetime parsing**: including combining date and time fields spread over multiple columns
  - **Iterating**: support for iterating over chunks of very large files
  - **Unclean data issues**: skipping rows or a footer, comments

Reading Text Files

- Let's start with a small comma-separated (CSV) text file
- Create myfile.csv in your Notebook folder, open it with a text editor and type:

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



%%writefile myfile.csv

- Since this is comma-delimited, we can use **read_csv()** to read it into a DataFrame:

```
df = pd.read_csv('myfile.csv')
df
```

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

Reading Text Files

- A file will not always have a header row. Consider myfile2.csv:

```
!type myfile2.csv
```

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

  - !type executes the type shell command that prints the file contents (same as cat in Linux)
- To read this in, you can allow pandas to assign default column names, or you can specify names yourself:

```
pd.read_csv('myfile2.csv', header=None)
```

try reading it without header=False

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

```
pd.read_csv('myfile2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

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

so why is Jup so effective for this phase of R&D?

Reading Text Files

- To set the index of the DataFrame to one of the columns, you can either use the **index_col** argument of read_csv():

```
pd.read_csv('myfile.csv', index_col='message')
```

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

- Or you can call the **set_index()** method of the DataFrame:

```
df = pd.read_csv('myfile.csv')
df = df.set_index('message')
df
```

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

Reading Text Files

- If the fields of each row are not separated by comma, you can use the **sep** parameter to specify a character sequence or a regular expression to use to split the fields
  - You can use either read_csv() or read_table() in this case
  - read_table() is read_csv() with sep=',' replaced by sep='\t'
- For example, consider the following text file:

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

- The fields in each row are separated by a space, thus we use sep=' ' to read it:

```
pd.read_table('myfile3.txt', sep=' ')
```

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

Reading Text Files

- In some cases, a table might not have a fixed delimiter
- For example, consider this text file:

```
            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
```

https://regexr.com

- In this case fields are separated by a variable amount of whitespace
- This can be expressed by the regular expression \s+

\s = space
\+ = 1 or more

```
pd.read_table('myfile4.txt', sep='\s+')
```

	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

Because there was one fewer column name than the number of data rows, read_table() infers that the first column should be the DataFrame's index

Reading Text Files

- The parser functions have many additional arguments to help you handle the wide variety of exception file formats that might occur:

Argument	Description
path	String indicating filesystem location, URL, or file-like object
sep	Character sequence or regular expression to use to split fields in each row
header	Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row.
names	List of column names for result, combine with header=None
index_col	Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index.
skiprows	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
skip_footer	Number of lines to ignore at end of file
na_values	Sequence of values to replace with NA
comment	Character or characters to split comments off the end of lines

Reading Text Files

Experis Academy
ManpowerGroup

Argument	Description
converters	Dict containing column number of name mapping to functions. For example {'foo': f} would apply the function f to all values in the 'foo' column.
nrows	Number of rows to read from beginning of file
chunksize	For iteration, size of file chunks
encoding	Text encoding for unicode. For example 'utf-8' for UTF-8 encoded text.
squeeze	If the parsed data only contains one column return a Series
thousands	Separator for thousands, e.g. ',' or '.'
parse_dates	Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse.
dayfirst	When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False.
date_parser	Function to use to parse dates.

Reading Text Files

- For example, you can skip the first, third, and fourth rows of a file with skiprows:

```
!type myfile5.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
```

```
pd.read_csv('myfile5.csv', skiprows=[0, 2, 3])
```

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

Reading Text Files

- Handling missing values is an important part of the file parsing process
- Missing data is usually either not present (empty string) or marked by some **sentinel**
- By default, pandas recognizes as NaN a set of commonly occurring sentinels, such as NA, nan, n/a, -1.#IND, null, etc.

```
!type myfile6.csv
```

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,null,11,12,foo
```

```
pd.read_csv('myfile6.csv')
```

	something	a	b	c	d	message
0	one	1	2.0	3.0	4	NaN
1	two	5	6.0	NaN	8	world
2	three	9	NaN	11.0	12	foo

Reading Text Files

- The **na_values** option can take either a list of strings to consider missing values:

```
pd.read_csv('myfile6.csv', na_values=['foo'])
```

	something	a	b	c	d	message
0	one	1	2.0	3.0	4	NaN
1	two	5	6.0	NaN	8	world
2	three	9	NaN	11.0	12	NaN

Exercise

- Download the file euro_winners.csv from https://goo.gl/KnFPfC
- This dataset, obtained from Wikipedia, contains data for the finals of the European club championship since its inception in 1955
  - https://en.wikipedia.org/wiki/List_of_European_Cup_and_UEFA_Champions_League_finals
- Read the table into a DataFrame
- Define the season column as the index
- Display the last 5 rows of the table
- Display the name of the team who won the championship at season 2000–01 (note the dash in the middle, typed by Alt+0150)
- Display the three games with the highest number of attendees

***** is there a correlation between total number of goals each game to #of attend'?**
use apply function and split with regex (re.split(...,string))

Writing Data To Text Format

- Data can also be exported to delimited format
- Let's consider one of the CSV files read above, and change one of the table cells:

```
df = pd.read_csv('myfile6.csv')
df.loc[0, 'message'] = 'test'
df
```

	something	a	b	c	d	message
0	one	1	2.0	3.0	4	test
1	two	5	6.0	NaN	8	world
2	three	9	NaN	11.0	12	foo

- The DataFrame's **to_csv()** method writes the data out to a comma-separated file:

```
df.to_csv('out.csv')
```

```
!type out.csv
```

```
,something,a,b,c,d,message
0,one,1,2.0,3.0,4,test
1,two,5,6.0,,8,world
2,three,9,,11.0,12,foo
```

Writing Data To Text Format

- Other delimiters can be used, of course:

```
df.to_csv('out.csv', sep='|')
!type out.csv
```

```
|something|a|b|c|d|message
0|one|1|2.0|3.0|4|test
1|two|5|6.0||8|world
2|three|9||11.0|12|foo
```

- Missing values appear as empty strings in the output
- You might want to denote them by some other sentinel value:

```
df.to_csv('out.csv', na_rep='NULL')
!type out.csv
```

```
,something,a,b,c,d,message
0,one,1,2.0,3.0,4,test
1,two,5,6.0,NULL,8,world
2,three,9,NULL,11.0,12,foo
```

Writing Data To Text Format

- With no other options specified, both the row and column labels are written
- Both of these can be disabled:

```
df.to_csv('out.csv', index=False, header=False)
!type out.csv
```

```
one,1,2.0,3.0,4,test
two,5,6.0,,8,world
three,9,,11.0,12,foo
```

- You can also write only a subset of the columns, and in an order of your choosing:

```
df.to_csv('out.csv', index=False, columns=['a', 'b', 'c'])
!type out.csv
```

```
a,b,c
1,2.0,3.0
5,6.0,
9,,11.0
```

Reading Text Files In Pieces

- When processing very large files, you may only want to read in a small piece of a file or iterate through smaller chunks of the file
- Let's create a CSV file with 10,000 rows containing random numbers in four columns:

```
df = pd.DataFrame(np.random.rand(10000, 4),
                  columns=['one', 'two', 'three', 'four'])
df.to_csv('myfile7.csv', index=False)
```

- If you want to only read out a small number of rows, specify that with **nrows**:

```
pd.read_csv('myfile7.csv', nrows=5)
```

	one	two	three	four
0	0.763827	0.784181	0.141241	0.759184
1	0.495430	0.864463	0.467473	0.957617
2	0.892346	0.322783	0.591786	0.011954
3	0.209003	0.520671	0.209522	0.474396
4	0.464871	0.802502	0.387418	0.330254

first rows?
random rows?
try it...

Reading Text Files In Pieces

- To read out a file in pieces, specify a **chunksize** as a number of rows:

```
reader = pd.read_csv('myfile7.csv', chunksize=1000)
reader
```

```
<pandas.io.parsers.TextFileReader at 0x25db1899588>
```

- The TextFileReader object returned by read_csv is an iterator that allows you to iterate over the parts of the file according to the chunksize
- The following code shows the average of values of the first column in each chunk:

```
for chunk in reader:
    print(chunk['one'].mean())
```

```
0.503411622981677
0.5031108600109644
0.5086342971877635
0.5041450364490075
0.506953671321647
0.5097013179565795
0.49392377946905364
0.4947612599954584
0.4887523295903048
0.4915130217928475
```

Reading Text Files In Pieces

- TextFileReader also has a method get_chunk() that enables you to read chunks of arbitrary size:

```
reader = pd.read_csv('myfile7.csv', chunksize=1000)
reader.get_chunk(5)
```

