pandas

In [1]:

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

basic data structure in padas

pandas provides two types of classes for handling data:

```
1.Series:
    a one -dimensional lableled array holding data of any type


 2.Dataframe:
    a two dimensional data structure that holds data like a two dimension array
or table with rows and columns.
```

Object creation

Creating a Series by passing a list of values, letting pandas create a default RangeIndex.

In [17]:

```
s=pd.Series([1,3,5,np.nan,6,8])
```

In [18]:

```
s
```

Out[18]:

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a DataFrame by passing a NumPy array with a datetime index using date_range() and labeled columns:

In [19]:

```
dates=pd.date_range("20130101",periods=6)
```

In [20]:

```
dates
```

Out[20]:

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [21]:

```
#dates=pd.date_range("20090501",periods=4)
```

In [22]:

```
#dates
```

In [24]:

```
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
```

In [25]:

```
df
```

Out[25]:

	A	B	C	D
2013-01-01	-0.788755	-1.317542	0.963032	-1.938138
2013-01-02	-0.995125	2.827648	0.118524	-1.156898
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126
2013-01-04	-0.615557	0.599002	-0.791178	1.423080
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057
2013-01-06	-1.205508	-0.792925	0.913221	-1.263441

Creating a DataFrame by passing a dictionary of objects where the keys are the column labels and the values are the column values.

In [28]:

```
df2=pd.DataFrame({
    "A":1.0,
    "B":pd.Timestamp("20130102"),
    "c":pd.Series(1,index=list(range(4)),dtype="float32"),
    "D":np.array([3]*4,dtype="int32"),
    "E": pd.Categorical(["test","train","test","train"]),
    "F": "foo",
}
)
```

In [29]:

```
df2
```

Out[29]:

	A	B	c	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

The columns of the resulting DataFrame have different dtypes:

In [30]:

```
df2.dtypes
```

Out[30]:

```
A             float64
B     datetime64[ns]
c             float32
D               int32
E            category
F              object
dtype: object
```

In [31]:

```
df.dtypes
```

Out[31]:

```
A    float64
B    float64
C    float64
D    float64
dtype: object
```

Viewing data

Use DataFrame.head() and DataFrame.tail() to view the top and bottom rows of the frame respectively:

In [32]:

```
df.head()
```

Out[32]:

	A	B	C	D
2013-01-01	-0.788755	-1.317542	0.963032	-1.938138
2013-01-02	-0.995125	2.827648	0.118524	-1.156898
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126
2013-01-04	-0.615557	0.599002	-0.791178	1.423080
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057

In [35]:

```
df.tail(3)
```

Out[35]:

	A	B	C	D
2013-01-04	-0.615557	0.599002	-0.791178	1.423080
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057
2013-01-06	-1.205508	-0.792925	0.913221	-1.263441

Display the DataFrame.index or DataFrame.columns:

In [36]:

```
df.index
```

Out[36]:

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [37]:

```
df.columns
```

Out[37]:

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

Return a NumPy representation of the underlying data with DataFrame.to_numpy() without the index or column labels:

In [38]:

```
df.to_numpy()
```

Out[38]:

```
array([[-0.7887555 , -1.31754249,  0.96303236, -1.93813802],
       [-0.99512492,  2.82764777,  0.11852408, -1.15689844],
       [-2.56535854, -0.40027113, -1.51486135, -0.51712612],
       [-0.61555737,  0.59900234, -0.79117839,  1.42308007],
       [-1.11237874, -2.75195616, -0.69791217,  0.68705682],
       [-1.20550806, -0.79292509,  0.91322097, -1.2634412 ]])
```

Note

NumPy arrays have one dtype for the entire array while pandas DataFrames have one dtype per column. When you call DataFrame.to_numpy(), pandas will find the NumPy dtype that can hold all of the dtypes in the DataFrame. If the common data type is object, DataFrame.to_numpy() will require copying data.

