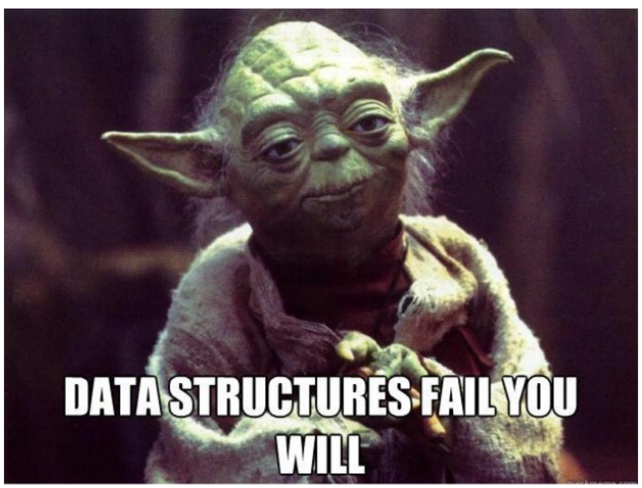
Data Structures in Python

October 2, 2017



pandas

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

What is a data structure?

- Way to store data and have some method to retrieve and manipulate it

- Lots of examples in python:
  - List, dict, tuple, set, string
  - Array
  - Series, DataFrame

- Some of these are “built-in” (meaning you can just use them), others are contained within other python packages, like numpy and pandas

Basic Python Data Structures (built-in)

- List, dict, tuple, set, string
- Each of these can be accessed in a variety of ways
- Decision on which to use? Depends on what sort of features you need (easy indexing, immutability, etc)
    - Mutable vs immutable
        - Mutable – can change
        - Immutable – doesn't change

```
x = something # immutable type
print x
func(x)
print x # prints the same thing

x = something # mutable type
print x
func(x)
print x # might print something different
```

Basic Structure: List

- Very versatile, can have items of different types, **is mutable**

- To create: use square brackets [] to contain comma separated values

- Example: >> l = [‘a’, ‘b’, 123]
  - >> l

  [’a’, ‘b’, 123]

- To get values out: >> l[1]   (use index, starts with 0)

  >> b

- We saw these back in lab 3

Basic Structure: Set

- Set is an unordered collection with no duplicate values, **is mutable**
- Create using {}
- Example: >> s = {1, 2, 3}
  - >> s

    set([1,2,3])

- Useful for eliminating duplicate values from a list, doing operations like intersection, difference, union

Basic Structure: Tuple

- Tuple holds values separated by commas, are **immutable**
- Create using , or () to create empty
- Example: >> t = 1,2,3
  - >> t
    (1,2,3)
    >> type(t)
    type ‘tuple’

- Useful when storing data that does not change, when needing to optimize performance of code (python knows how much memory needed)

Basic Structure: Dict

- Represented by key:value pair
  - Keys: can by any immutable type and unique
  - Values: can be any type (mutable or immutable)

- To create: use curly braces {} or dict() and list both key and value
- >>> letters = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
  >>> type(letters)
  <type 'dict'>
- To access data in dictionary, call by the key
  - >>> letters[2]
    'b'
- Have useful methods like keys(),values(),iteritems(),itervalues() useful for accessing dictionary entries

- Useful when:
  - Need association between key:value pair
  - Need to quickly look up data based on a defined key
  - Values are modified

Array: Use NumPy!

- What is an array?
  - "list of lists"
  - Similar to Matlab in some ways
    - Create a 2x3 array
      - [1 2 3; 4 5 6] : matlab
      - np.array([[1.,2.,3.],[4.,5.,6.]])

```
>>> import numpy as np
>>>  y  =  np.array([[1.,2.,3.],  [4.,5.,6.]])
>>>  y
array([[  1.,    2.,    3.],
        [  4.,    5.,    6.]])
>>>
```

- What is NumPy?
  - Numerical Python
  - Python library very useful for scientific computing

- How to access NumPy?
  - Need to import it into your python workspace or into your script

  - >> import numpy as np

Why use a NumPy array?