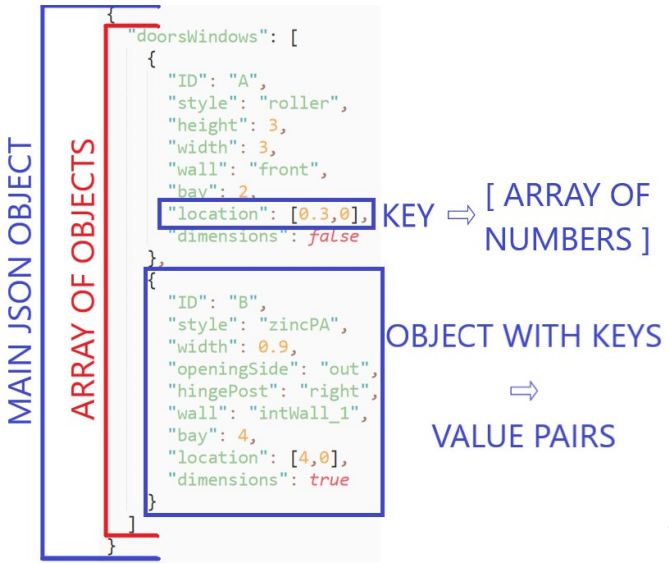
	one	two	three	four
0	0.763827	0.784181	0.141241	0.759184
1	0.495430	0.864463	0.467473	0.957617
2	0.892346	0.322783	0.591786	0.011954
3	0.209003	0.520671	0.209522	0.474396
4	0.464871	0.802502	0.387418	0.330254

```
reader.get_chunk(5)
```

	one	two	three	four
5	0.973990	0.783628	0.647234	0.985077
6	0.337996	0.616457	0.762881	0.742263
7	0.771908	0.879579	0.300900	0.246773
8	0.658600	0.622567	0.496328	0.436382
9	0.294799	0.735844	0.756918	0.258002

JSON

- JSON (JavaScript Object Notation) has become one of the standard formats for sending data between applications
- It is a much more flexible data format than a tabular text form like CSV
  - In fact, the Jupyter Notebooks themselves (.ipynb files) are stored in JSON format
- JSON types include objects (dicts), arrays (lists), strings, numbers, booleans, and nulls
- All of the keys in an object must be strings




```
{
  "Time"      : "2014-02-12 14:20:05",
  "Latitude"  : 37.33233141,
  "Longitude" : -122.0312186,
  "Count"     : 101,
  "Comments"  : "Bad Data. SNOW DAY!!",
  "Luma"      : 0,
  "Habitat"   : "Back yard, grass",
  "Types"     : [
                  "5",
                  "6"
                ],
  "Address"   : {
                  "Street"  : "2522 West Georgia Road",
                  "City"    : "Piedmont",
                  "State"   : "South Carolina",
                  "Country" : "United States",
                }
}
```

JSON

- Here is an example for defining a string in a JSON format:

```
json_str = """
{"name": "Mark",
 "age": 35,
 "places_lived": ["US", "Israel", "Spain"],
 "wife": null,
 "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
              {"name": "Katie", "age": 33, "pet": "Kitty"}]}
"""
```

- JSON is very nearly valid Python code with the exception of its null value null (Python uses None) and some other nuances (such as disallowing trailing commas at the end of lists)
- There are several Python libraries for reading and writing JSON data
- We'll use the json module which is part of the Python standard library

JSON

- **json.loads()** converts a JSON string into a Python object (a dictionary or a list):

```
import json
```

```
obj = json.loads(json_str)
obj
```

```
{'age': 35,
 'name': 'Mark',
 'places_lived': ['US', 'Israel', 'Spain'],
 'siblings': [{'age': 25, 'name': 'Scott', 'pet': 'Zuko'},
  {'age': 33, 'name': 'Katie', 'pet': 'Kitty'}],
 'wife': None}
```

```
{"employees":[
   {"name":"Ram", "email":"ram@gmail.com", "age":23},
   {"name":"Shyam", "email":"shyam23@gmail.com", "age":28},
   {"name":"John", "email":"john@gmail.com", "age":33},
   {"name":"Bob", "email":"bob32@gmail.com", "age":41}
]}
```

create json obj
retrieve age 28

- **json.dumps()** converts a Python object back to JSON:

```
obj['places_lived'].append('Germany')
json_str2 = json.dumps(obj)
json_str2
```

```
'{"name": "Mark", "age": 35, "places_lived": ["US", "Israe
l", "Spain", "Germany"], "wife": null, "siblings": [{"name":
"Scott", "age": 25, "pet": "Zuko"}, {"name": "Katie", "age":
33, "pet": "Kitty"}]}'
```

JSON

- How you convert a JSON object or list of objects to a DataFrame will be up to you
- For example, you can pass a list of JSON objects (list of dictionaries) to the DataFrame constructor and select a subset of the data fields:

```
siblings_df = pd.DataFrame(obj['siblings'], columns=['name', 'age'])
siblings_df
```

	name	age
0	Scott	25
1	Katie	33

JSON

- There is also a fast native JSON export (to_json) and decoding (read_json) to pandas
- For example:

```
df = pd.DataFrame([['a', 'b'], ['c', 'd']],
                  index=['Row1', 'Row2'],
                  columns=['Col1', 'Col2'])
```

```
df.to_json('myfile8.json')
!type myfile8.json
```

```
{"Col1":{"Row1":"a","Row2":"c"},"Col2":{"Row1":"b","Row2":"d"}}
```

```
pd.read_json('myfile8.json')
```

	Col1	Col2
Row1	a	b
Row2	c	d

Vectorized String Operations

- Pandas provides a comprehensive set of *vectorized string operations* via the **str** attribute of Pandas Series and Index object containing strings
- These vectorized string operations become most useful in the process of cleaning up messy, real-world data
- The operations correctly handle missing data (None values)
- For example, suppose we create a Pandas Series with the following data:

```
names = pd.Series(['peter', 'Paul', None, 'MARY', 'gUIDO'])
names
```

```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

Vectorized String Operations

- We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
```

```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

- Using tab completion on this str attribute will list all the vectorized string methods available to Pandas

Methods Similar to Python String Methods

- Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method
- Here is a list of Pandas str methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Methods Similar to Python String Methods

- These methods have various return values
- Some, like lower(), return a series of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

create this

```
monte.str.lower()
```

```
0    graham chapman
1       john cleese
2     terry gilliam
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

Methods Similar to Python String Methods

- But some others return numbers:

```
monte.str.len()
```

```
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

print the longest name...

- Or Boolean values:

```
monte.str.startswith('T')
```

```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

Methods Similar to Python String Methods

- Still others return lists or other compound values for each element:

```
monte.str.split()
```

```
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

- Passing **expand=True** to split() expands the splitted strings into separate columns:

```
monte.str.split(expand=True)
```

=features engineering

	0	1
0	Graham	Chapman
1	John	Cleese
2	Terry	Gilliam
3	Eric	Idle
4	Terry	Jones
5	Michael	Palin

Miscellaneous Methods

- Finally, there are some miscellaneous methods that enable other operations:

Method	Description
get()	Index each element
slice()	Slice each element
slice_replace()	Replace slice in each element with passed value
cat()	Concatenate strings
repeat()	Repeat values
normalize()	Return Unicode form of string
pad()	Add whitespace to left, right, or both sides of strings
wrap()	Split long strings into lines with length less than a given width
join()	Join strings in each element of the Series with passed separator
get_dummies()	extract dummy variables as a dataframe

Vectorized Item Access and Slicing

- The **get()** and **slice()** operations enable vectorized element access from each array
- For example, we can get a slice of the first three characters of each array using df.str.slice(0, 3)
- Note that this behavior is also available through Python's normal indexing syntax, e.g., df.str.slice(0, 3) is equivalent to df.str[0:3]:

```
monte.str[0:3]
```

```
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

- Indexing via df.str.get(i) and df.str[i] is likewise similar

Vectorized Item Access and Slicing

- These get() and slice() methods also let you access elements of arrays returned by **split()**
- For example, to extract the last name of each entry, we can combine split() and get():

```
monte.str.split().str[-1]
```

```
0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

Indicator Variables

- The method **str.get_dummies()** is useful when your data has a column containing some sort of coded indicator
- For example, we might have a dataset that contains information in the form of codes, such as A="born in America", B="born in the United Kingdom", C="likes cheese", D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
full_monte
```

	info	name
0	B|C|D	Graham Chapman
1	B|D	John Cleese
2	A|C	Terry Gilliam
3	B|D	Eric Idle
4	B|C	Terry Jones
5	B|C|D	Michael Palin

Indicator Variables

- The str.get_dummies() method lets you quickly split-out these indicator variables into a DataFrame:

```
full_monte['info'].str.get_dummies('|')
```

	A	B	C	D
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

Exercise

- Download the open recipes database from https://github.com/sameergarg/scala-elasticsearch/blob/master/conf/recipeitems-latest.json.gz
- The database is in JSON format
- Read the recipes data into a DataFrame
- Run the following queries on the data:
  - Find the name of the recipe that has the longest ingredient list
  - How many of the recipes are for breakfast food? (use the description column)
  - How many of the recipes contain precisely 3 ingredients?
- Write a function that given a list of ingredients, returns the names of all the recipes that use all those ingredients
- Test this function with the list ['parsley', 'paprika', 'tarragon']
  - You should get 10 recipes

use parameter lines=True
Read the file as a json object per line

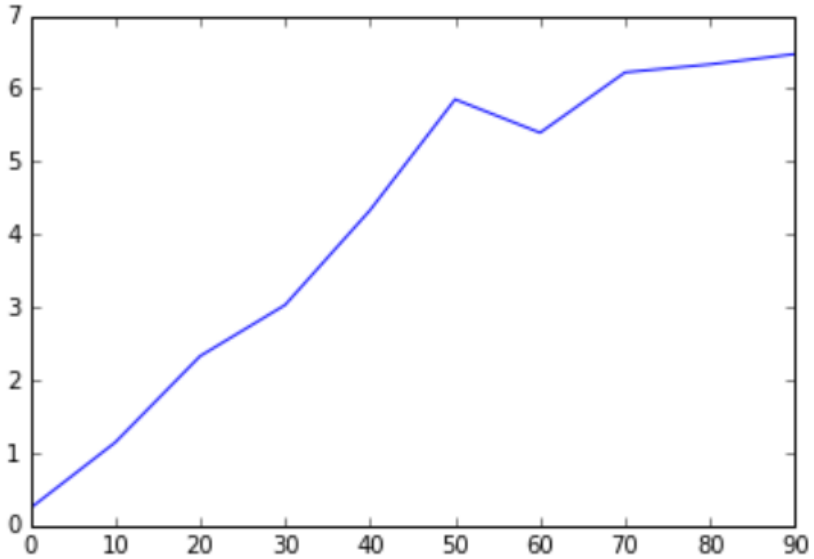
New in version 0.19.0.