In [39]:

```
df2.dtypes
```

Out[39]:

```
A           float64
B     datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

In [40]:

```
df2.to_numpy()
```

Out[40]:

```
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

describe() shows a quick statistic summary of your data:

In [41]:

```
df.describe()
```

Out[41]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	-1.213781	-0.306007	-0.168196	-0.460911
std	0.696196	1.891596	1.001355	1.279145
min	-2.565359	-2.751956	-1.514861	-1.938138
25%	-1.182226	-1.186388	-0.767862	-1.236806
50%	-1.053752	-0.596598	-0.289694	-0.837012
75%	-0.840348	0.349184	0.714547	0.386011
max	-0.615557	2.827648	0.963032	1.423080

Transposing your data:

In [42]:

```
df.T
```

Out[42]:

	2013-01-01	2013-01-02	2013-01-03	2013-01-04	2013-01-05	2013-01-06
A	-0.788755	-0.995125	-2.565359	-0.615557	-1.112379	-1.205508
B	-1.317542	2.827648	-0.400271	0.599002	-2.751956	-0.792925
C	0.963032	0.118524	-1.514861	-0.791178	-0.697912	0.913221
D	-1.938138	-1.156898	-0.517126	1.423080	0.687057	-1.263441

DataFrame.sort_index() sorts by an axis:

In [43]:

```
df.sort_index(axis=1,ascending=False)
```

Out[43]:

	D	C	B	A
2013-01-01	-1.938138	0.963032	-1.317542	-0.788755
2013-01-02	-1.156898	0.118524	2.827648	-0.995125
2013-01-03	-0.517126	-1.514861	-0.400271	-2.565359
2013-01-04	1.423080	-0.791178	0.599002	-0.615557
2013-01-05	0.687057	-0.697912	-2.751956	-1.112379
2013-01-06	-1.263441	0.913221	-0.792925	-1.205508

DataFrame.sort_values() sorts by values:

In [44]:

```
df.sort_values(by="B")
```

Out[44]:

	A	B	C	D
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057
2013-01-01	-0.788755	-1.317542	0.963032	-1.938138
2013-01-06	-1.205508	-0.792925	0.913221	-1.263441
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126
2013-01-04	-0.615557	0.599002	-0.791178	1.423080
2013-01-02	-0.995125	2.827648	0.118524	-1.156898

Selection

Note

While standard Python / NumPy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, DataFrame.at(), DataFrame.iat(), DataFrame.loc() and DataFrame.iloc().

See the indexing documentation Indexing and Selecting Data and MultiIndex / Advanced Indexing.

Getitem ([])

For a DataFrame, passing a single label selects a columns and yields a Series equivalent to df.A:

In [45]:

```
df["A"]
```

Out[45]:

```
2013-01-01   -0.788755
2013-01-02   -0.995125
2013-01-03   -2.565359
2013-01-04   -0.615557
2013-01-05   -1.112379
2013-01-06   -1.205508
Freq: D, Name: A, dtype: float64
```

For a DataFrame, passing a slice : selects matching rows:

In [46]:

```
df[0:3]
```

Out[46]:

	A	B	C	D
2013-01-01	-0.788755	-1.317542	0.963032	-1.938138
2013-01-02	-0.995125	2.827648	0.118524	-1.156898
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126

In [47]:

```
df["20130102":"20130104"]
```

Out[47]:

	A	B	C	D
2013-01-02	-0.995125	2.827648	0.118524	-1.156898
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126
2013-01-04	-0.615557	0.599002	-0.791178	1.423080

Selection by label

See more in Selection by Label using DataFrame.loc() or DataFrame.at().