- What is it?
  - “multidimensional array of objects of **all the same type**”

- More compact for than list (don’t need to store both value and type like in a list)
- Reading/writing faster with NumPy
- Get a lot of vector and matrix operations
  - Can’t do “vectorized” operations on list (like element-wise addition, multiplication)

- Can also do the standard stuff, like indexing, comparisons, logical operations

Creating NumPy Arrays

```
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

Creating NumPy array and checking if each element is > 3

```
>>> b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
>>> print(b.shape)                     # Prints "(2, 3)"
(2, 3)
>>> print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
(1, 2, 4)
```

Create NumPy array, print out array dimensions, and use indexing tools

```
>>> c = np.zeros((2,2))
>>> print(c)
[[ 0.  0.]
 [ 0.  0.]]
```

Create 2x2 NumPy array with just zeros

More Creating NumPy Arrays

- arange: like "range", returns an ndarray

```
>>> a = np.arange(6)
>>> print(a)
[0 1 2 3 4 5]
```

- Use reshape to define/change shape of array

```
>>> b = np.arange(12).reshape(4,3)
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Operations with NumPy Arrays

- Arithmetic operations (e.g. +, -, *, /, **) with scalars and between equal-size arrays – done element by element
  - A new array is created with the result

```
>>> b = np.arange(12).reshape(4,3)
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> c = b + 5
>>> print(c)
[[ 5  6  7]
 [ 8  9 10]
 [11 12 13]
 [14 15 16]]
```

- Universal functions (for example: sin, cos, exp) also operate elementwise on an array, new array results

Be careful: * vs dot

- * is product operator, operates elementwise in NumPy arrays

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A*B
array([[2, 0],
       [0, 4]])
>>> np.dot(A,B)
array([[5, 4],
       [3, 4]])
```

A*B – elementwise multiplication

.dot – matrix product

Other Useful NumPy Array Operations

- Sum, min, max: can be used to get values for all elements in array

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.30541447,  0.64099062,  0.05487081],
       [ 0.9990191 ,  0.05537393,  0.38775904]])
>>> a.sum()
2.4434279566031463
>>> a.min()
0.05487080789064569
>>> a.max()
0.99901909872534389
```

Get sum of all elements in array, also min and max within array

- Can use (axis=#) to specify certain rows and columns

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b.sum(axis=0)
array([12, 15, 18, 21])
>>> b.min(axis=1)
array([0, 4, 8])
>>> b.cumsum(axis=1)
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Sum of each column (axis=0)

Min of each row (axis = 1)

Cumulative sum along each row

Indexing with NumPy Arrays

- 1D arrays (just like lists)

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
```

Create array using arange

Pull out element at position 3

Pull out elements in positions starting at 3, before 6

- Multidimensional arrays: work with an index per axis

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Element at row 3, column 4

Each row in 2nd column

Each row in 2nd column

Each column in 2nd and 3rd row

What is pandas?

panel **da**ta: multidimensional structured datasets

- Open source package with user friendly data structures and data analysis tools for Python
  - Built on top of NumPy, gives more tools

- Very useful for tabular data in columns (i.e. spreadsheets), time series data, matrix data, etc

- Two main data structures:
  - Series (1-dimensional)
  - DataFrame (2-dimensional)

- How to access:
  - Need to import it into your python workspace or into your script

```
>> import pandas as pd
```

Pandas: Series

- Effectively a 1-D NumPy array with an index
- 1D labeled array that can hold any data type, with labels known as the “index”

```
>>> import pandas as pd
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>> s
a   -0.896461
b   -0.268122
c   -1.097631
d   -2.069645
e   -0.289530
dtype: float64
```

data can be an array, scalar, or a dict

```
>>>
>>> s = {'a': 2., 'b': 4., 'c': 6., 'd': 8.}
>>> pd.Series(s)
a    2.0
b    4.0
c    6.0
d    8.0
dtype: float64
```

Pandas: Series

```
>>>
>>> s = {'a': 2., 'b': 4., 'c': 6., 'd': 8.}
>>> pd.Series(s)
a    2.0
b    4.0
c    6.0
d    8.0
dtype: float64
```

- Can using slicing to grab out values

```
>>> sd = pd.Series(s)
>>> sd[1]
4.0
```

- Can also use index to grab out values

```
>>> sd['b']
4.0
```

```
>>> sd
a    2.0
b    4.0
c    6.0
d    8.0
dtype: float64
>>> type(sd)
<class 'pandas.core.series.Series'>
```

Pandas: DataFrame

- Most commonly used pandas object
- DataFrame is basically a table made up of named columns of series
  - Think spreadsheet or table of some kind
  - Can take data from
    - Dict of 1D arrays, lists, dicts, Series
    - 2D numpy array
    - Series
    - Another DataFrame
  - Can also define index (row labels) and columns (column labels)
  - Series can be dynamically added to or removed from the DataFrame

Creating DataFrames

- From dict of Series or dicts:

Have 2 series (one and two)