Plotting Functions in Pandas

- Pandas has a number of high-level plotting methods for creating standard visualizations that take advantage of how data is organized in DataFrame objects
- Series and DataFrame have a **plot()** method for making many different plot types
- By default, they make line plots:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('classic')
import pandas as pd
import numpy as np
```

```
ser = pd.Series(np.random.randn(10).cumsum(),
                index=np.arange(0, 100, 10))
ser.plot();
```



- The Series object's index is passed to matplotlib for plotting on the X axis

Customizing Plot

- You can customize these plots by using one of the following arguments:
  - These arguments are passed through to the respective matplotlib plotting function

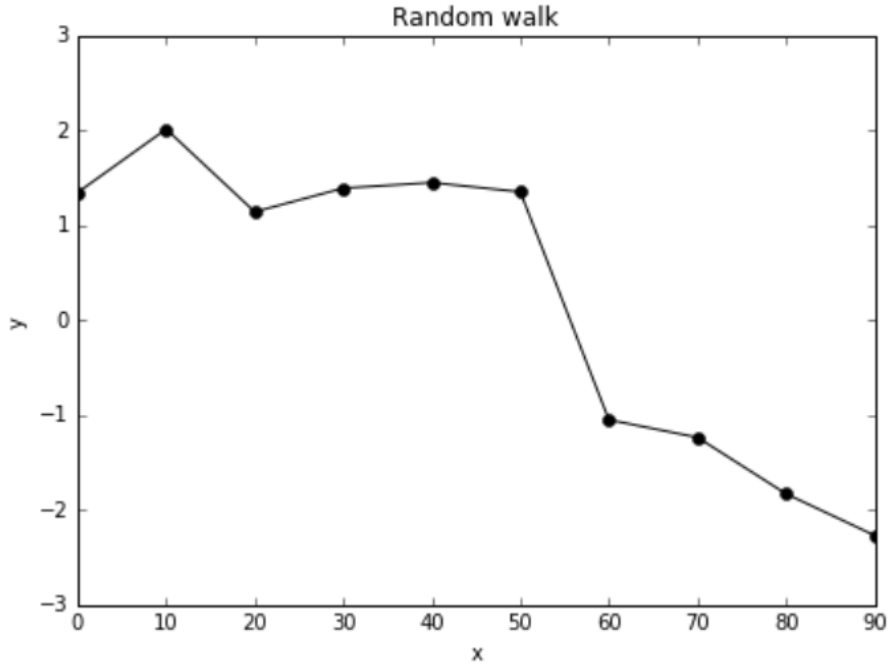
Argument	Description
kind	Can be 'line', 'bar', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie'
ax	matplotlib axes to plot on
figsize	Size of figure to create as tuple
use_index	Use the object index for tick labels
title	Title to use for the plot
grid	Axis grid lines
legend	Add a subplot legend (True by default)
style	Style string, like 'ko--', to be passed to matplotlib (one per column)
logx	Use log scaling on the X axis
logy	Use log scaling on the Y axis

Argument	Description
xticks	Values to use for X axis ticks
yticks	Values to use for Y axis ticks
xlim,	X axis limits (e.g. [0, 10])
ylim	Y axis limits
rot	Rotation of tick labels (0 through 360)
fontsize	Font size for xticks and yticks
colormap	Colormap to select colors from
xerr	X axis error data
yerr	Y axis error data
label	label argument to provide to plot
**kwds	Options to pass to matplotlib plotting method

Customizing Plot

- In addition, the plot() function returns an Axes object, so you can further customize your plot using that object, e.g., to set the labels for the x and y axes:

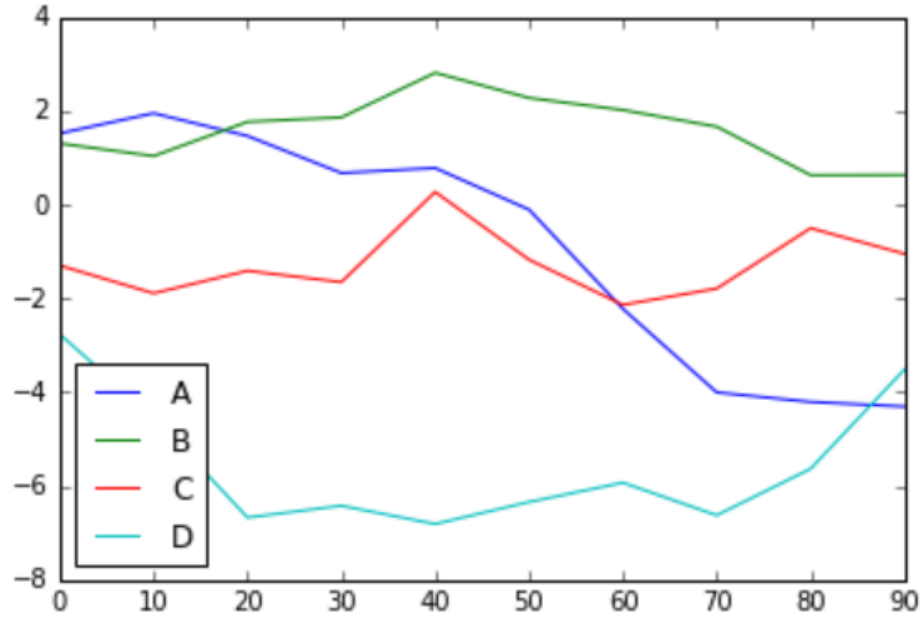
```
ser = pd.Series(np.random.randn(10).cumsum(),
                index=np.arange(0, 100, 10))
ax = ser.plot(title='Random walk', style='-ok', figsize=(7, 5))
ax.set_xlabel('x')
ax.set_ylabel('y');
```



Plotting a DataFrame

- DataFrame's plot() method plots each of its columns as a different line on the same subplot, creating a legend automatically

```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(axis=0),
                  columns=['A', 'B', 'C', 'D'],
                  index=np.arange(0, 100, 10))
df.plot();
```



Plotting a DataFrame

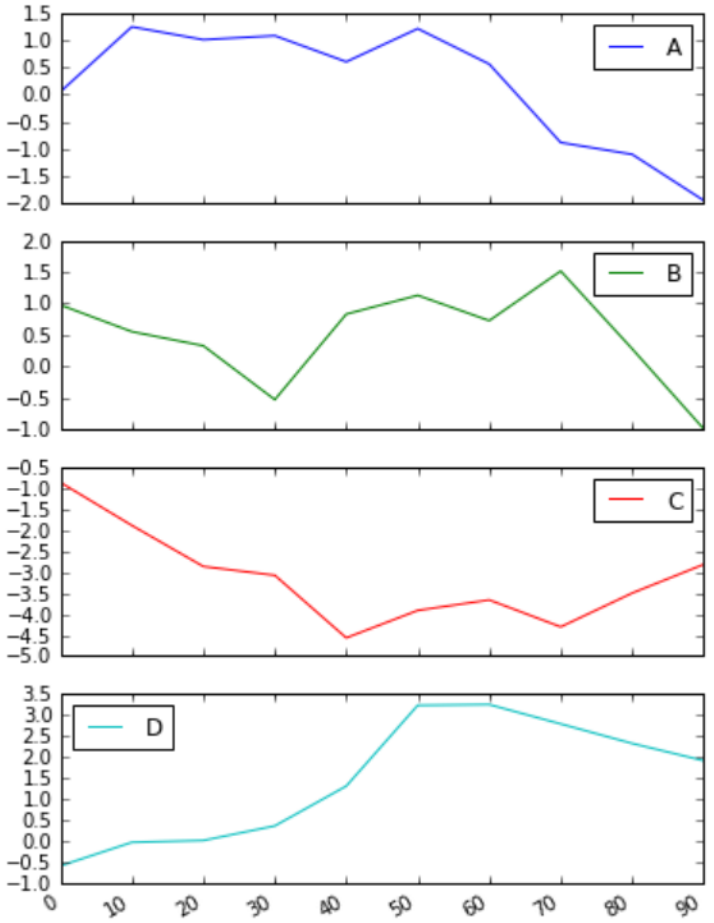
- DataFrame has a number of options allowing some flexibility with how the columns are handled
  - For example, whether to plot them all on the same subplot or to create separate subplots
- DataFrame-specific plot arguments:

Argument	Description
subplots	Plot each DataFrame column in a separate subplot
sharex	If subplots=True, share the same X axis, linking ticks and limits
sharey	If subplots=True, share the same Y axis
sort_columns	Plot columns in alphabetical order; by default uses existing column order
secondary_y	Whether to plot on the secondary y-axis If a list/tuple, which columns to plot on secondary y-axis

Plotting a DataFrame

- Example for creating a separate subplot for each column:

```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(axis=0),
                  columns=['A', 'B', 'C', 'D'],
                  index=np.arange(0, 100, 10))
df.plot(subplots=True, figsize=(6, 9));
```



Plotting a DataFrame

Experis Academy
ManpowerGroup

- The DataFrame supports the following plot types via the kind argument:

kind	Description
line	line plot (default)
bar	vertical bar plot
barh	horizontal bar plot
hist	histogram
box	boxplot
kde, density	Kernel Density Estimation plot
area	area plot
pie	pie plot
scatter	scatter plot

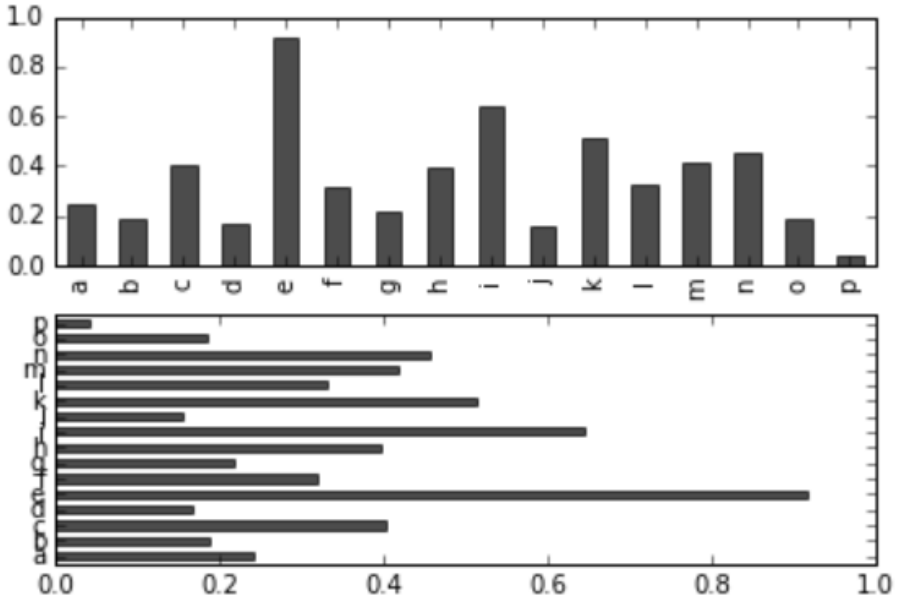
- Each plot kind has a corresponding method on the DataFrame.plot accessor, e.g., df.plot(kind='scatter') is equivalent to df.plot.scatter()

Bar Plots

- In bar plots, the Series or DataFrame index will be used as the X (bar) or Y (barh) ticks

```
fig, axes = plt.subplots(2, 1)
data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))

data.plot.bar(ax=axes[0], color='k', alpha=0.7)
data.plot.barh(ax=axes[1], color='k', alpha=0.7);
```



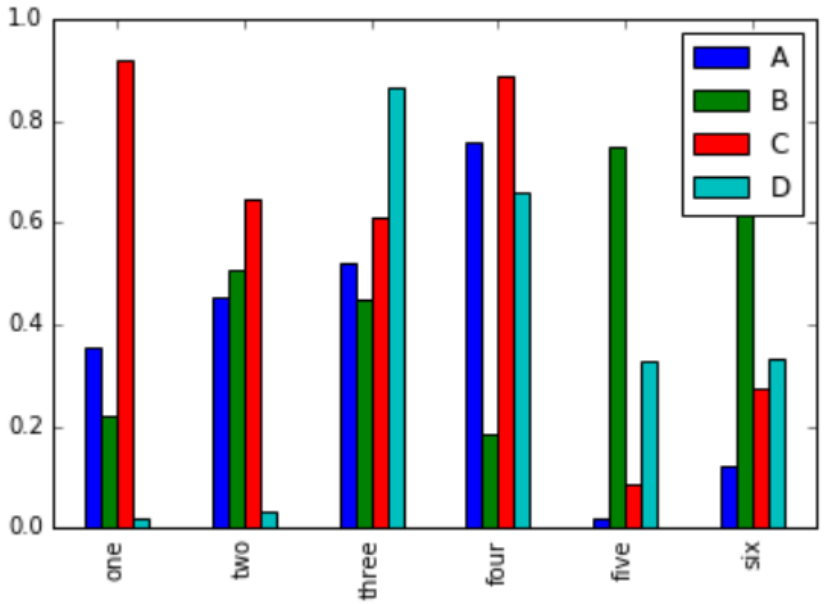
Bar Plots

- With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value:

```
df = pd.DataFrame(np.random.rand(6, 4),
                  index=['one', 'two', 'three', 'four',
                         'five', 'six'],
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
one	0.731235	0.882648	0.890212	0.711207
two	0.593067	0.290674	0.449996	0.218162
three	0.468419	0.325028	0.434517	0.162489
four	0.803957	0.652598	0.311046	0.281698
five	0.738825	0.155606	0.976214	0.032735
six	0.185331	0.288600	0.205113	0.176076

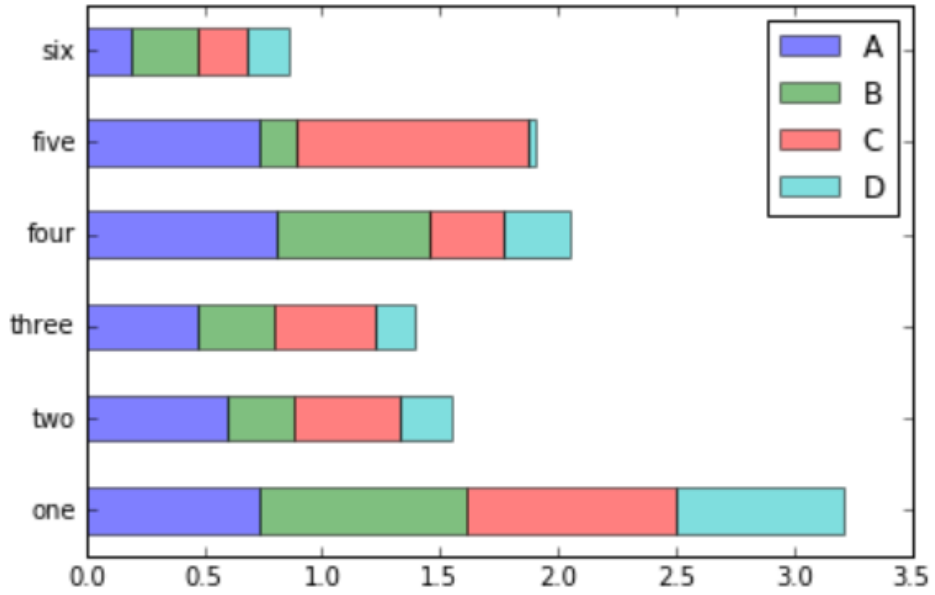
```
df.plot.bar();
```



Bar Plots

- Stacked bar plots are created from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together:

```
df.plot.barh(stacked=True, alpha=0.5);
```

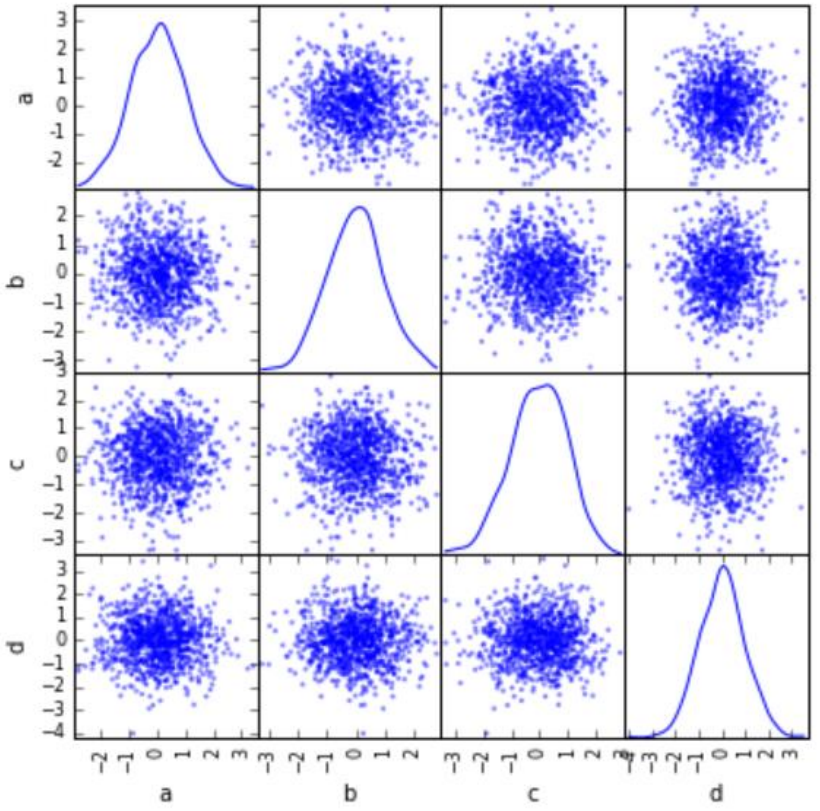


Scatter Matrix Plot

- You can create a scatter plot matrix using the **scatter_matrix()** method in the module **pandas.plotting**:

```
from pandas.plotting import scatter_matrix
```

```
df = pd.DataFrame(np.random.randn(1000, 4),
                  columns=['a', 'b', 'c', 'd'])
scatter_matrix(df, alpha=0.5, figsize=(6, 6), diagonal='kde');
```