Selecting a row matching a label:

In [48]:

```
df.loc[dates[0]]
```

Out[48]:

```
A   -0.788755
B   -1.317542
C    0.963032
D   -1.938138
Name: 2013-01-01 00:00:00, dtype: float64
```

```
Selecting all rows (:) with a select column labels:
```

In [49]:

```
df.loc[:,["A","B"]]
```

Out[49]:

	A	B
2013-01-01	-0.788755	-1.317542
2013-01-02	-0.995125	2.827648
2013-01-03	-2.565359	-0.400271
2013-01-04	-0.615557	0.599002
2013-01-05	-1.112379	-2.751956
2013-01-06	-1.205508	-0.792925

For label slicing, both endpoints are included:

In [51]:

```
In [29]: df.loc["20130102":"20130104", ["A", "B"]]
```

Out[51]:

	A	B
2013-01-02	-0.995125	2.827648
2013-01-03	-2.565359	-0.400271
2013-01-04	-0.615557	0.599002

Selecting a single row and column label returns a scalar:

In [52]:

```
df.loc[dates[0],"A"]
```

Out[52]:

```
-0.7887554958810737
```

Selecting a single row and column label returns a scalar:

In [53]:

```
df.loc[dates[0],"A"]
```

Out[53]:

```
-0.7887554958810737
```

Selection by position

See more in Selection by Position using DataFrame.iloc() or DataFrame.iat().

In [54]:

```
df.iloc[3]
```

Out[54]:

```
A   -0.615557
B    0.599002
C   -0.791178
D    1.423080
Name: 2013-01-04 00:00:00, dtype: float64
```

Integer slices acts similar to NumPy/Python:

In [55]:

```
df.iloc[3:5, 0:2]
```

Out[55]:

	A	B
2013-01-04	-0.615557	0.599002
2013-01-05	-1.112379	-2.751956

Lists of integer position locations:

In [56]:

```
df.iloc[[1,2,4],[0,2]]
```

Out[56]:

	A	C
2013-01-02	-0.995125	0.118524
2013-01-03	-2.565359	-1.514861
2013-01-05	-1.112379	-0.697912

For slicing rows explicitly:

In [57]:

```
df.iloc[1:3,:]
```

Out[57]:

	A	B	C	D
2013-01-02	-0.995125	2.827648	0.118524	-1.156898
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126

For slicing columns explicitly:

In [58]:

```
df.iloc[:,1:3]
```

Out[58]:

	B	C
2013-01-01	-1.317542	0.963032
2013-01-02	2.827648	0.118524
2013-01-03	-0.400271	-1.514861
2013-01-04	0.599002	-0.791178
2013-01-05	-2.751956	-0.697912
2013-01-06	-0.792925	0.913221

For getting a value explicitly:

In [59]:

```
df.iloc[1,1]
```

Out[59]:

```
2.8276477735020307
```

For getting fast access to a scalar (equivalent to the prior method):

In [60]:

```
df.iat[1,1]
```

Out[60]:

```
2.8276477735020307
```

Boolean indexing

Select rows where df.A is greater than 0.

In [65]:

```
df.A[df["A"]>0]
```

Out[65]:

```
Series([], Freq: D, Name: A, dtype: float64)
```

In [63]:

```
df[df["A"] > 0]
```

Out[63]:

A	B	C	D

Selecting values from a DataFrame where a boolean condition is met:

In [64]:

```
df[df > 0]
```

Out[64]:

	A	B	C	D
2013-01-01	NaN	NaN	0.963032	NaN
2013-01-02	NaN	2.827648	0.118524	NaN
2013-01-03	NaN	NaN	NaN	NaN
2013-01-04	NaN	0.599002	NaN	1.423080
2013-01-05	NaN	NaN	NaN	0.687057
2013-01-06	NaN	NaN	0.913221	NaN

Using isin() method for filtering:

In [66]:

```
df2=df.copy()
```

In [67]:

```
df2["E"]=["one","one","two","three","four","thee"]
```

In [68]:

```
df2
```

Out[68]:

	A	B	C	D	E
2013-01-01	-0.788755	-1.317542	0.963032	-1.938138	one
2013-01-02	-0.995125	2.827648	0.118524	-1.156898	one
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126	two
2013-01-04	-0.615557	0.599002	-0.791178	1.423080	three
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057	four
2013-01-06	-1.205508	-0.792925	0.913221	-1.263441	thee

In [69]:

```
df2[df2["E"].isin(["two", "four"])]
```

Out[69]:

	A	B	C	D	E
2013-01-03	-2.565359	-0.400271	-1.514861	-0.517126	two
2013-01-05	-1.112379	-2.751956	-0.697912	0.687057	four

Setting

Setting a new column automatically aligns the data by the indexes:

In [70]:

```
s1=pd.Series([1,2,3,4,5,6],index=pd.date_range("20130102",periods=6))
```

In [71]:

```
s1
```

Out[71]:

```
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64
```

In [74]:

```
df["F"]=s1
```

Setting values by label:

In [76]:

```
df.at[dates[0],"A"]=0
```

Setting values by position:

In [77]:

```
df.iat[0,1]=0
```

Setting by assigning with a NumPy array:

In [78]:

```
df.loc[:, "D"] = np.array([5] * len(df))
```

The result of the prior setting operations:

In [79]:

```
df
```

Out[79]:

	A	B	C	D	F
2013-01-01	0.000000	0.000000	0.963032	5	NaN
2013-01-02	-0.995125	2.827648	0.118524	5	1.0
2013-01-03	-2.565359	-0.400271	-1.514861	5	2.0
2013-01-04	-0.615557	0.599002	-0.791178	5	3.0
2013-01-05	-1.112379	-2.751956	-0.697912	5	4.0
2013-01-06	-1.205508	-0.792925	0.913221	5	5.0

A where operation with setting:

In [80]:

```
df2=df.copy()
```

In [81]:

```
df2[df2>0]=-df2
```

In [82]:

```
df2
```

Out[82]:

	A	B	C	D	F
2013-01-01	0.000000	0.000000	-0.963032	-5	NaN
2013-01-02	-0.995125	-2.827648	-0.118524	-5	-1.0
2013-01-03	-2.565359	-0.400271	-1.514861	-5	-2.0
2013-01-04	-0.615557	-0.599002	-0.791178	-5	-3.0
2013-01-05	-1.112379	-2.751956	-0.697912	-5	-4.0
2013-01-06	-1.205508	-0.792925	-0.913221	-5	-5.0

Missing data

For NumPy data types, np.nan represents missing data. It is by default not included in computations. See the Missing Data section.

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data:

In [83]:

```
df1=df.reindex(index=dates[0:4],columns=list(df.columns)+["E"])
```

In [84]:

```
df1.loc[dates[0]:dates[1],"E"]=1
df1
```

Out[84]:

	A	B	C	D	F	E
2013-01-01	0.000000	0.000000	0.963032	5	NaN	1.0
2013-01-02	-0.995125	2.827648	0.118524	5	1.0	1.0
2013-01-03	-2.565359	-0.400271	-1.514861	5	2.0	NaN
2013-01-04	-0.615557	0.599002	-0.791178	5	3.0	NaN

DataFrame.dropna() drops any rows that have missing data:

In [85]:

```
df1.dropna(how="any")
```

Out[85]:

	A	B	C	D	F	E
2013-01-02	-0.995125	2.827648	0.118524	5	1.0	1.0

DataFrame.fillna() fills missing data:

In [87]:

```
df1.fillna(value=5)
```

Out[87]:

	A	B	C	D	F	E
2013-01-01	0.000000	0.000000	0.963032	5	5.0	1.0
2013-01-02	-0.995125	2.827648	0.118524	5	1.0	1.0
2013-01-03	-2.565359	-0.400271	-1.514861	5	2.0	5.0
2013-01-04	-0.615557	0.599002	-0.791178	5	3.0	5.0

isna() gets the boolean mask where values are nan:

In [88]:

```
pd.isna(df1)
```

Out[88]:

	A	B	C	D	F	E
2013-01-01	False	False	False	False	True	False
2013-01-02	False	False	False	False	False	False
2013-01-03	False	False	False	False	False	True
2013-01-04	False	False	False	False	False	True

Operations

See the Basic section on Binary Ops.

Stats

Operations in general exclude missing data.