```
>>>
>>> d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
...      'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
>>>
>>> df = pd.DataFrame(d)
>>> df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

New DataFrame (df) is union of the 2 Series indices

Output includes row labels (index) and column labels as specified

Note the NaN reported because of no 4th value in "one"

Using arrays/lists is similar:

```
>>> d = {'one' : [1., 2., 3., 4.],
...      'two' : [4., 3., 2., 1.]}
>>> pd.DataFrame(d)
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
```

If no index is given, index will be range(n) where n is array length

Accessing DataFrame Info

```
>>> df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
>>> pd.DataFrame(d, index=['d', 'b', 'a'])
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
>>> pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

Can access specific rows

Can access specific rows and columns

```
>>> df['two']
a    1.0
b    2.0
c    3.0
d    4.0
Name: two, dtype: float64
```

Grab specific column from existing DataFrame

Accessing DataFrame Info

```
>>> df['two']
a    1.0
b    2.0
c    3.0
d    4.0
Name: two, dtype: float64
```

Grab specific column from existing DataFrame

```
>>> df['three'] = df['one'] * df['two']
>>> df
   one  two  three
a  1.0  1.0    1.0
b  2.0  2.0    4.0
c  3.0  3.0    9.0
d  NaN  4.0    NaN
```

Make a new column through operations on others

```
>>> del df['two']
>>> df
   one  three
a  1.0    1.0
b  2.0    4.0
c  3.0    9.0
d  NaN    NaN
```

Get rid of columns

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	df[col]	Series
Select row by label	df.loc[label]	Series
Select row by integer location	df.iloc[loc]	Series
Slice rows	df[5:10]	DataFrame
Select rows by boolean vector	df[bool_vec]	DataFrame

Working with DataFrames

```
>>> df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
>>> df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
>>> df
          A         B         C         D
0  0.925605 -0.101996 -0.856505  0.041030
1  0.683239 -0.666745 -0.746350 -0.765129
2 -1.149823  0.256815 -1.844288 -0.182581
3  1.283709 -0.421631 -0.439489  1.059241
4 -0.072631  0.604832  0.033367 -0.484844
5  1.429966 -0.863785  1.076024 -2.696585
6 -1.175667  0.911034  0.156114  0.054385
7 -1.452214  1.480580 -1.615911  0.298720
8 -1.630938  0.176167  0.269268 -0.822671
9  1.782930 -0.029016 -0.268635  0.623897
>>> df2
          A         B         C
0  0.104096 -0.689956 -0.256331
1  0.387243 -0.663118 -1.149259
2 -1.754454  0.859184 -0.420377
3  0.333610 -0.549891  1.220547
4 -0.632296  0.571345  0.256852
5  1.092259 -0.420193  0.311140
6  1.036973 -0.387453 -0.498542
>>> df + df2
          A         B         C   D
0  1.029701 -0.791951 -1.112835 NaN
1  1.070481 -1.329862 -1.895610 NaN
2 -2.904277  1.115999 -2.264664 NaN
3  1.617319 -0.971522  0.781058 NaN
4 -0.704926  1.176177  0.290219 NaN
5  2.522225 -1.283977  1.387164 NaN
6 -0.138694  0.523581 -0.342428 NaN
7       NaN       NaN       NaN NaN
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

Create 2 different DataFrames

Add the dataframes together

Note elementwise addition, with the result having the union of row and column labels, even if you don't have values in each position

Lots of NumPy elementwise functions work on DataFrames, as do operations like transpose (.T), .dot

Other cool things to do with DataFrames

Basic statistics

```
>>> df.describe()
               A          B          C          D
count  10.000000  10.000000  10.000000  10.000000
mean    0.062418   0.134626  -0.423640  -0.287454
std     1.318020   0.721369   0.883781   1.031773
min    -1.630938  -0.863785  -1.844288  -2.696585
25%    -1.169206  -0.341722  -0.828966  -0.695058
50%     0.305304   0.073575  -0.354062  -0.070775
75%     1.194183   0.517827   0.125428   0.237637
max     1.782930   1.480580   1.076024   1.059241
```