Combining and Merging Datasets

Experis Academy
ManpowerGroup

- Some of the most interesting studies of data come from combining different data sources
- Data contained in pandas objects can be combined together in a number of built-in ways:
  - **pandas.concat()** glues or stacks together objects along an axis
  - **pandas.merge()** connects rows in DataFrames based on one or more keys
    - This will be familiar to users of SQL or other relational databases, as it implements database *join operations*

Recall: Concatenation of NumPy Arrays

- Concatenation of Series and DataFrame objects is very similar to concatenation of Numpy arrays, which can be done via the **np.concatenate()** function:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- The first argument is a list or tuple of arrays to concatenate
- Additionally, it takes an axis keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
     [3, 4]]
np.concatenate([x, x], axis=1)
```

```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Simple Concatenation

- **pd.concat()** has a similar syntax to np.concatenate(), but contains a number of additional options:

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

- For example, it can be used for a simple concatenation of two Series:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Simple Concatenation

- It also works to concatenate higher-dimensional objects, such as DataFrames:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
concat_df1_df2 = pd.concat([df1, df2])
display('df1', 'df2', 'concat_df1_df2')
```

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

concat_df1_df2

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Simple Concatenation

- By default, the concatenation takes place row-wise within the DataFrame (axis=0)
- Like np.concatenate, pd.concat() allows specification of an **axis** along which concatenation will take place:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
concat_df3_df4 = pd.concat([df3, df4], axis=1)
display('df3', 'df4', 'concat_df3_df4')
```

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

concat_df3_df4

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Concatenation with Joins

- In practice, data from different sources might have different sets of column names, and pd.concat() offers several options in this case
- Consider the concatenation of the following two DataFrames, which have some (but not all!) columns in common:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
concat_df5_df6 = pd.concat([df5, df6])
display('df5', 'df6', 'concat_df5_df6')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

concat_df5_df6

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

Concatenation with Joins

- By default, the entries for which no data is available are filled with NA values
- To change this, we can use the **join** and **join_axes** parameters of pd.concat()
- By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns using join='inner':

```
concat_df5_df6 = pd.concat([df5, df6], join='inner')
display('df5', 'df6', 'concat_df5_df6')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

concat_df5_df6

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Exercise

- Predict the result of the following operations:

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

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

```
pd.concat([s1, s2, s3], axis=1)
```

```
s4 = pd.concat([s1 * 5, s3])
pd.concat([s1, s4], axis=1)
```

```
pd.concat([s1, s4], axis=1, join='inner')
```

```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

```
pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

Merging Datasets

- One essential feature offered by Pandas is its high-performance, in-memory join and merge operations
- Merge or join operations combine datasets by linking rows using one or more keys
- These operations are central to relational databases
- The **pd.merge()** function is the main entry point for using these algorithms
- pd.merge() implements a number of types of joins:
  - one-to-one
  - many-to-one
  - many-to-many

One-To-One Joins

- The simplest type of merge situation is the **one-to-one** join, in which each DataFrame has one value for each key
- Consider the following two DataFrames which contain information on several employees in a company:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

One-To-One Joins

- We can use **pd.merge()** to combine this information into a single DataFrame:

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

- pd.merge() recognizes that each DataFrame has an "employee" column, and automatically joins using this column as a key
  - If not specified, pd.merge() uses the overlapping column names as the keys
- The order of entries in each column is not necessarily maintained, and generally the merge discards the index (unless performing merges by index)

Many-To-One Joins

- Joins in which one of the two key columns contains duplicates
- The resulting DataFrame will preserve those duplicate entries as appropriate
- For example:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

Many-To-Many Joins

- Joins in which both the left and right DataFrames contain duplicates
- Consider the following, where we have a DataFrame showing one or more skills associated with a particular group:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

Specifying the Merge Key

- By default pd.merge() looks for one or more matching column names between the two inputs, and uses this as the key
- However, often the column names will not match so nicely, and pd.merge() provides a variety of options for handling this
- Most simply, you can explicitly specify the name of the key column using the **on** keyword, which takes a column name or a list of column names:

```
df3 = pd.merge(df1, df2, on='employee')
display('df1', 'df2', 'df3')
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Specifying the Merge Key

- At times you may wish to merge two datasets with different column names
- For example, we may have a dataset in which the employee name is labeled as "name" rather than "employee“
- In this case, we can use the **left_on** and **right_on** keywords to specify the two column names:

```
df6 = pd.DataFrame({'name': ['Jake', 'Sue', 'Lisa', 'Bob'],
                    'salary': [70000, 80000, 120000, 90000]})
df7 = pd.merge(df1, df6, left_on='employee', right_on='name')
display('df1', 'df6', 'df7')
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df6

	name	salary
0	Jake	70000
1	Sue	80000
2	Lisa	120000
3	Bob	90000

df7

	employee	group	name	salary
0	Bob	Accounting	Bob	90000
1	Jake	Engineering	Jake	70000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	80000

Overlapping Column Names

- Finally, you may end up in a case where your two input DataFrames have conflicting column names

```
df10 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'rank': [1, 2, 3, 4]})
df11 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'rank': [3, 1, 4, 2]})
display('df10', 'df11', 'pd.merge(df10, df11, on="name")')
```

df10

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df11

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df10, df11, on="name")

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

- Because the output would have two conflicting column names, the merge function automatically appends a suffix _x or _y to make the output columns unique

Overlapping Column Names

- If these defaults are inappropriate, it is possible to specify a custom suffix using the **suffixes** keyword:

```
pd.merge(df10, df11, on='name', suffixes=['_L', '_R'])
```

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

- These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns

Exercise

- Download the following data files containing some data about US states and their populations from https://goo.gl/KnFPfC
  - state-population.csv
  - state-areas.csv
  - state-abbrevs.csv
- Use Pandas to rank the states by their 2010 total population density from the most dense state to the least dense state
- The first 5 rows of the result should look like this:

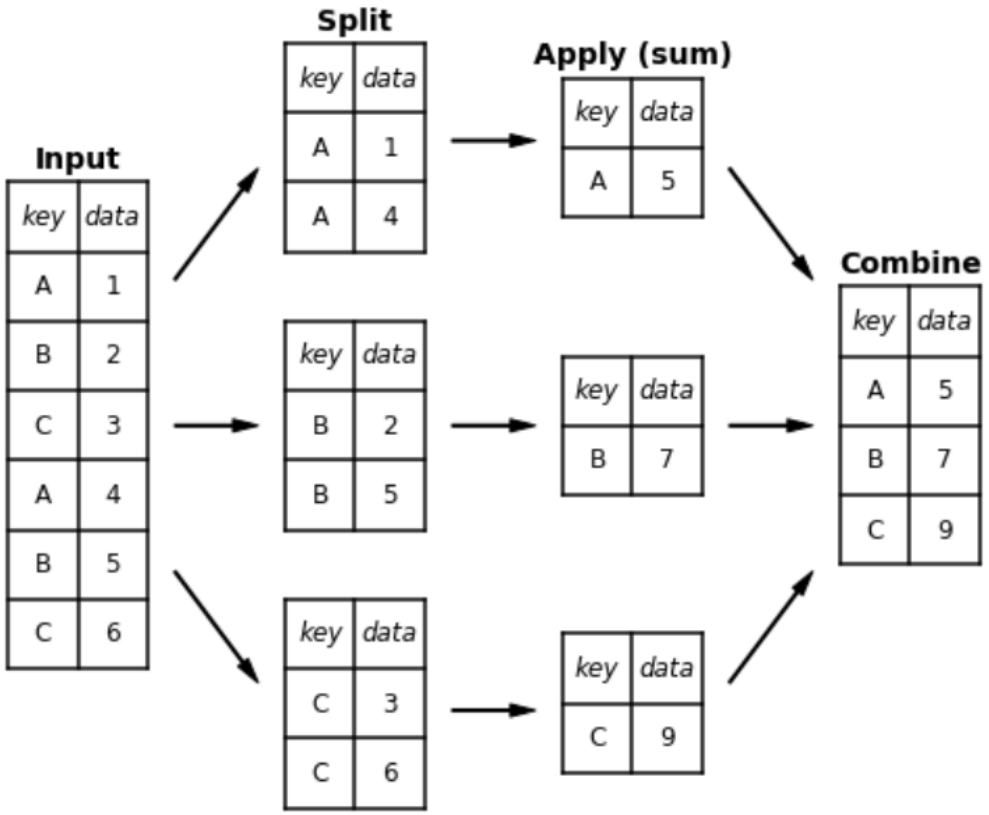
	state	population	area (sq. mi)	density
0	District of Columbia	605125.0	68	8898.897059
1	Puerto Rico	3721208.0	3515	1058.665149
2	New Jersey	8802707.0	8722	1009.253268
3	Rhode Island	1052669.0	1545	681.339159
4	Connecticut	3579210.0	5544	645.600649

Aggregation and Grouping

- Categorizing a data set and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow
- After loading, merging, and preparing a data set, a familiar task is to compute group statistics or possibly pivot tables for reporting or visualization purposes
- Pandas provides a flexible and high-performance groupby facility, enabling you to slice and dice, and summarize data sets in a natural way

GroupBy

- Group operations can be described by the **split-apply-combine** process:



The *split* step breaks up and group a DataFrame depending on the value of the specified key.

The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

The *combine* step merges the results of these operations into an output array.

GroupBy

- GroupBy typically performs all these steps in a single pass over the data
- The user need not think about *how* the computation is done under the hood, but rather think about the *operation as a whole*
- As a concrete example, let's take a look at using Pandas for the computation shown in this diagram. We'll start by creating the input DataFrame:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(1, 7)},
                  columns=['key', 'data'])
df
```