Calculate the mean value for each column

In [89]:

```
df.mean()
```

Out[89]:

```
A   -1.082321
B   -0.086417
C   -0.168196
D    5.000000
F    3.000000
dtype: float64
```

Calculate the mean value for each row:

In [90]:

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

Out[90]:

```
2013-01-01    1.490758
2013-01-02    1.590209
2013-01-03    0.503902
2013-01-04    1.438453
2013-01-05    0.887551
2013-01-06    1.782958
Freq: D, dtype: float64
```

Operating with another Series or DataFrame with a different index or column will align the result with the union of the index or column labels. In addition, pandas automatically broadcasts along the specified dimension and will fill unaligned labels with np.nan.

In [92]:

```
s=pd.Series([1,3,5,np.nan,6,8],index=dates).shift(2)
```

In [93]:

```
s
```

Out[93]:

```
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
Freq: D, dtype: float64
```

In [94]:

```
df.sub(s, axis="index")
```

Out[94]:

	A	B	C	D	F
2013-01-01	NaN	NaN	NaN	NaN	NaN
2013-01-02	NaN	NaN	NaN	NaN	NaN
2013-01-03	-3.565359	-1.400271	-2.514861	4.0	1.0
2013-01-04	-3.615557	-2.400998	-3.791178	2.0	0.0
2013-01-05	-6.112379	-7.751956	-5.697912	0.0	-1.0
2013-01-06	NaN	NaN	NaN	NaN	NaN

User defined functions

DataFrame.agg() and DataFrame.transform() applies a user defined function that reduces or broadcasts its result respectively.

In [95]:

```
df.agg(lambda x: np.mean(x)*5.6)
```

Out[95]:

```
A    -6.060999
B    -0.483935
C    -0.941896
D    28.000000
F    16.800000
dtype: float64
```

In [96]:

```
df.transform(lambda x:x*101.2)
```

Out[96]:

	A	B	C	D	F
2013-01-01	0.000000	0.000000	97.458874	506.0	NaN
2013-01-02	-100.706642	286.157955	11.994636	506.0	101.2
2013-01-03	-259.614284	-40.507438	-153.303969	506.0	202.4
2013-01-04	-62.294406	60.619036	-80.067253	506.0	303.6
2013-01-05	-112.572728	-278.497964	-70.628712	506.0	404.8
2013-01-06	-121.997416	-80.244019	92.417962	506.0	506.0

Value Counts

See more at Histogramming and Discretization.

In [97]:

```
s=pd.Series(np.random.randint(0,7,size=10))
```

In [98]:

```
s
```

Out[98]:

```
0    4
1    0
2    5
3    0
4    1
5    4
6    4
7    5
8    5
9    4
dtype: int32
```

In [99]:

```
s.value_counts()
```

Out[99]:

```
4    4
5    3
0    2
1    1
dtype: int64
```

String Methods

Series is equipped with a set of string processing methods in the str attribute that make it easy to operate on each element of the array, as in the code snippet below. See more at Vectorized String Methods.

In [101]:

```
s=pd.Series(["A","B","C","Aaba","Baca",np.nan,"CABA","dog","cat"])
s.str.lower()
```

Out[101]:

```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

Merge

Concat

pandas provides various facilities for easily combining together Series` and DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the Merging section.

Concatenating pandas objects together row-wise with concat():

In [104]:

```
df=pd.DataFrame(np.random.randn(10,4))
```

In [105]:

```
df
```

Out[105]:

	0	1	2	3
0	-0.666612	0.326692	0.711639	0.332411
1	-0.302767	-0.319118	-1.071033	0.843598
2	-0.009899	-0.729487	-0.719703	-0.295934
3	0.016593	-1.370889	0.555931	0.070138
4	-1.564163	-0.114751	-0.686397	-0.941804
5	0.780575	-0.801795	0.271547	1.239176
6	0.531233	0.496186	0.661872	0.206256
7	-0.509461	1.301223	0.538260	0.916769
8	0.794850	1.982715	-0.240516	0.723906
9	-0.645806	-1.847865	-0.382601	0.073680

In [106]:

```
pieces=[df[:3],df[3:7],df[7:1]]
pd.concat(pieces)
```

Out[106]:

	0	1	2	3
0	-0.666612	0.326692	0.711639	0.332411
1	-0.302767	-0.319118	-1.071033	0.843598
2	-0.009899	-0.729487	-0.719703	-0.295934
3	0.016593	-1.370889	0.555931	0.070138
4	-1.564163	-0.114751	-0.686397	-0.941804
5	0.780575	-0.801795	0.271547	1.239176
6	0.531233	0.496186	0.661872	0.206256

Note

Adding a column to a DataFrame is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the DataFrame constructor instead of building a DataFrame by iteratively appending records to it.

Join

merge() enables SQL style join types along specific columns. See the Database style joining section.

In [107]:

```
left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})

right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})
```

In [108]:

```
left
```

Out[108]:

	key	lval
0	foo	1
1	foo	2

In [109]:

```
right
```

Out[109]:

	key	rval
0	foo	4
1	foo	5

In [111]:

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

Out[111]:

	key	lval	rval
0	foo	1	4
1	foo	1	5
2	foo	2	4
3	foo	2	5

merge() on unique keys:

In [113]:

```
left = pd.DataFrame({"key": ["foo", "bar"], "lval": [1, 2]})

right = pd.DataFrame({"key": ["foo", "bar"], "rval": [4, 5]})
```

In [114]:

```
left
```

Out[114]:

	key	lval
0	foo	1
1	bar	2

In [115]:

```
right
```

Out[115]:

	key	rval
0	foo	4
1	bar	5

In [116]:

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

Out[116]:

	key	lval	rval
0	foo	1	4
1	bar	2	5

Grouping

By “group by” we are referring to a process involving one or more of the following steps:

Splitting the data into groups based on some criteria

Applying a function to each group independently

Combining the results into a data structure

See the Grouping section.

In [117]:

```
df = pd.DataFrame(
    {
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
    }
)
```

In [118]:

```
df
```

Out[118]:

	A	B	C	D
0	foo	one	-1.344861	-0.627419
1	bar	one	-2.037580	0.902300
2	foo	two	-2.377953	-0.124501
3	bar	three	-0.996630	0.485040
4	foo	two	-0.603325	1.026214
5	bar	two	0.936329	-1.008643
6	foo	one	-0.369022	-0.544231
7	foo	three	1.481801	-1.689892

Grouping by a column label, selecting column labels, and then applying the sum() function to the resulting groups:

In [120]:

```
df.groupby("A")[["C","D"]].sum()
```

Out[120]:

	C	D
A		
bar	-2.097881	0.378697
foo	-3.213361	-1.959829

Grouping by multiple columns label forms MultiIndex.

In [122]:

```
df.groupby(["A","B"]).sum()
```

Out[122]:

A	B	C	D
bar	one	-2.037580	0.902300
	three	-0.996630	0.485040
	two	0.936329	-1.008643
foo	one	-1.713883	-1.171650
	three	1.481801	-1.689892
	two	-2.981278	0.901713

Reshaping

See the sections on Hierarchical Indexing and Reshaping.

Stack

In [126]:

```
arrays = [
   ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ["one", "two", "one", "two", "one", "two", "one", "two"],
]


index = pd.MultiIndex.from_arrays(arrays, names=["first", "second"])

df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])

df2 = df[:4]
df2
```

Out[126]:

first	second	A	B
bar	one	0.501710	0.286028
	two	-0.464888	-0.761557
baz	one	-0.632288	-0.205617
	two	0.110614	0.589187

Pivot tables

In [130]:

```
df = pd.DataFrame(
    {
        "A": ["one", "one", "two", "three"] * 3,
        "B": ["A", "B", "C"] * 4,
        "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 2,
        "D": np.random.randn(12),
        "E": np.random.randn(12),
    }
)


df
```

Out[130]:

	A	B	C	D	E
0	one	A	foo	-0.297625	-0.805798
1	one	B	foo	-1.401747	-1.208780
2	two	C	foo	0.210532	1.067108
3	three	A	bar	-1.062858	-0.208819
4	one	B	bar	-0.716851	-0.247451
5	one	C	bar	0.076298	-0.819127
6	two	A	foo	-0.844794	1.757198
7	three	B	foo	1.664711	-1.209800
8	one	C	foo	0.076802	0.488167
9	one	A	bar	-0.420130	-0.234488
10	two	B	bar	0.974855	0.374892
11	three	C	bar	-1.077588	-1.735761

pivot_table() pivots a DataFrame specifying the values, index and columns

In [131]:

```
pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
```

Out[131]:

	C	bar	foo
A	B		
one	A	-0.420130	-0.297625
	B	-0.716851	-1.401747
	C	0.076298	0.076802
three	A	-1.062858	NaN
	B	NaN	1.664711
	C	-1.077588	NaN
two	A	NaN	-0.844794
	B	0.974855	NaN
	C	NaN	0.210532

Time series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the Time Series section.

In [132]:

```
rng = pd.date_range("1/1/2012", periods=100, freq="S")

ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

ts.resample("5Min").sum()
```

Out[132]:

```
2012-01-01    25437
Freq: 5T, dtype: int32
```

Series.tz_localize() localizes a time series to a time zone:

In [133]:

```
rng = pd.date_range("3/6/2012 00:00", periods=5, freq="D")

ts = pd.Series(np.random.randn(len(rng)), rng)

ts
```

Out[133]:

```
2012-03-06    0.548801
2012-03-07    1.952777
2012-03-08    0.235469
2012-03-09    0.767419
2012-03-10    0.589082
Freq: D, dtype: float64
```

In [134]:

```

ts_utc = ts.tz_localize("UTC")
ts_utc
```

Out[134]:

```
2012-03-06 00:00:00+00:00    0.548801
2012-03-07 00:00:00+00:00    1.952777
2012-03-08 00:00:00+00:00    0.235469
2012-03-09 00:00:00+00:00    0.767419
2012-03-10 00:00:00+00:00    0.589082
Freq: D, dtype: float64
```

Series.tz_convert() converts a timezones aware time series to another time zone:

In [135]:

```
ts_utc.tz_convert("US/Eastern")
```

Out[135]:

```
2012-03-05 19:00:00-05:00    0.548801
2012-03-06 19:00:00-05:00    1.952777
2012-03-07 19:00:00-05:00    0.235469
2012-03-08 19:00:00-05:00    0.767419
2012-03-09 19:00:00-05:00    0.589082
Freq: D, dtype: float64
```

Adding a non-fixed duration (BusinessDay) to a time series:

In [136]:

```
rng
```

Out[136]:

```
DatetimeIndex(['2012-03-06', '2012-03-07', '2012-03-08', '2012-03-09',
               '2012-03-10'],
              dtype='datetime64[ns]', freq='D')
```

In [137]:

```
rng+pd.offsets.BusinessDay(5)
```

Out[137]:

```
DatetimeIndex(['2012-03-13', '2012-03-14', '2012-03-15', '2012-03-16',
               '2012-03-16'],
              dtype='datetime64[ns]', freq=None)
```

Categoricals

pandas can include categorical data in a DataFrame. For full docs, see the categorical introduction and the API documentation.

In [138]:

```
df = pd.DataFrame(
    {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
)
```

Converting the raw grades to a categorical data type:

In [140]:

```
df["grade"] = df["raw_grade"].astype("category")

df["grade"]
```

Out[140]:

```
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

Rename the categories to more meaningful names:

In [141]:

```
new_categories = ["very good", "good", "very bad"]

df["grade"] = df["grade"].cat.rename_categories(new_categories)
```

In [143]:

```
df["grade"] = df["grade"].cat.set_categories(
    ["very bad", "bad", "medium", "good", "very good"])
```

In [144]:

```
df["grade"]
```

Out[144]:

```
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

Sorting is per order in the categories, not lexical order:

In [145]:

```
df.sort_values(by="grade")
```

Out[145]:

	id	raw_grade	grade
5	6	e	very bad
1	2	b	good
2	3	b	good
0	1	a	very good
3	4	a	very good
4	5	a	very good

In [146]:

```
df.groupby("grade", observed=False).size()
```

Out[146]:

```
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

Plotting

See the Plotting docs.