sorting

```
>>> df
          A         B         C         D
0  0.925605 -0.101996 -0.856505  0.041030
1  0.683239 -0.666745 -0.746350 -0.765129
2 -1.149823  0.256815 -1.844288 -0.182581
3  1.283709 -0.421631 -0.439489  1.059241
4 -0.072631  0.604832  0.033367 -0.484844
5  1.429966 -0.863785  1.076024 -2.696585
6 -1.175667  0.911034  0.156114  0.054385
7 -1.452214  1.480580 -1.615911  0.298720
8 -1.630938  0.176167  0.269268 -0.822671
9  1.782930 -0.029016 -0.268635  0.623897
>>> df.sort_values(by='B')
          A         B         C         D
5  1.429966 -0.863785  1.076024 -2.696585
1  0.683239 -0.666745 -0.746350 -0.765129
3  1.283709 -0.421631 -0.439489  1.059241
0  0.925605 -0.101996 -0.856505  0.041030
9  1.782930 -0.029016 -0.268635  0.623897
8 -1.630938  0.176167  0.269268 -0.822671
2 -1.149823  0.256815 -1.844288 -0.182581
4 -0.072631  0.604832  0.033367 -0.484844
6 -1.175667  0.911034  0.156114  0.054385
7 -1.452214  1.480580 -1.615911  0.298720
...
```

Other cool things to do with DataFrames

```
>>> df[df.A > 0]
          A         B         C         D
0  0.925605 -0.101996 -0.856505  0.041030
1  0.683239 -0.666745 -0.746350 -0.765129
3  1.283709 -0.421631 -0.439489  1.059241
5  1.429966 -0.863785  1.076024 -2.696585
9  1.782930 -0.029016 -0.268635  0.623897
```

Grabbing data that meet a certain condition

```
>>> df2 = df.copy()
>>> df2['E'] = ['one','one','two','three','four','five','four','one','two','four']
>>> df2
          A         B         C         D      E
0  0.925605 -0.101996 -0.856505  0.041030    one
1  0.683239 -0.666745 -0.746350 -0.765129    one
2 -1.149823  0.256815 -1.844288 -0.182581    two
3  1.283709 -0.421631 -0.439489  1.059241  three
4 -0.072631  0.604832  0.033367 -0.484844   four
5  1.429966 -0.863785  1.076024 -2.696585   five
6 -1.175667  0.911034  0.156114  0.054385   four
7 -1.452214  1.480580 -1.615911  0.298720    one
8 -1.630938  0.176167  0.269268 -0.822671    two
9  1.782930 -0.029016 -0.268635  0.623897   four
```

Add a new column at end of dataframe

```
>>> df2[df2['E'].isin(['two','four'])]
          A         B         C         D     E
2 -1.149823  0.256815 -1.844288 -0.182581   two
4 -0.072631  0.604832  0.033367 -0.484844  four
6 -1.175667  0.911034  0.156114  0.054385  four
8 -1.630938  0.176167  0.269268 -0.822671   two
9  1.782930 -0.029016 -0.268635  0.623897  four
```

Filtering data to grab only data that contains certain values using .isin

DataFrames: groupby

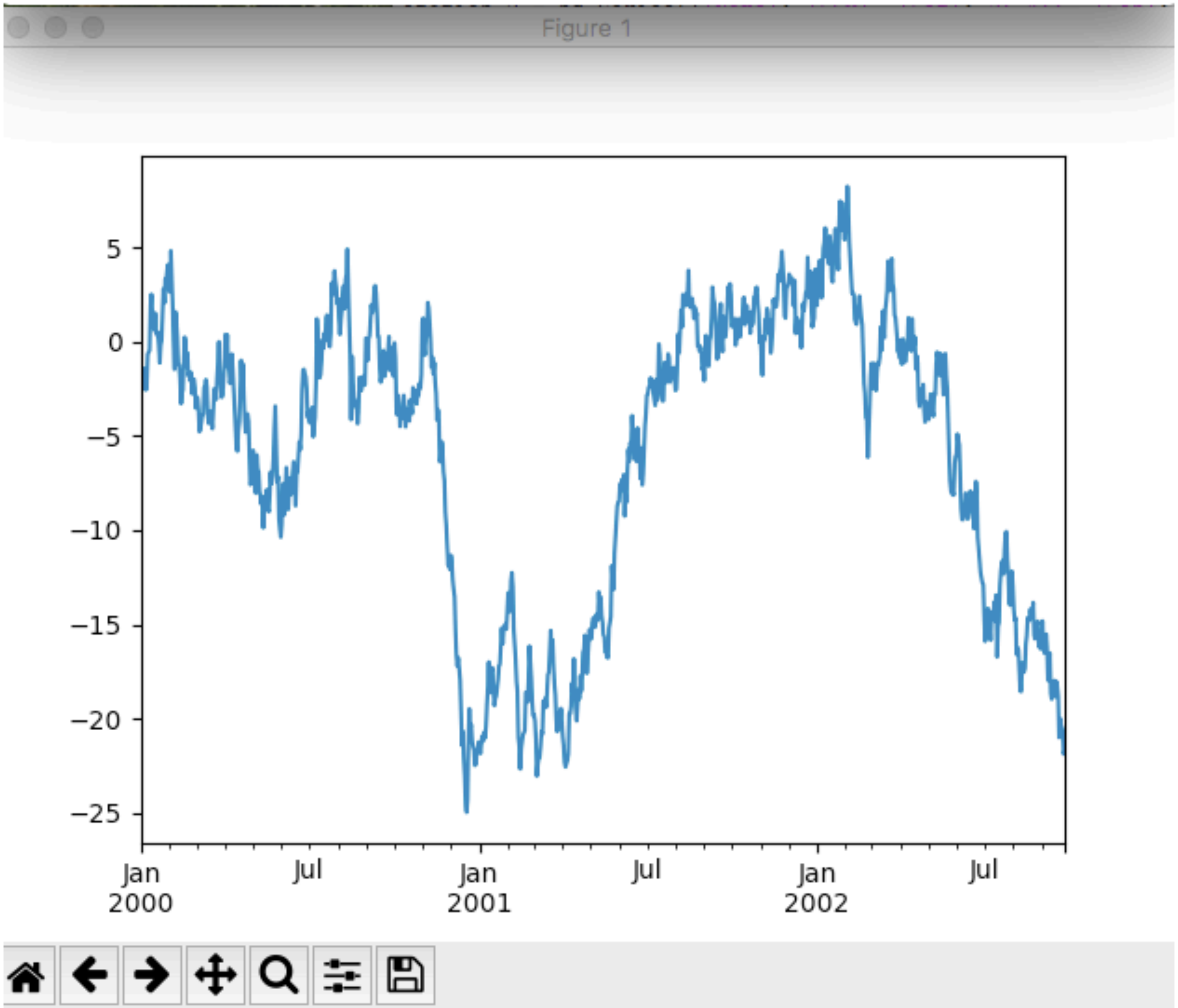
- This allows you to split up data into groups based on some criteria, apply some function, and get a result

```
>>> df2
          A         B         C         D      E
0  0.925605 -0.101996 -0.856505  0.041030    one
1  0.683239 -0.666745 -0.746350 -0.765129    one
2 -1.149823  0.256815 -1.844288 -0.182581    two
3  1.283709 -0.421631 -0.439489  1.059241  three
4 -0.072631  0.604832  0.033367 -0.484844   four
5  1.429966 -0.863785  1.076024 -2.696585   five
6 -1.175667  0.911034  0.156114  0.054385   four
7 -1.452214  1.480580 -1.615911  0.298720    one
8 -1.630938  0.176167  0.269268 -0.822671    two
9  1.782930 -0.029016 -0.268635  0.623897   four
>>> df2.groupby('E')
<pandas.core.groupby.DataFrameGroupBy object at 0x10c03bf50>
>>> df2.groupby('E').sum()
              A         B         C         D
E
five   1.429966 -0.863785  1.076024 -2.696585
four   0.534633  1.486849 -0.079153  0.193438
one    0.156630  0.711840 -3.218765 -0.425378
three  1.283709 -0.421631 -0.439489  1.059241
two   -2.780760  0.432982 -1.575020 -1.005252
```

Using “groupby” to select rows that contain same value in E, then sum those values

Plotting Data in Series

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',periods=1000))
>>> ts = ts.cumsum()
>>> ts.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x10f74d690>
>>> plt.show()
```



Created a series of 1000 random numbers, with an index of dates starting at 1/1/2000

Plotted the cumulative sum of those random numbers

Plotting Data in DataFrames

```
>>> 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'); plt.show()
<matplotlib.figure.Figure object at 0x111c9e990>
<matplotlib.axes._subplots.AxesSubplot object at 0x111c6fc10>
<matplotlib.legend.Legend object at 0x111c93690>
```

Using .plot() with DataFrames will plot all of the columns with labels

Next up:

- Lab today – working with data structures
- Next week: how to get data into and out of python (I/O topics)