	key	data
0	A	1
1	B	2
2	C	3
3	A	4
4	B	5
5	C	6

GroupBy

- The most basic group by operation can be computed with the **groupby()** method of DataFrame, passing the name of the desired key column:

```
df.groupby('key')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x000001C2E7A0ECF8>
```

- The result is a DataFrameGroupBy object, which is a special view of the DataFrame, that has all of the information needed to then apply some operation to each group
- To produce a result, we can apply an aggregate to this object, which will perform the appropriate apply/combine steps to produce the desired result:

```
df.groupby('key').sum()
```

	data
key	
A	5
B	7
C	9

The GroupBy Object

- The GroupBy object is a very flexible abstraction
- In many ways, you can treat it as if it's a collection of DataFrames
- The most important operations made available by a GroupBy are *aggregate*, *filter*, *transform*, and *apply*
- Let's see some examples using the Planets data, available via the seaborn package

Planets Data

- The Planets dataset gives information on planets that astronomers have discovered around other stars
- It can be downloaded with a simple Seaborn command:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

```
(1035, 6)
```

```
planets.head()
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

- This has some details on the 1,000+ extrasolar planets discovered up to 2014

Column Indexing

- The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object:

```
planets.groupby('method')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x000001C2E9B95860>
```

```
planets.groupby('method')['orbital_period']
```

```
<pandas.core.groupby.SeriesGroupBy object at 0x000001C2E9B810B8>
```

- Here we've selected a particular Series group from the original DataFrame group by reference to its column name

Column Indexing

- As with the GroupBy object, no computation is done until we call some aggregate on the object:

```
planets.groupby('method')['orbital_period'].median()
```

```
method
Astrometry                       631.180000
Eclipse Timing Variations       4343.500000
Imaging                        27500.000000
Microlensing                    3300.000000
Orbital Brightness Modulation      0.342887
Pulsar Timing                     66.541900
Pulsation Timing Variations     1170.000000
Radial Velocity                  360.200000
Transit                            5.714932
Transit Timing Variations         57.011000
Name: orbital_period, dtype: float64
```

- This gives an idea of the general scale of orbital periods (in days) that each method of discovery is sensitive to

Aggregation

- We're already familiar with GroupBy aggregations such as sum(), median(), etc.
- But the aggregate() method allows for even more flexibility
- It can take a string, a function, or a list thereof, and compute all the aggregates at once
- Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Aggregation

- Another useful pattern is to pass a dictionary mapping column names to operations to be applied on that column:

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

key	data1	data2
A	0	5
B	1	7
C	2	9

Filtering

- A filtering operation allows you to drop data based on the group properties
- For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

```
def filter_func(x):
    return x['data2'].std() > 4

filtered = df.groupby('key').filter(filter_func)
display('df', "df.groupby('key').std()", 'filtered')
```

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').std()

key	data1	data2
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

filtered

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

Transformation

- While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine
- The method **transform()** gets a Series containing the group elements and returns the same Series after transformation
- A common example is to center the data by subtracting the group-wise mean:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

The apply() Method

- The apply() method lets you apply an arbitrary function to the group results
- The function should take a DataFrame, and return either a Pandas object (e.g., DataFrame, Series) or a scalar
- The combine operation will be tailored to the type of output returned.
- For example, we can normalize the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

normalized = df.groupby('key').apply(norm_by_data2)
normalized
display('df', 'normalized')
```

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

normalized

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

Specifying the Split Key

- So far we split the DataFrame on a single column name
- This is just one of many options by which the groups can be defined
- For example, you can specify a list of column names to group by:

```
planets.groupby(['method', 'year'])['distance'].mean()
```

```
method                          year
Astrometry                      2010      14.980000
                                2013      20.770000
Eclipse Timing Variations       2008     130.720000
                                2009            NaN
                                2010     500.000000
                                2011            NaN
                                2012            NaN
Imaging                         2004     110.466667
                                2005      45.520000
                                2006      29.620000
                                2007     165.000000
```

Specifying the Split Key

- If you want to be more specific, the split key can be any series or list with a length matching that of the DataFrame:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
```

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(L).sum()

	data1	data2
0	7	17
1	4	3
2	4	7

Specifying the Split Key

- Another method is to provide a dictionary that maps **index** values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(mapping).sum()

	data1	data2
consonant	12	19
vowel	3	8

Specifying the Split Key

- Similar to mapping, you can pass any Python function that will input the **index** value and output the group:

```
display('df2', 'df2.groupby(str.lower).mean()')
```

df2

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(str.lower).mean()

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

Specifying the Split Key

- Further, any of the preceding key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
```

		data1	data2
a	vowel	1.5	4.0
b	consonant	2.5	3.5
c	consonant	3.5	6.0

Exercise

- Using the planet data, count the discovered planets by method and by decade
- Your output should look like this:

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

Pivot Tables

- We have seen how GroupBy lets us explore relationships within a dataset
- A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data
- The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data
- You can think of pivot tables as a multidimensional version of GroupBy aggregation
- That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid

Motivating Pivot Tables

- For the illustration of pivot tables, we'll use the database of passengers on the *Titanic*, available through the Seaborn library:

```
import numpy as np
import pandas as pd
import seaborn as sns
```

```
titanic = sns.load_dataset('titanic')
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	de
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	N
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	N
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	N

Pivot Tables by Hand

- To start learning more about this data, we might begin by grouping according to gender, survival status, or some combination thereof
- For example, let's look at survival rate by gender, using a GroupBy operation:

```
titanic.groupby('sex')['survived'].mean()
```

```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

- This is useful, but we might like to go one step deeper and look at survival by both sex and class

Pivot Tables by Hand

- Using a GroupBy operation, we might proceed like this:
  - *We group by* class and gender
  - *select* survival
  - *apply* a mean aggregate
  - *combine* the resulting groups
  - *unstack* the hierarchical index to reveal the hidden multidimensionality

```
titanic.groupby(['sex', 'class'])['survived'].mean().unstack()
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

- This 2D GroupBy is common enough that Pandas includes the function **pivot_table()**, which succinctly handles this type of multi-dimensional aggregation

Pivot Table Syntax

- Here is the equivalent to the preceding operation using the pivot_table() method of DataFrames:

```
titanic.pivot_table('survived', index='sex', columns='class')
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

- This is definitely more readable than the groupby approach, and produces the same result

Multi-Level Pivot Tables

- Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options
- For example, we might be interested in looking at age as a third dimension
- We'll bin the age using **pd.cut()**, which bins values into discrete intervals:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

Multi-Level Pivot Tables

- We can apply the same strategy when working with the columns as well
- Let's add info on the fare paid using **pd.qcut()** to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 3)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

	fare	(-0.001, 8.662]		(8.662, 26.0]			(26.0, 512.329]		
	class	First	Third	First	Second	Third	First	Second	Third
sex	age								
female	(0, 18]	NaN	0.700000	NaN	1.000000	0.583333	0.909091	1.0	0.111111
	(18, 80]	NaN	0.523810	1.0	0.877551	0.433333	0.972222	1.0	0.125000
male	(0, 18]	NaN	0.166667	NaN	0.500000	0.500000	0.800000	0.8	0.052632
	(18, 80]	0.0	0.127389	0.0	0.086957	0.102564	0.400000	0.0	0.500000

- The result is a four-dimensional aggregation with hierarchical indices

Additional Pivot Table Options

- The **aggfunc** keyword controls what type of aggregation is applied, which is a mean by default
- As in the GroupBy, the aggregation specification can be a string representing one of several common choices (e.g., 'sum', 'mean', 'count', 'min', 'max', etc.) or a function that implements an aggregation (e.g., np.sum(), min(), sum(), etc.)
- Also, it can be specified as a dictionary mapping a column to any of the above:

```
titanic.pivot_table(index='sex', columns='class',
                    aggfunc={'survived': sum, 'fare': 'mean'})
```

	fare			survived		
class	First	Second	Third	First	Second	Third
sex						
female	106.125798	21.970121	16.118810	91	70	72
male	67.226127	19.741782	12.661633	45	17	47

Additional Pivot Table Options

- At times it's useful to compute totals along each grouping
- This can be done via the **margins** keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

class	First	Second	Third	All
sex				
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

  - This automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%
  - The margin label can be specified with the margins_name keyword, which defaults to "All"

Time Series

- Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data
- Date and time data comes in a few flavors:
  - *Time stamps* reference particular moments in time (e.g., July 4th, 2015 at 7:00am).
  - *Time intervals* reference a length of time between a particular beginning and end point, for example, the year 2015
  - *Periods* usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods comprising days)
  - *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds)

Time Series Data Structures

- For time stamps, Pandas provides the **Timestamp** type
  - This is essentially a replacement for Python's native datetime, but is based on the more efficient numpy.datetime64 data type
  - The associated Index structure is DatetimeIndex
- For time Periods, Pandas provides the **Period** type
  - This encodes a fixed-frequency interval based on numpy.datetime64
  - The associated index structure is PeriodIndex
- For time deltas or durations, Pandas provides the **Timedelta** type
  - Timedelta is a more efficient replacement for Python's native datetime.timedelta type, and is based on numpy.timedelta64
  - The associated index structure is TimedeltaIndex

Timestamp

- Pandas provides a **Timestamp** object, which combines the ease-of-use of Python's datetime with the efficient storage and vectorized interface of numpy.datetime64
- **pd.to_datetime()** can be used to create a Timestamp from a wide variety of formats:

```
date = pd.to_datetime("4th of July, 2018")
date
```

```
Timestamp('2018-07-04 00:00:00')
```

```
date.strftime('%A')
```

```
'Wednesday'
```

- We can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

```
DatetimeIndex(['2018-07-04', '2018-07-05', '2018-07-06', '2018-07-07',
               '2018-07-08', '2018-07-09', '2018-07-10', '2018-07-11',
               '2018-07-12', '2018-07-13', '2018-07-14', '2018-07-15'],
              dtype='datetime64[ns]', freq=None)
```

Indexing By Time

- Where the Pandas time series tools really become useful is when you begin to *index data by timestamps*
- For example, we can construct a Series object that has time indexed data:

```
index = pd.DatetimeIndex(['2017-07-04', '2017-08-04',
                          '2018-07-04', '2018-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2017-07-04    0
2017-08-04    1
2018-07-04    2
2018-08-04    3
dtype: int64
```

- We can now use any of the Series indexing patterns, passing date values:

```
data['2017-07-04':'2018-07-04']
```

```
2017-07-04    0
2017-08-04    1
2018-07-04    2
dtype: int64
```

Indexing By Time

- There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
data['2018']
```

```
2018-07-04    2
2018-08-04    3
dtype: int64
```

- When **pd.to_datetime()** is called with a series of dates, it yields a DatetimeIndex:

```
dates = pd.to_datetime([datetime(2018, 7, 3), '4th of July, 2018',
                        '2018-Jul-6', '07-07-2018', '20180708'])
dates
```

```
DatetimeIndex(['2018-07-03', '2018-07-04', '2018-07-06', '2018-07-07',
               '2018-07-08'],
              dtype='datetime64[ns]', freq=None)
```

Indexing By Time

- Any DatetimeIndex can be converted to a PeriodIndex with the **to_period()** function with the addition of a frequency code
- Here we'll use 'D' to indicate daily frequency:

```
dates.to_period('D')
```

```
PeriodIndex(['2018-07-03', '2018-07-04', '2018-07-06', '2018-07-07',
             '2018-07-08'],
            dtype='period[D]', freq='D')
```

- A TimedeltaIndex is created, for example, when a date is subtracted from another:

```
dates - dates[0]
```

```
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
dtype='timedelta64[ns]', freq=None)
```

Regular Sequences

- To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose:
  - **pd.date_range()** for timestamps
  - **pd.period_range()** for periods
  - **pd.timedelta_range()** for time deltas
- We've seen that Python's range() and NumPy's np.arange() turn a startpoint, endpoint, and optional stepsize into a sequence
- Similarly, pd.date_range() accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates
- By default, the frequency is one day:

```
pd.date_range('2018-07-27', '2018-08-03')
```

```
DatetimeIndex(['2018-07-27', '2018-07-28', '2018-07-29', '2018-07-30',
               '2018-07-31', '2018-08-01', '2018-08-02', '2018-08-03'],
              dtype='datetime64[ns]', freq='D')
```

Regular Sequences

- Alternatively, the date range can be specified not with a start and endpoint, but with a startpoint and a number of periods:

```
pd.date_range('2018-07-27', periods=8)
```

```
DatetimeIndex(['2018-07-27', '2018-07-28', '2018-07-29', '2018-07-30',
               '2018-07-31', '2018-08-01', '2018-08-02', '2018-08-03'],
              dtype='datetime64[ns]', freq='D')
```

- The spacing can be modified by altering the **freq** argument, which defaults to D
- For example, here we will construct a range of hourly timestamps:

```
pd.date_range('2018-07-27', periods=8, freq='H')
```

```
DatetimeIndex(['2018-07-27 00:00:00', '2018-07-27 01:00:00',
               '2018-07-27 02:00:00', '2018-07-27 03:00:00',
               '2018-07-27 04:00:00', '2018-07-27 05:00:00',
               '2018-07-27 06:00:00', '2018-07-27 07:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Regular Sequences

- To create regular sequences of Period or Timedelta values, the very similar **pd.period_range()** and **pd.timedelta_range()** functions are useful
- Here are some monthly periods:

```
pd.period_range('2018-07', periods=8, freq='M')
```

```
PeriodIndex(['2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-12',
             '2019-01', '2019-02'],
            dtype='period[M]', freq='M')
```

- And a sequence of durations increasing by an hour:

```
pd.timedelta_range(0, periods=10, freq='H')
```

```
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

Frequencies and Offsets

- Just as we saw the D (day) and H (hour) codes above, we can use such codes to specify any desired frequency spacing
- The following table summarizes the main codes available:

Code	Description
D	Calendar day
W	Weekly
M	Month end
Q	Quarter end
A	Year end
H	Hours
T	Minutes
S	Seconds
L	Milliseconds
U	Microseconds

Code	Description
B	Business day
BM	Business month end
BQ	Business quarter end
BA	Business year end
BH	Business hours

Frequencies and Offsets

- The monthly, quarterly, and annual frequencies are all marked at the end of the specified period
- By adding an **S** suffix to any of these, they instead will be marked at the beginning:

Code	Description
MS	Month start
QS	Quarter start
YS	Year start

Code	Description
BMS	Business month start
BQS	Business quarter start
BAS	Business year start

- Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:
  - Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
  - A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.
- In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code: W-SUN, W-MON, W-TUE, W-WED, etc.

Frequencies and Offsets

- On top of this, codes can be combined with numbers to specify other frequencies
- For example, for a frequency of 2 hours 30 minutes, we can combine the hour (H) and minute (T) codes as follows:

```
pd.timedelta_range(0, periods=9, freq='2H30T')
```

```
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
                '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
               dtype='timedelta64[ns]', freq='150T')
```

Resampling, Shifting and Windowing

- The ability to use dates and times as indices provides the benefits of indexed data in general (e.g., automatic alignment during operations, intuitive data slicing and access, etc.), and Pandas provides several additional time series-specific operations
- We will take a look at a few of those, using Google's stock closing price history:

```
from pandas_datareader import data

goog = data.DataReader('GOOG', start='2014', end='2018',
                       data_source='iex')
goog.head()
```

5y

date	open	high	low	close	volume
2014-03-27	568.000	568.00	552.92	558.46	13052
2014-03-28	561.200	566.43	558.67	559.99	41003
2014-03-31	566.890	567.00	556.93	556.97	10772
2014-04-01	558.710	568.45	558.71	567.16	7932
2014-04-02	565.106	604.83	562.19	567.00	146697

Resampling, Shifting and Windowing

- For simplicity, we'll use just the closing price:

```
goog = goog['close']
```

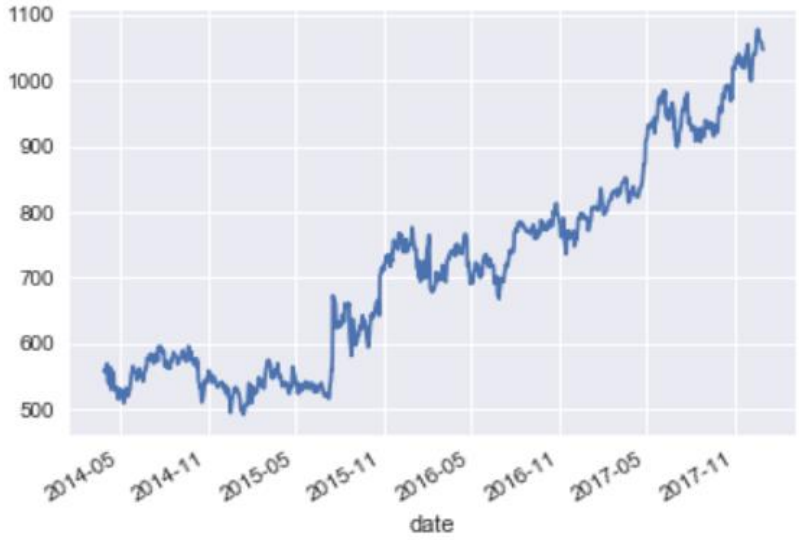
- We will also convert the index into a datetime index:

```
# Converting the index into a datetime index
goog.index = pd.to_datetime(goog.index)
```

- We can visualize this using plot(), after the normal Matplotlib setup boilerplate:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn
seaborn.set()
```

```
goog.plot();
```



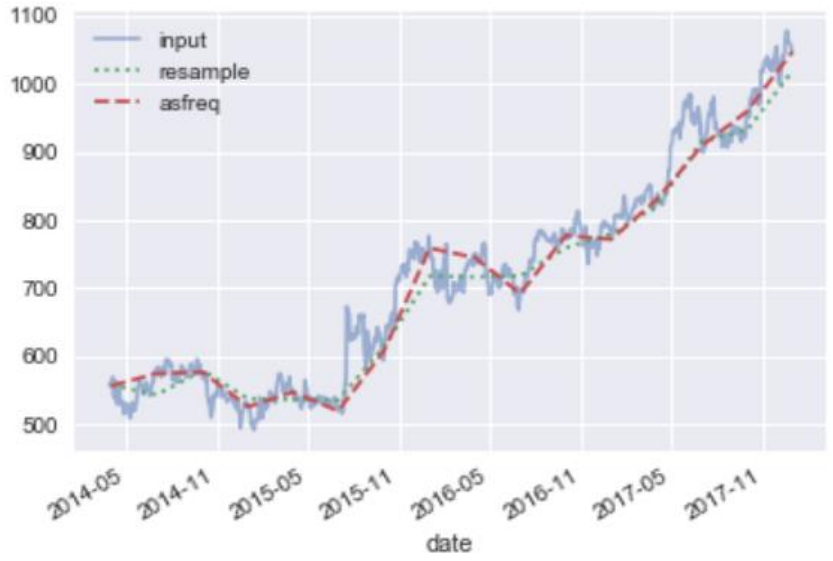
Resampling

- One common need for time series data is resampling at a higher or lower frequency
- This can be done using the **resample()** method, or the much simpler **asfreq()** method
- The primary difference between the two is that resample() is fundamentally a *data aggregation*, while asfreq() is fundamentally a *data selection*
- For example, let's down-sample the data at the end of business quarters:

```
goog.plot(alpha=0.5, style='-')

# report the average of the previous quarter
goog.resample('BQ').mean().plot(style=':')

# report the value at the end of the quarter
goog.asfreq('BQ').plot(style='--')
plt.legend(['input', 'resample', 'asfreq']);
```



Resampling

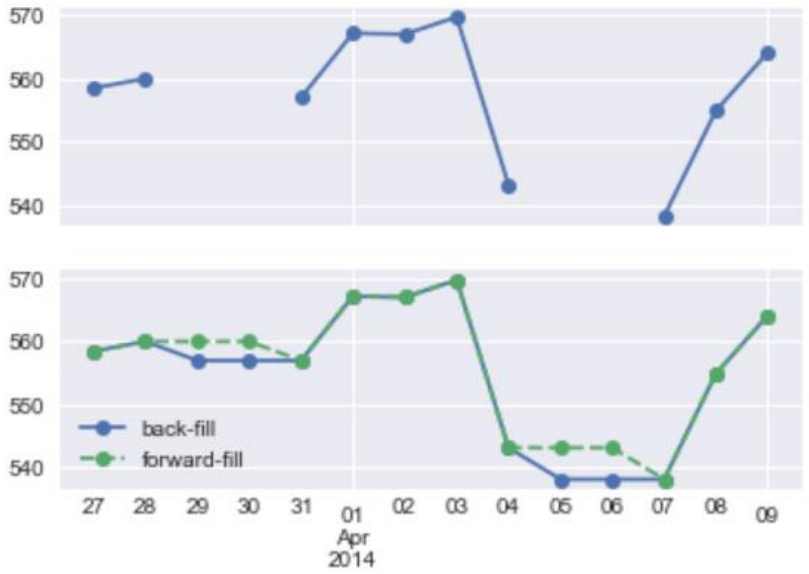
- For up-sampling, resample() and asfreq() are largely equivalent, though resample has many more options available
- The default for both methods is to fill the up-sampled points with NA values
- However, they accept a **method** argument to specify how values are imputed
- For example, let's resample the stock data at a daily frequency (including weekends):

```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')

ax[1].legend(["back-fill", "forward-fill"]);
```



Time Shifts

- Another common time series-specific operation is shifting of data in time
- Pandas has two closely related methods for computing this: **shift()** and **tshift()**
- The difference is that shift() *shifts the data*, while tshift() *shifts the index*
- In both cases, the shift is specified in multiples of mdates.DateFormatter the frequency
- Here we will both shift() and tshift() by 400 days:

Time Shifts

```
fig, ax = plt.subplots(3, figsize=(7, 10))

# apply a frequency to the data
goog = goog.asfreq('D', method='ffill')

goog.plot(ax=ax[0])
goog.shift(400).plot(ax=ax[1])
goog.tshift(400).plot(ax=ax[2])

# legends and annotations
local_max = pd.to_datetime('2015-7-17')
offset = pd.Timedelta(400, 'D')

ax[0].legend(['input'], loc=2)
ax[0].axvline(local_max, alpha=0.3, color='red')
# set monthly locator

ax[1].legend(['shift(400)'], loc=2)
ax[1].axvline(local_max + offset, alpha=0.3, color='red')

ax[2].legend(['tshift(400)'], loc=2)
ax[2].axvline(local_max + offset, alpha=0.3, color='red');
```

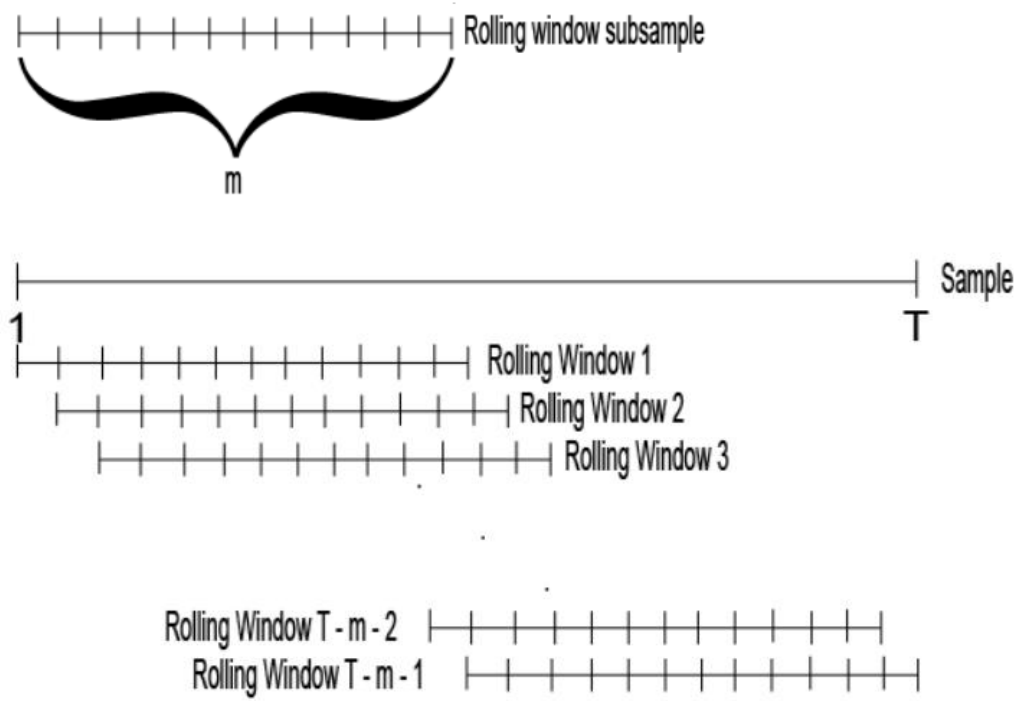
Time Shifts

- A common context for this type of shift is in computing differences over time
- For example, we can use shifted values to compute the one-year return on investment for Google stock over the course of the dataset:

```
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');
```

Rolling Windows

- In rolling windows analysis, we divide the data set into overlapping sub-samples, and calculate the aggregated values by moving the window forward one observation at a time:



Rolling Windows

- The **rolling()** method of Series and DataFrame provides rolling window calculations:

```
ser = pd.Series(np.arange(10, 16))
ser
```

```
0    10
1    11
2    12
3    13
4    14
5    15
dtype: int32
```

```
# Rolling sum for a 3 rows window
ser.rolling(3).sum()
```

```
0     NaN
1     NaN
2    33.0
3    36.0
4    39.0
5    42.0
dtype: float64
```

- As with group-by operations, the aggregate() and apply() methods can be used for custom rolling computations

Rolling Windows

- Rolling-window analysis of a time-series model assesses:
  - The stability of the model over time, by examining whether the coefficients of the model are time-invariant
  - The forecast accuracy of the model
- For example, here is the one-year centered rolling mean and standard deviation of the Google stock prices:

```
rolling = goog.rolling(365, center=True)
data = pd.DataFrame({'input': goog,
                     'one-year rolling_mean': rolling.mean(),
                     'one-year rolling_std': rolling.std()})
ax = data.plot(style=['-', '--', ':']);
ax.lines[0].set_alpha(0.5)
```