We use the standard convention for referencing the matplotlib API:

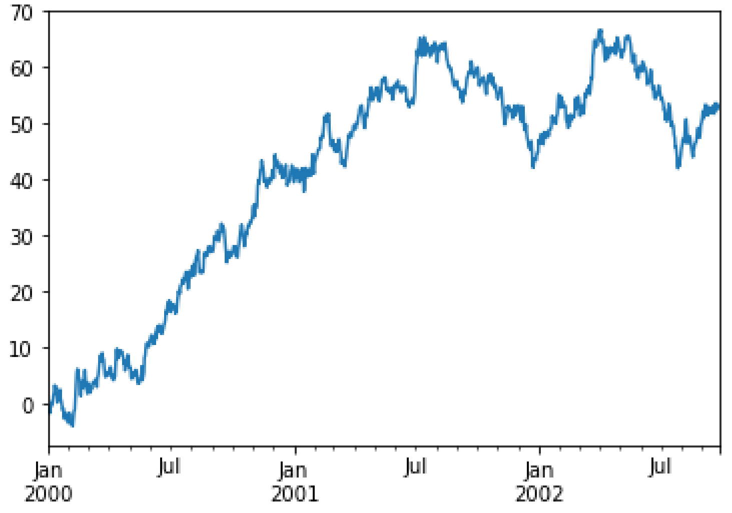
In [147]:

```
import matplotlib.pyplot as plt

plt.close("all")
```

The plt.close method is used to close a figure window:

In [148]:

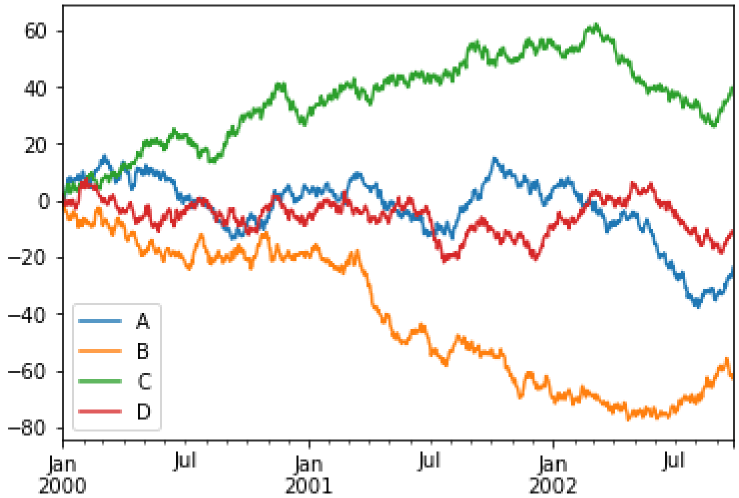
```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

ts = ts.cumsum()

ts.plot();
```



In [149]:

```
df = pd.DataFrame(
    np.random.randn(1000, 4), index=ts.index, columns=["A", "B", "C", "D"]
)


df = df.cumsum()

plt.figure();

df.plot();

plt.legend(loc='best');
```

```
<Figure size 432x288 with 0 Axes>
```



Gotchas

If you are attempting to perform a boolean operation on a Series or DataFrame you might see an exception like:

In [151]:

```
if pd.Series([False, True, False]):
    print("I was true")
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Input In [151], in <cell line: 1>()
----> 1 if pd.Series([False, True, False]):
      2     print("I was true")

File ~\anaconda3\lib\site-packages\pandas\core\generic.py:1527, in NDFram
e.__nonzero__(self)
   1525 @final
   1526 def __nonzero__(self):
-> 1527     raise ValueError(
   1528         f"The truth value of a {type(self).__name__} is ambiguous.
"
   1529         "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
   1530     )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool
(), a.item(), a.any() or a.all().
```

In [ ]: