Applied Data Science with Python

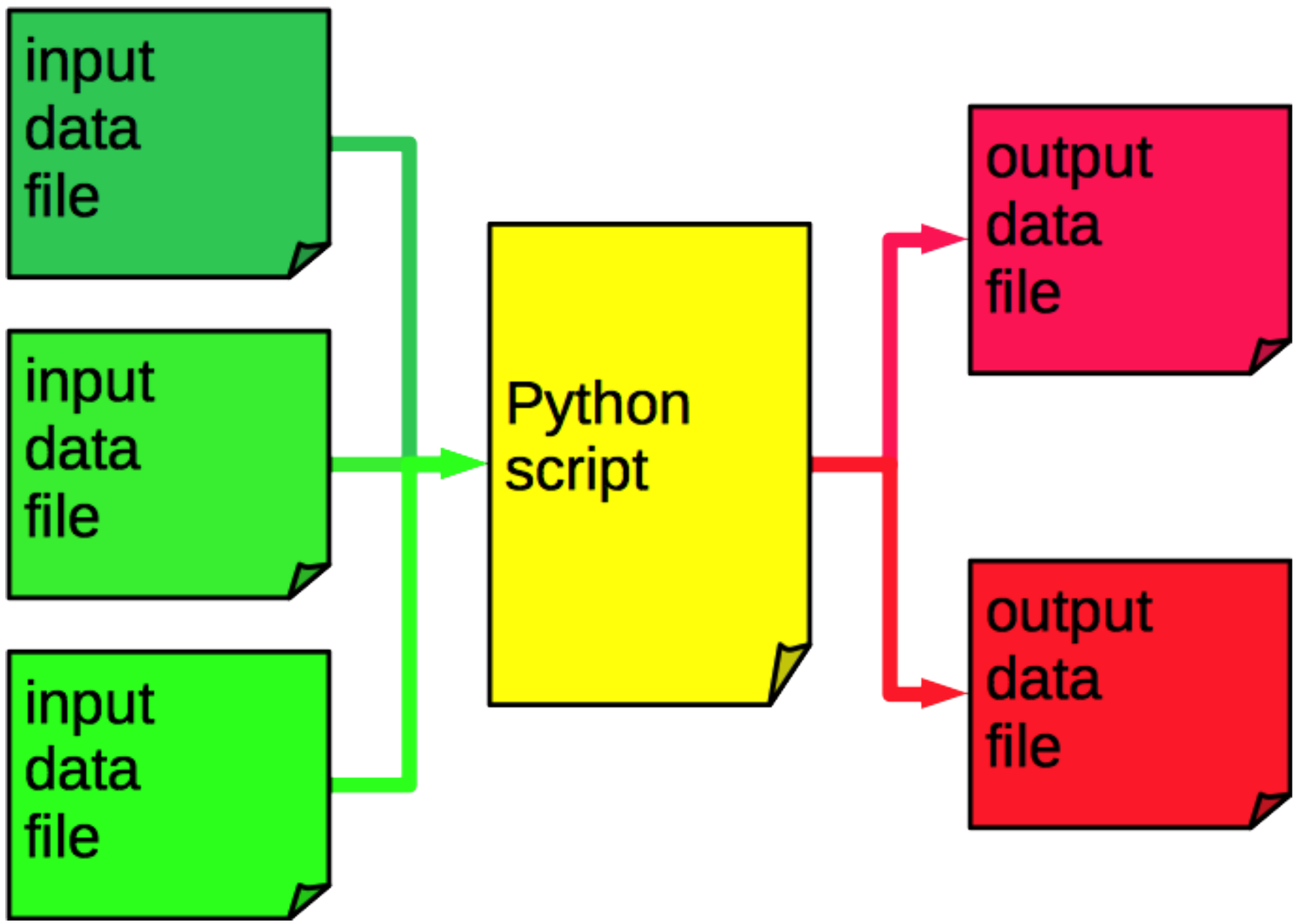
Kamini Garg
kaminigarg32@gmail.com

Lead Data Scientist, UPC Switzerland

This lecture will teach .....

- Files basic
- Numpy & Arrays
- Pandas

Accessing multiple files



File Reading

```
line one\n
line two\n
line three\n
line four\n
```

data.txt

```
f = open('data.txt', 'r')
for line in f:
   print(line)
f.close()
```

Print each file of the file

```
f.readline() -> only read a line

f.readlines()-> read all lines and
store them in a list
```

File Reading

```
line one\n
line two\n
line three\n
line four\n
```

data.txt

```
with open('data.txt', 'r')as f:
    for line in f:
        print(line)
```

The “with” statement closes the file automatically. A more Pythonic implementation.

File Writing

```
line one\n
line two\n
line three\n
line four\n
```

data.txt

```
words = [‘line one’,‘line
two’,‘line three’,‘line four’]

with open(’data.txt’, ’w’)as f:

    for word in words:

        f.write(word+”\n”)
```

Write permission with ‘w’ when opening the file

Python csv module

- Splitting of string from text files can be tedious process sometimes.
- Python has special module for comma separated files or CSV files.

Reading a csv file:

```
import csv
with open('some.csv', 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            print(row)
```

Python csv module

Writing a csv file:

```
import csv
input_file = open('test.csv',"r")
reader = csv.reader(input_file)
output_file = open('test_write.csv',"w")
writer = csv.writer(output_file ,
delimiter=' ',quotechar="")
for row in reader:
   writer.writerow(row)
input_file.close()
output_file.close()
```

Numpy

- Core library for scientific computing in Python.
- Provides high-performance multidimensional array object and tools for working with these arrays.
- Widely used in academia, finance and industry.
- Mature, fast, stable and under continuous development.

More information: http://docs.scipy.org/doc/numpy/reference/

Numpy Installation

Install NumPy with the Anaconda Prompt

To install NumPy, open the **Anaconda Prompt** and type:

```
> conda install numpy
```

Type `y` for yes when prompted.

Install NumPy with pip

To install NumPy with **pip**, bring up a terminal window and type:

```
$ pip install numpy
```

This command installs NumPy in the current working Python environment.

Numpy Arrays

- Lists OK for storing small amounts of one-dimensional data

```
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8,
10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

- But, can't use directly with arithmetical operators (+, -, *, /, ...)
- Need efficient arrays with arithmetic and better multidimensional tools
- Numpy is similar to lists, but much more capable for scientific computing

Creating Arrays

To create a NumPy array containing only zeros: **np.zeros**

```
>>> import numpy as np
>>> a = np.zeros(3)
>>> a
array([ 0.,  0.,  0.])
>>> type(a)
numpy.ndarray
```

n-dimensional array

Creating Arrays

To create a NumPy array containing only zeros: **np.zeros**

```
>>> import numpy as np
>>> a = np.zeros(3)
```

- Similar to Lists except homogenous data type (or dtype in numpy)
- Default dtype for arrays: float64
- To have integer dtype: a = np.zeros(3, dtype=int)

n-dimensional array

Creating Arrays

```
>>> import numpy as np
# can also be created by list or tuples
>>> a = array([1,2,3,4,5])
>>> a
array([1, 2, 3, 4, 5])
np.ones(3), np.empty(3) ???? (Try it
out)
>>> b = np.arange(5) # similar to range
>>> b
array([0, 1, 2, 3, 4])
```

Creating Arrays

To set up a grid of evenly spaced numbers: **np.linspace**

```
>>> import numpy as np

>>> a = np.linspace(1, 10, 3)# generate 3 equally
spaces elements from 1 to 10 (1.,5.5,10.)
```

2D array from a list of list:

```
>>> import numpy as np
>>> z = np.array([[1, 2], [3, 4]])
>>> z
array([[1, 2],
       [3, 4]])
>>> z.shape # (2,2)
```

Array Indexing

- Elements of arrays are accessed using square brackets.
- Python is row major NOT column major, the first index is the row, not the column.
- Indexing starts at zero.
- For a flat array, indexing is the same as Python sequences

```
>>> import numpy as np
>>> z = np.linspace(1, 2, 5)
>>> z
array([ 1., 1.25, 1.5 , 1.75, 2.])
>>> z[0]
1.0
>>> z[0:2] ???
>>> z[-1] ???
```

2-D Array Indexing

```
>>> import numpy as np
>>> z = np.array([[1, 2], [3, 4]])
>>> z[0, 0]
1
>>> z[0, 1]
2
>>> z[0,:]
array([1, 2])
```

Array I/O

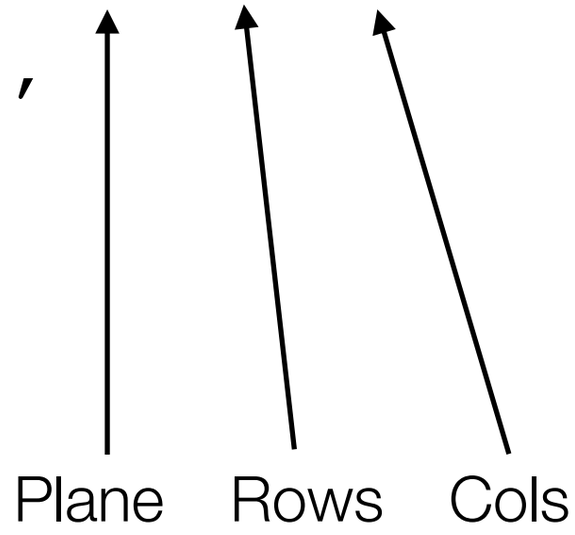
```
import numpy as np

# create an array,write to file, read from file

arr = np. array([[ 1 , 2, 3] , [4 , 5, 6]] )

# save to a text file, default is space delimited

np.savetxt(fname='array out.txt', X=arr, fmt=%d)

# load text file

loaded_arr = np.loadtxt (fname='array_out.txt')
```

Reshaping arrays

```
import numpy as np
a = np.arange(6) # 1d array
array([0, 1, 2, 3, 4, 5])
# reshape the array
b = arange(6).reshape(3, 2) # 2d array
array([[ 0, 1],
[ 2,3],
[ 4,5]])
```

Reshaping arrays

```
import numpy as np
c = arange(24).reshape(2, 3, 4) # 3d array
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]],
[[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23]]])
```



Array attributes

```
arr = np.arange(10).reshape((2, 5))
```

arr.ndim # 2 number of dimensions

arr.shape # (2, 5) shape of the array

arr.size #10 number of elements

arr.dtype # data type of elements in the array

Array methods

A.astype(T)

```
>>> s1 = np.arange(5, 10)
>>> print(s1)
[5 6 7 8 9]
>>> s2 = s1.astype(np.float)
>>> print(s2)
[ 5. 6. 7. 8. 9.]
```

Array methods

A.copy()

```
>>>s2
[ 5. 6. 7. 8. 9.]
>>> s3 = s2.copy()
>>> s2[2] = 73.88
>>> print(s2)
[ 5. 6. 73.88 8. 9. ]
>>> print(s3)
[ 5. 6. 7. 8. 9.]
```

Array methods

a.sort()

```
>>> a
[1, 2, 3, 4]
```

a = np.array((4, 3, 2, 1))

a = array([4, 3, 2, 1])

a.sum()

```
>>> a.sum()
10
```

a.mean()

```
>>> a.mean()
2.5
```

a.min()

```
>>> a.min()
1
```

a.max()

```
>>> a.max()
4
```

Array methods

a.argmax()

```
# Returns the index of the maximal element
>>> a.argmax()
3
```

a = array([1, 2, 3, 4])

a. cumsum()

```
#Cumulative sum of the elements of A
>>> a.cumsum()
[1, 3, 6, 10]
```

Array methods

a = array([1, 2, 3, 4])

a.cumprod()

```
# Cumulative product of
the elements of A
>>> a.cumprod()
[1, 2, 6, 24]
```

a. std()

```
#standard deviation
>>> a.std()
1.1180339887498949
```

a. var()

```
#Variance
>>> a.var()
1.25
```

a.transpose()

```
# transpose of a matrix
>>> a.transpose()
array([1 2 3 4])
```

Array methods

searchsorted(()

```
# returns the index of the first element of
an array that is >= to a particular value

test = np.linspace(2, 4, 5)
test = array([ 2. ,  2.5,  3. ,  3.5,  4. ])
```

```
>>> test.searchsorted(2.2)
1
>>> test.searchsorted(2.5)
1
>>> test.searchsorted(2.6)
2
```

Array methods

Random numbers

numpy.random.rand —> Random values in a given shape.

```
np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

numpy.random.randint —> Return random integers from low (inclusive) to high (exclusive).

```
np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
```

For more information: http://docs.scipy.org/doc/numpy/reference/routines.random.html

Array operations: Algebraic

The algebraic operators +, -, *, / and ** all act **element-wise** on arrays

```
>>> v1 = np.arange(0.6, 1.6, 0.1)
>>> print(v1)
[ 0.6 0.7 0.8 0.9 1. 1.1 1.2 1.3 1.4 1.5]
>>> v2 = np.arange(40.0, 50.0, 1.0)
>>> print(v2)
[ 40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
>>> print(v1+v2)
[ 40.6 41.7 42.8 43.9 45. 46.1 47.2 48.3 49.4 50.5]
```

Array operations: Algebraic

```
>>> v1 = [ 0.6 0.7 0.8 0.9 1. 1.1 1.2 1.3 1.4 1.5]
>>> v2 = [40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
>>> print(v1*v2)
[ 24. 28.7 33.6 38.7 44. 49.5 55.2 61.1 67.2 73.5]
>>> print(v1-v2)
[-39.4 -40.3 -41.2 -42.1 -43. -43.9 -44.8 -45.7 -46.6
-47.5]
>>> print(v1**2)
[ 0.36 0.49 0.64 0.81 1. 1.21 1.44 1.69 1.96 2.25]
```

Array operations: Algebraic

We can add a scalar to each element as follows, also called **broadcasting**.

```
>>> print v1
[ 0.6 0.7 0.8 0.9 1. 1.1 1.2 1.3 1.4 1.5]
>>> print v1 + 0.4
[ 1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
>>> print v1 * 10
[ 6. 7. 8. 9. 10. 11. 12. 13. 14. 15.]
>>> print v1*10+100
[ 106. 107. 108. 109. 110. 111. 112. 113. 114. 115.]
```

Array operations: 2-D

```
A = np.ones((2, 2))

B = np.ones((2, 2))

A + B
array([[ 2.,  2.],
       [ 2.,  2.]])

A + 10
array([[ 11.,  11.],
       [ 11.,  11.]])

A * B
array([[ 1.,  1.],
       [ 1.,  1.]])
```

Element-wise operation on 2D-arrays

Matrix dot and cross products

To find the dot product of two arrays a and b: **np.dot(a, b)**

```
>>> a1 = np.array([[1, 2, -4], [3, -1, 5]])
>>> a2 = np.array([[6, -3], [1, -2], [2, 4]])
>>> print(a1)
[[ 1  2 -4]
 [ 3 -1  5]]
>>> print(a2)
[[ 6 -3]
 [ 1 -2]
 [ 2  4]]
>>> np.dot(a1, a2)
```

Similarly cross product: **np.cross(a, b)**

Vectorized/ufuncs functions

NumPy provides versions of the standard functions log, exp, sin, etc. that act element-wise on arrays.

```
z = np.array([1, 2, 3])
np.sin(z)

array([ 0.84147098,  0.90929743,  0.14112001])
```

This eliminates the need for explicit element-by-element loops:

for i in range(n):

*   y[i] = np.sin(z[i])*

Comparisons

Element-wise operation:

```
>>> z = np.array([2, 3])
>>> y = np.array([2, 3])
>>> z == y
array([ True,  True], dtype=bool)
>>> y[0] = 5
>>>  z == y
>>> array([False,  True], dtype=bool)
>>> z != y
Out[102]: array([ True, False], dtype=bool)
```

Linear algebra function

A number of linear algebra functions are available as sub-module **linalg** of numpy.

np.linalg.det(a)

```
# returns the determinant
>>> m = np.array(((2,3), (-1, -2)))
>>> print m
[[ 2 3]
[-1 -2]]
>>> print np.linalg.det(m)
-1.0
```

Linear algebra function

numpy.linalg.solve(a, b)

```
# Solve a linear matrix equation,or system
of linear scalar equations
```

3x0 + x1 = 9 and x0 + 2x1 = 8

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

For more information: https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Pandas

- Pandas is a package of fast, efficient data analysis tools for Python
- Similar to NumPy it provides basic array type plus core array operations
- Defines some fundamental structures for working with data
- Form the first steps of data analysis

Pandas Strength

- Fast reading of data
- Row and columns manipulation
- Working with dates and time series **(very strong)**
- Sorting, grouping, re-ordering and general data munging
- Dealing with missing values etc. etc.

Pandas Data type

- Series
  - ➡ Like a column of a data
  - ➡ Collection of observations on a single variable
- Data frame
  - ➡ an object for storing related columns of data
  - ➡ Can think of as highly optimised spread sheet

Lets start Pandas

```
import pandas as pd

import numpy as np
```

Pandas library import

Also import numpy to handle Series

Series

```
>>> s = pd.Series(np.random.randn(4),
name='vals')

>>> s
0    0.990678
1   -0.464966
2   -0.192195
3    0.116839
Name: vals, dtype: float64
```

Index

Name and type of the series

Series

- Series are built on top of NumPy arrays, and support many similar operations

```
>>> s*100
0     99.067830
1    -46.496589
2    -19.219480
3     11.683925
Name: daily returns, dtype: float64
>>> np.abs(s)
0     0.990678
1     0.464966
2     0.192195
3     0.116839
Name: daily returns, dtype: float64
```

Series

- But Series has something more than Numpy

```
>>> s.describe()
count     4.000000
mean      0.112589
std       0.631802
min      -0.464966
25%      -0.260388
50%      -0.037678
75%       0.335299
max       0.990678
Name: daily returns, dtype: float64
```

Series

- More flexible indices

```
>>> s.index = ['APPLE', 'ORANGE', 'BANANA','GRAPES']
>>> s
APPLE      0.990678
ORANGE    -0.464966
BANANA    -0.192195
GRAPES     0.116839
Name: daily returns, dtype: float64
```

Data Frame

- DataFrame is all the columns while Series is single column
- We will go though data frames using sample file of titanic disaster (source: kaggle.com)

Table 1

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S

Data Frame

Table 1

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S

Reading CSV file:

```
>>> df = pd.read_csv(path_to_file)
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
                                                Name     Sex  Age  SibSp  \
0                            Braund, Mr. Owen Harris    male   22      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38      1
2                             Heikkinen, Miss. Laina  female   26      0
```

Data Frame

Table 1

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S

Get particular rows:

```
>>> df[0:2]
                                                Name     Sex  Age  SibSp  \
0                            Braund, Mr. Owen Harris    male   22      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38      1
```

Data Frame

Table 1

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S

Get particular cols:

```
>>> df[['Name', 'Age']]
                                                 Name  Age
0                             Braund, Mr. Owen Harris   22
1   Cumings, Mrs. John Bradley (Florence Briggs Th...   38
2                              Heikkinen, Miss. Laina   26
3        Futrelle, Mrs. Jacques Heath (Lily May Peel)   35
4                            Allen, Mr. William Henry   35
5                                    Moran, Mr. James  NaN
```

Data Frame

Table 1

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S

Select both rows and cols:

```
>>> df.ix[0:2,['Name', 'Age']]
```

```
                                                Name  Age
0                            Braund, Mr. Owen Harris   22
1  Cumings, Mrs. John Bradley (Florence Briggs Th...   38
2                             Heikkinen, Miss. Laina   26
```

Data Frame

Conditional indexing

```
>>> df[df['Age'] > 25]
```

	Name	Sex	Age	SibSp
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38	1
2	Heikkinen, Miss. Laina	female	26	0
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1
4	Allen, Mr. William Henry	male	35	0
6	McCarthy, Mr. Timothy J	male	54	0
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27	0

Renaming Column

```
>>> df2 = df.rename(columns={'Name': 'Full Name'}) #copies the data to another DataFrame.
```

	Full Name	Sex	Age	SibSp	\
0	Braund, Mr. Owen Harris	male	22	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38	1	

Handling missing Values

Drop missing values:

The **dropna** can be used to drop rows or columns with missing data (NaN).

By default, it drops all rows with any missing entry.

```
>>> df.dropna()
```

Fill missing values:

The **fillna** can be used to fill missing data (NaN). Lets say for titanic data we want to fill missing ticket fare with mean ticket fare

```
>>> df3 = df.copy()
>>> mean_fare = df3[‘Fare’].mean()
>>> df3['Fare'].fillna(mean_fare)
```

Map and Apply

- No need to use for loops while using pandas.
- Vectorized computation by applying function over rows and columns using the **map**, **apply** and **applymap** methods.

	float_col	int_col	str_col
0	0.1	1	a
1	0.2	2	b
2	0.2	6	None
3	10.1	8	c
4	NaN	-1	a

Map

The map operation operates over each element of a Series.

Lets try to add “map” in str column.

```
df['str_col'].dropna().map(lambda x : 'map_' + x)
```

```
0     map_a
1     map_b
3     map_c
4     map_a
Name: str_col
```

Apply

Applies function along input axis of DataFrame.

Lets try to increment each value of int column.

```
df.apply(lambda x:x+1,axis=0))
```

axis = 0 —> along the rows

axis = 1 —> along the cols

Applymap

The **applymap** operation can be used to apply the function to each element of the DataFrame.

Lets try to add “applymap” to each string column

```
def add_substring(x):
    if type(x)== str:
        return 'applymap_' + x
    else:
        return
df.applymap(add_substring)
```

Applymap

```
def add_substring(x):
    if type(x)== str:
        return 'applymap_' + x
    else:
        return
df.applymap(add_substring)
```

	float_col	int_col	str_col
0	0.1	1	applymap_a
1	0.2	2	applymap_b
2	0.2	6	None
3	10.1	8	applymap_c
4	NaN	-1	applymap_a

Vectorized Operations

```
df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3]})
df["C"] = df["A"]+df["B"]
 df
```

	A	B	C
0	1	1.2	2.2
1	2	1.3	3.3

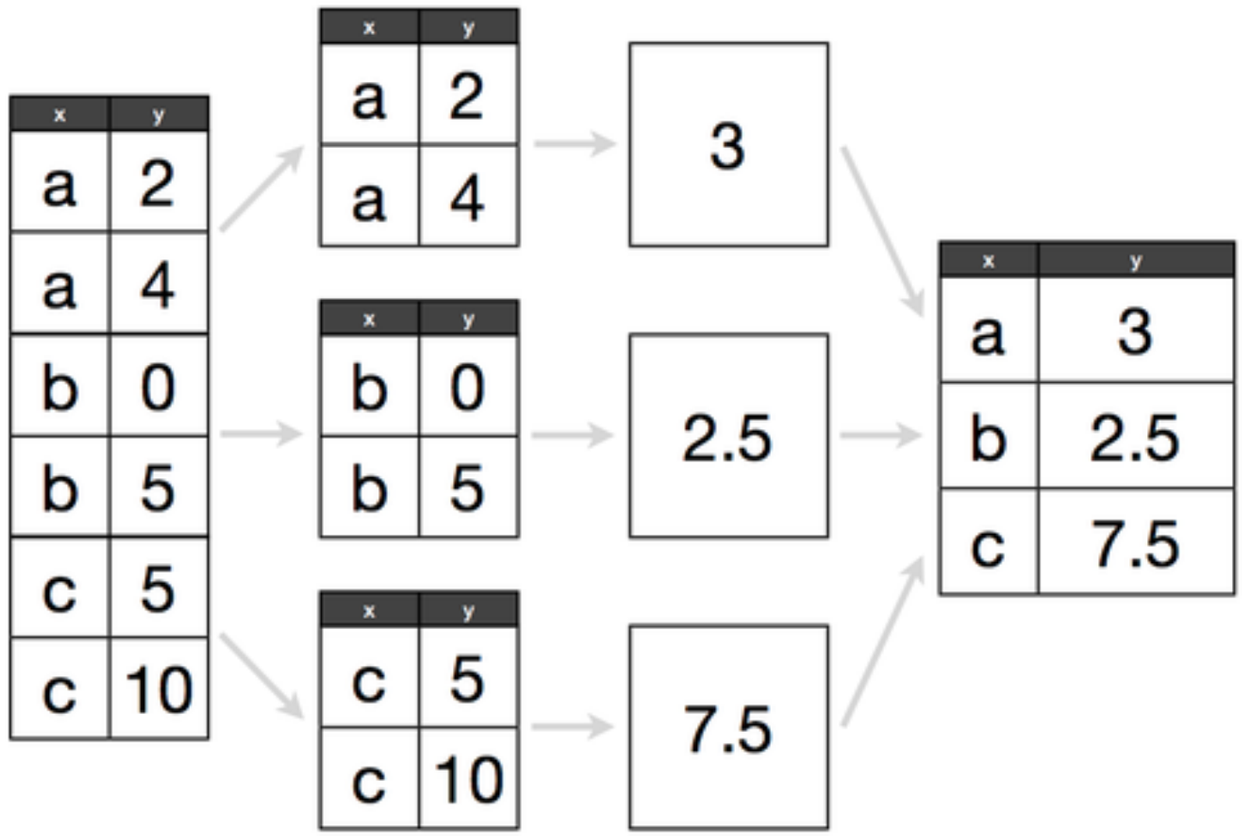
Try:

```
df["D"] = df["A"]*3
df["E"] = np.sqrt(df["A"])
```

Pandas Grouping

pandas **groupby** method draws largely from the split-apply-combine strategy for data analysis.

Very similar to SQL GROUP BY



Source: Gratuitously borrowed from Hadley Wickham's Data Science in R slides

Pandas Grouping

Lets try to groupby titanic data based on gender type male/ female

```
by_gender = df.groupby('Sex')
by_gender
<pandas.core.groupby.DataFrameGroupBy object at
0x105343a20>
by_gender.size()# number of records
Sex
female    24
male      25
dtype: int64
```

Pandas Grouping

Apply some operations after splitting

```
by_gender.Age.sum()
ex
female     484
male       487
Name: Age, dtype: float64
```

Try:

mean, median of age

Pandas Grouping

All keys after grouping:

```
>>> gender = list(by_gender.groups.keys())
>>> gender
['female', 'male']
```

Get data associated to each key:

```
>>> gender_female_df = by_gender.get_group('male')
```

Pandas Stats

```
   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

describe()

```
>>> df.describe()
```

```
       float_col   int_col
count    4.00000  5.000000
mean     2.65000  3.200000
std      4.96689  3.701351
min      0.10000 -1.000000
25%      0.17500  1.000000
50%      0.20000  2.000000
75%      2.67500  6.000000
max     10.10000  8.000000
```

Pandas Stats

```
   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

correlation()

```
>>> df.corr()
```

```
           float_col   int_col
float_col   1.000000  0.760678
int_col     0.760678  1.000000
```

Merge and Join

- Pandas supports database-like joins
- Makes it easy to link data frames.
- {'left', 'right', 'outer', 'inner'}, default 'inner'
  - ➡ left: use only keys from left frame (SQL: left outer join)
  - ➡ right: use only keys from right frame (SQL: right outer join)
  - ➡ outer: use union of keys from both frames (SQL: full outer join)
  - ➡ inner: use intersection of keys from both frames (SQL: inner join)

Left Join

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                            'right_value': ['f', 'g', 'h', 'i', 'j']})
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e

   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

Left Join

```
pd.merge(left_frame, right_frame, on='key', how
='left')
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h

Examples taken from http://www.gregreda.com/2013/10/26/working-with-pandas-dataframes/

Right Join

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                            'right_value': ['f', 'g', 'h', 'i', 'j']})
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e

   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

Right Join

```
pd.merge(left_frame, right_frame, on='key', how
='right')
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h
3	5	NaN	i
4	6	NaN	j

Examples taken from http://www.gregreda.com/2013/10/26/working-with-pandas-dataframes/

Outer Join

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                            'right_value': ['f', 'g', 'h', 'i', 'j']})
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e

   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

Outer Join

```
pd.merge(left_frame, right_frame, on='key', how
='outer')
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h
5	5	NaN	i
6	6	NaN	j

Examples taken from http://www.gregreda.com/2013/10/26/working-with-pandas-dataframes/

Inner Join

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                            'right_value': ['f', 'g', 'h', 'i', 'j']})
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e

   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

Inner Join

```
pd.merge(left_frame, right_frame, on='key', how
='inner')
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h

Examples taken from http://www.gregreda.com/2013/10/26/working-with-pandas-dataframes/

Time series

- Lets understand it through example
- Download monthly Artic Oscillation (sea level pressure) data from:

http://www.cpc.ncep.noaa.gov/products/precip/CWlink/daily_ao_index/monthly.ao.index.b50.current.ascii

```
ao = np.loadtxt('monthly.ao.index.b50.current.ascii.txt')
```

- Data start date: January 1950
- Data end date: February 2016

```
# create some dates with frequency as month

 dates = pd.date_range('1950-01', '2016-02', freq='M')
```

Time series

```
dates

DatetimeIndex(['1950-01-31', '1950-02-28', '1950-03-31',
 '1950-04-30',
                ...
               '2015-04-30', '2015-05-31', '2015-06-30',
'2015-07-31',
               '2015-08-31', '2015-09-30', '2015-10-31',
'2015-11-30'],
              dtype='datetime64[ns]', length=793,
 freq='M', tz=None)

>>> dates.shape

(793,)
```

Time series

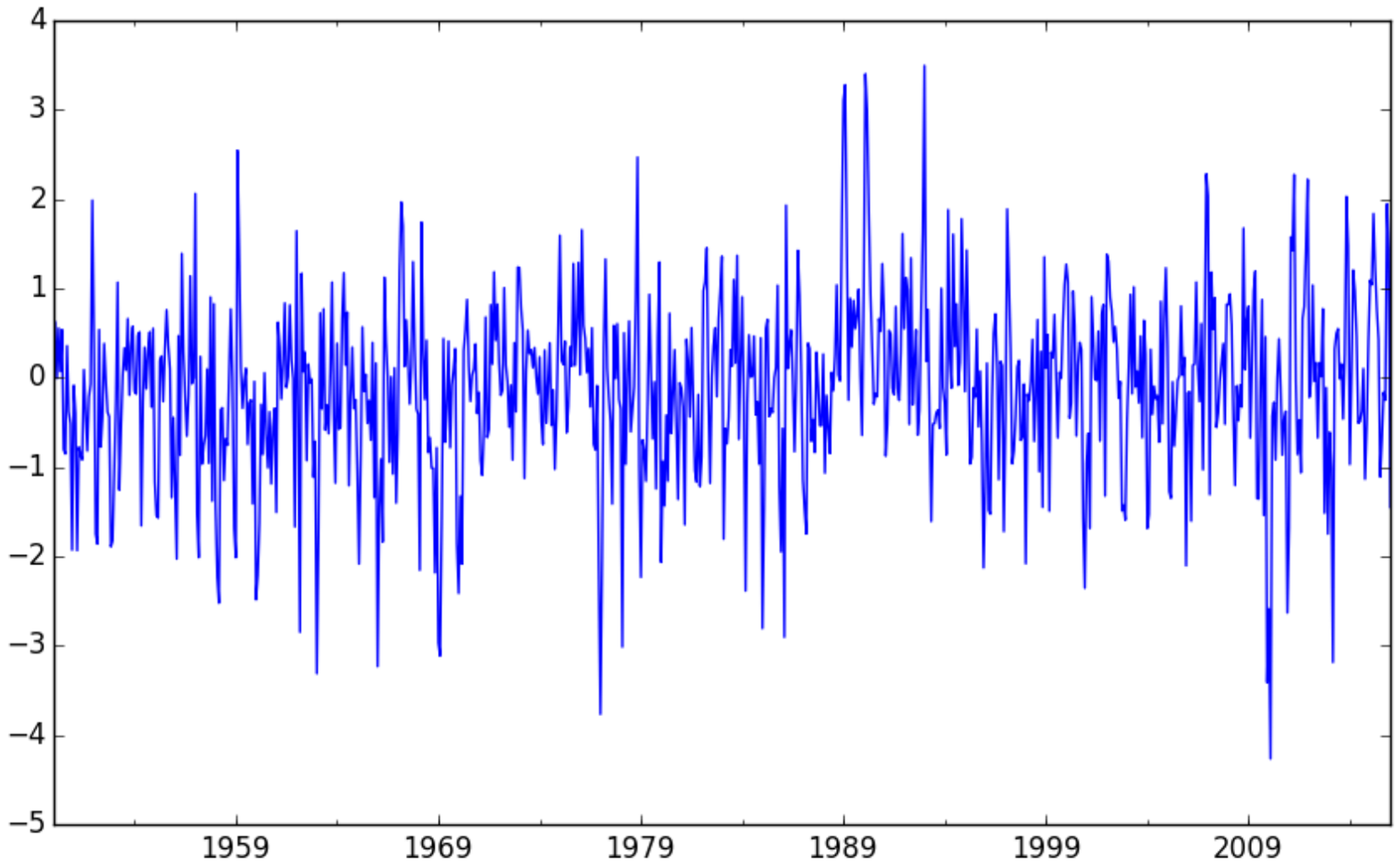
- Get all ao_values for each date

```
>>> AO_data = pd.Series(ao[:793,2], index=dates)
>>> AO_data
1950-01-31   -0.060310
1950-02-28    0.626810
1950-03-31   -0.008127
1950-04-30    0.555100
1950-05-31    0.071577
1950-06-30    0.538570
```

Time series

- Now lets try to plot it

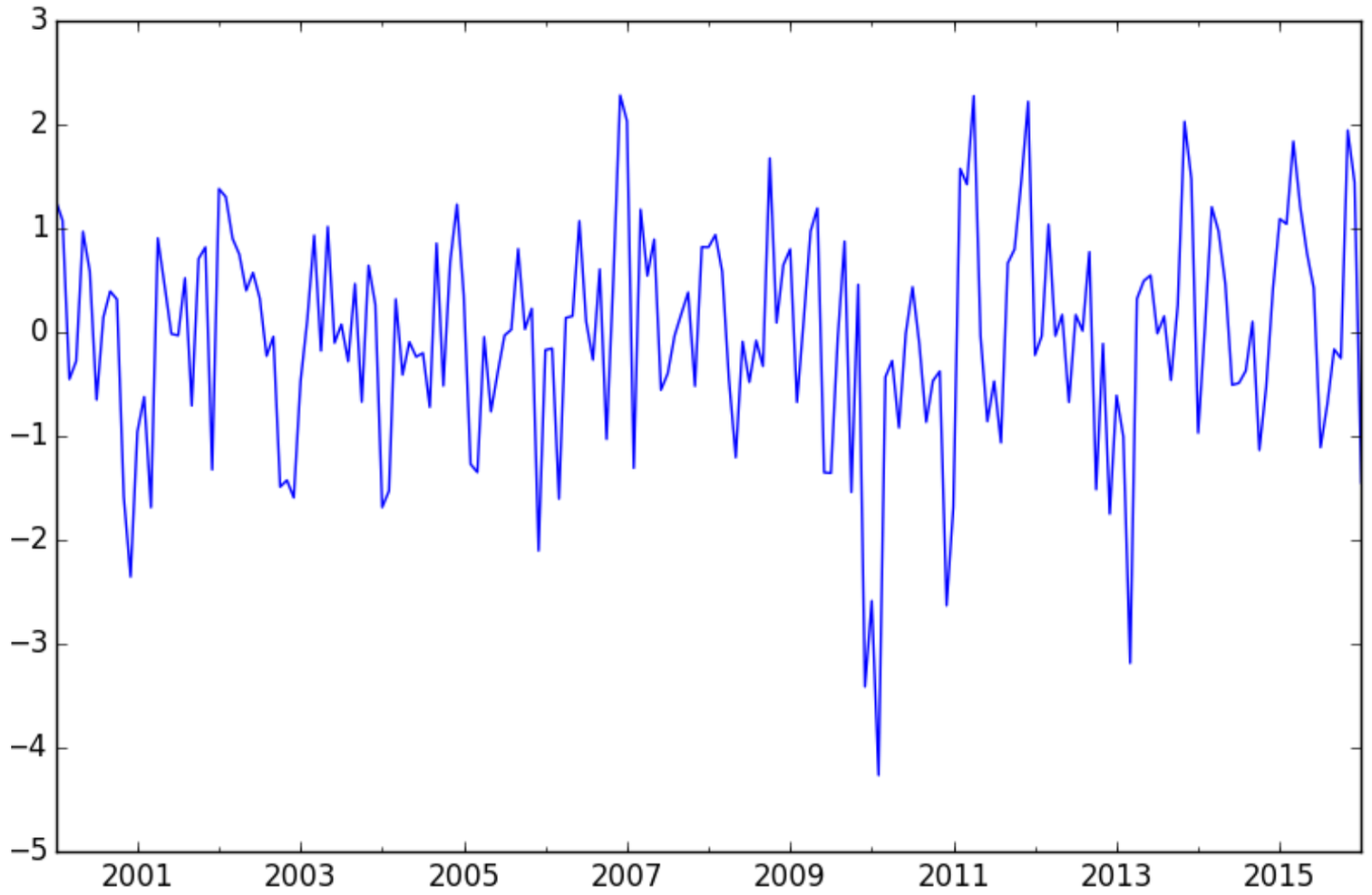
```
>>> AO_data.plot()
```



Time series

- You can also plot the part of the data

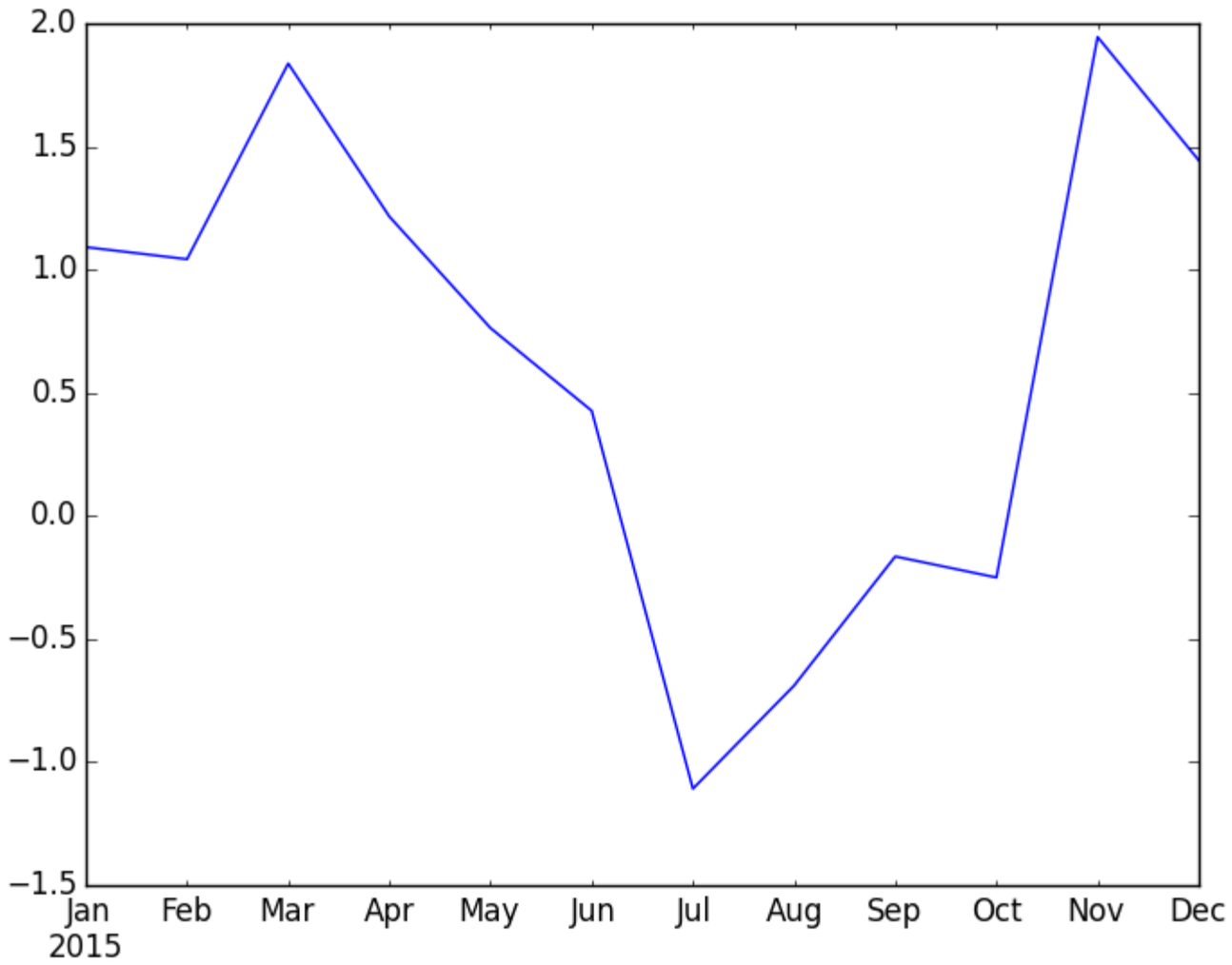
```
>>> AO_data['2000':'2016'].plot()
```



Time series

- Even much smaller

```
>>> AO_data['2015-01':'2015-12'].plot()
```



Time series

- Lets take more data to make it more interesting
- Windows download monthly NAO (north atlantic oscillation) data from:

http://www.cpc.ncep.noaa.gov/products/precip/CWlink/pna/
norm.nao.monthly.b5001.current.asci

- Repeat the same steps as AO data and create NAO_data frame, also try to play with data and create some plots

Time series

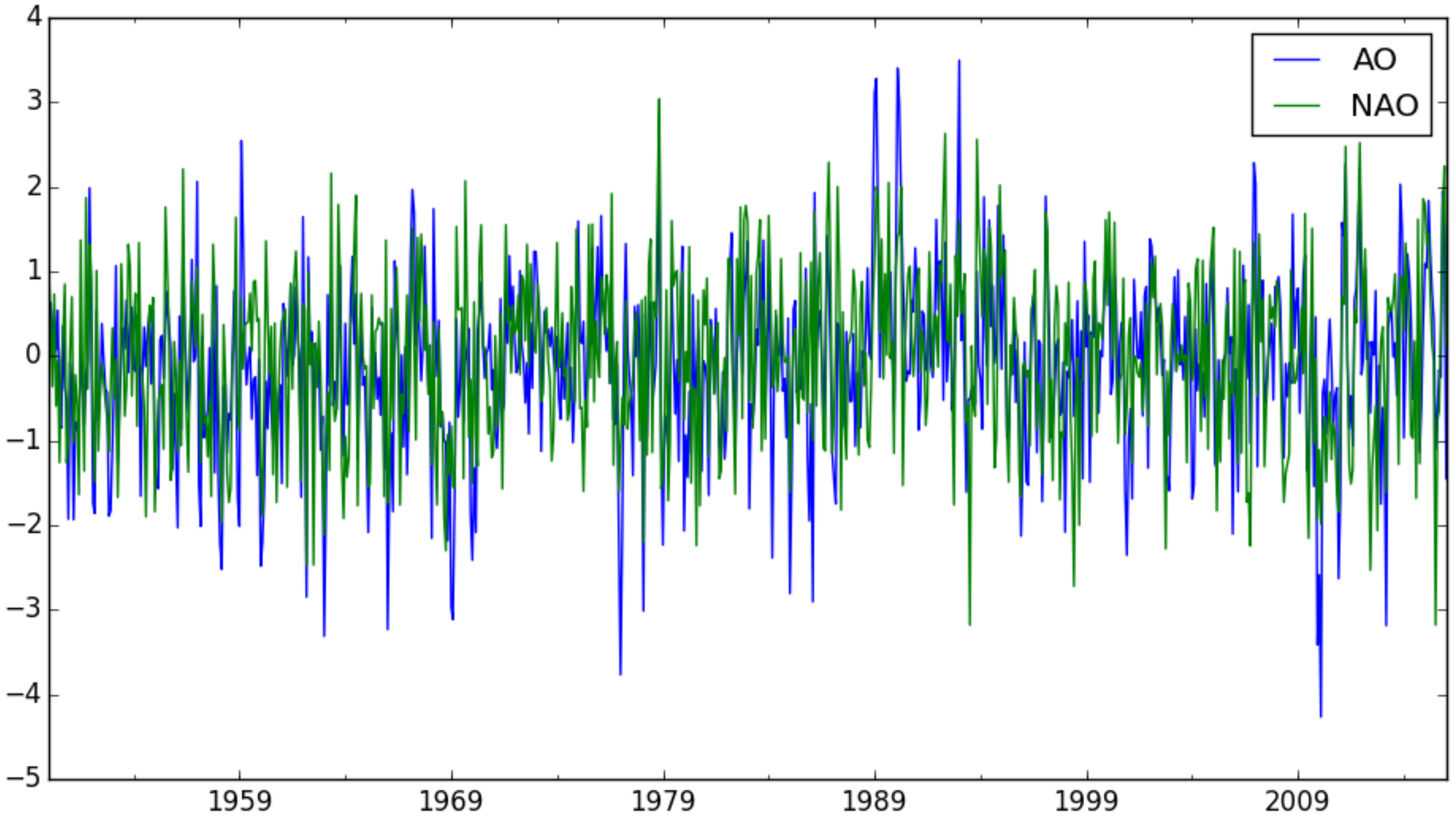
- Create data frame that contains both AO and NAO data

```
ao_nao = pd.DataFrame({'AO' : AO_data, 'NAO' :
NAO_data})
```

- Lets plot both data

```
ao_nao.plot()
```

Time series



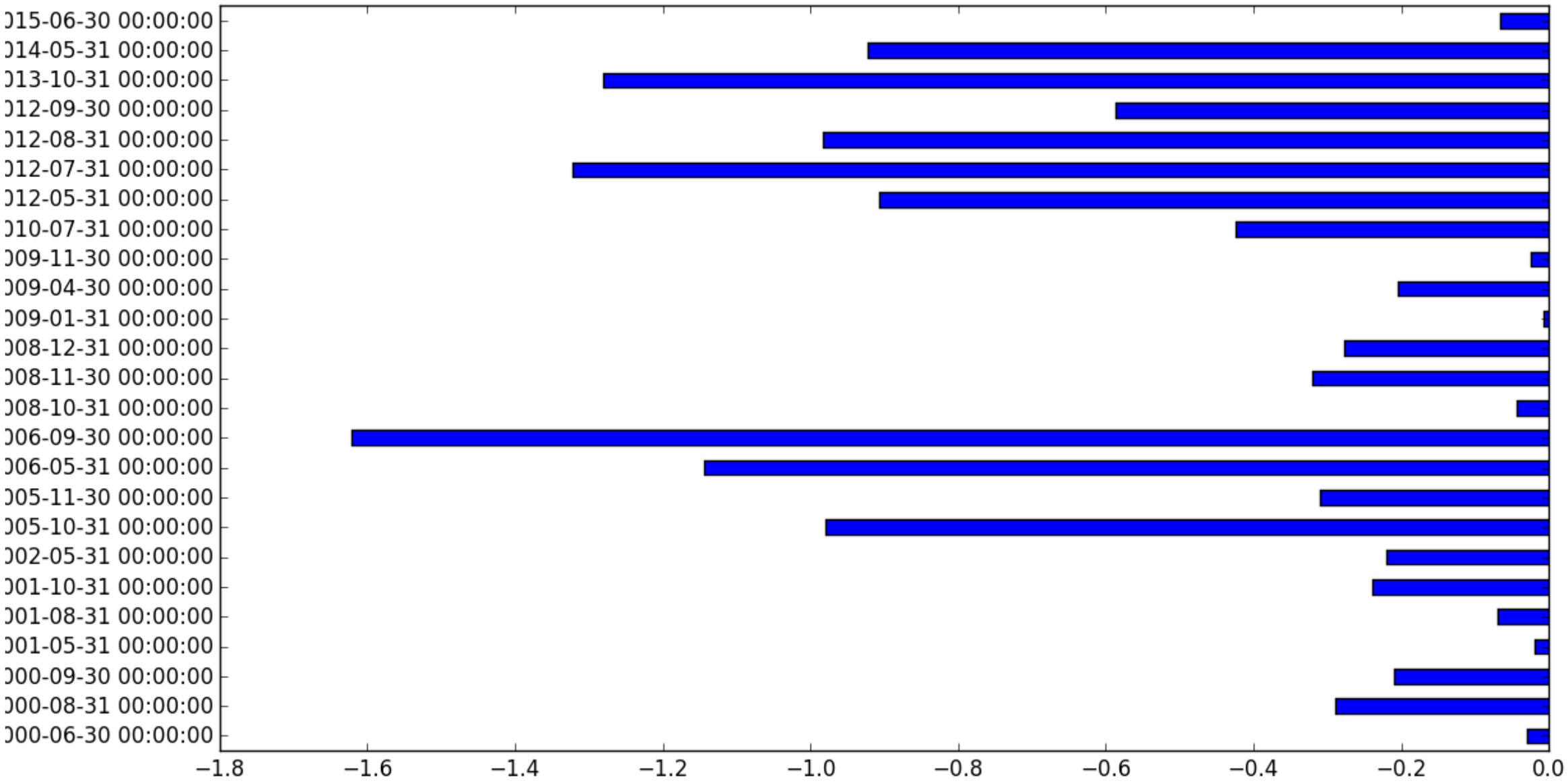
Time series

- You can also apply some conditions on data and plot it

```
import datetime

aonao.ix[(aonao.AO > 0)&(aonao.NAO < 0)&
(aonao.index > datetime.datetime(2000,1,1))&
(aonao.index<datetime.datetime(2016,1,1)),'NAO']
.plot(kind='barh')
```

What this extraction does ?

Time series


2015-06-30 00:00:00
2014-05-31 00:00:00
2013-10-31 00:00:00
2012-09-30 00:00:00
2012-08-31 00:00:00
2012-07-31 00:00:00
2012-05-31 00:00:00
2010-07-31 00:00:00
2009-11-30 00:00:00
2009-04-30 00:00:00
2009-01-31 00:00:00
2008-12-31 00:00:00
2008-11-30 00:00:00
2008-10-31 00:00:00
2006-09-30 00:00:00
2006-05-31 00:00:00
2005-11-30 00:00:00
2005-10-31 00:00:00
2002-05-31 00:00:00
2001-10-31 00:00:00
2001-08-31 00:00:00
2001-05-31 00:00:00
2000-09-30 00:00:00
2000-08-31 00:00:00
2000-06-30 00:00:00
−1.8
−1.6
−1.4
−1.2
−1.0
−0.8
−0.6
−0.4
−0.2
0.0

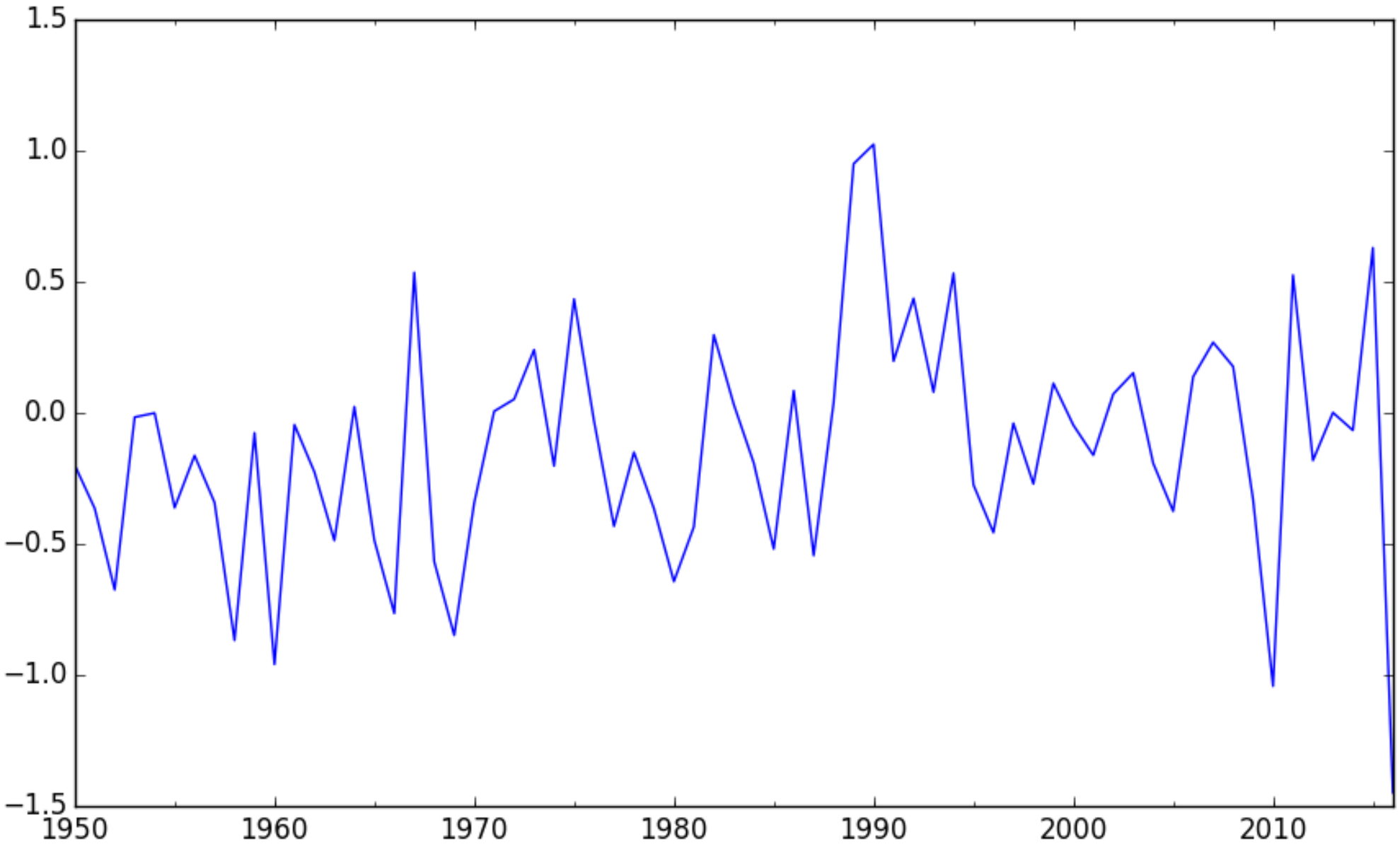
Resampling

- Resampling of data is done for different time frequency
- Resampling time period and method (default = mean) are important parameters
- Lets resample our data annually on mean

```
AO_data_mm = AO_data.resample("A", how = "mean")
```

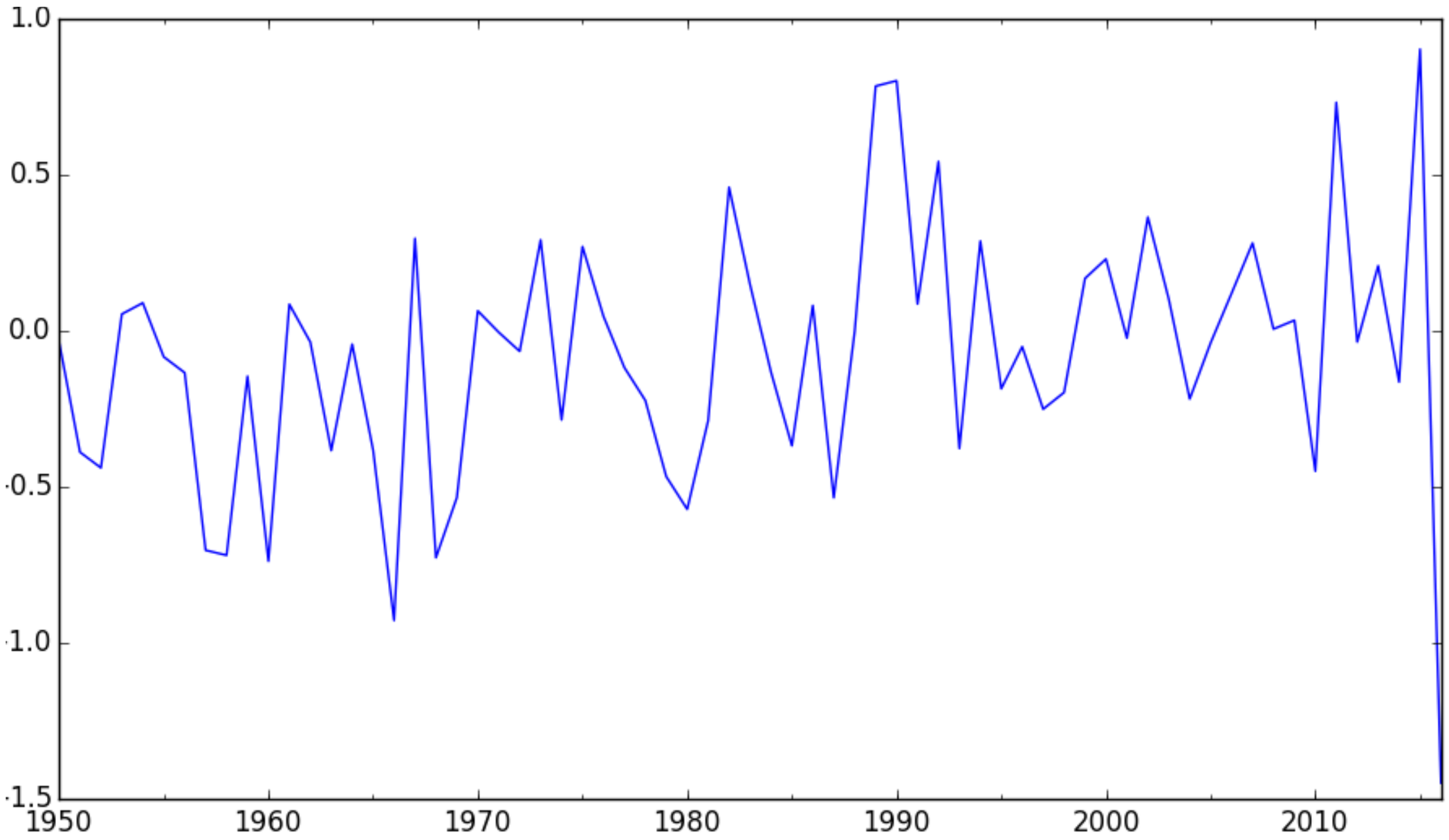
Annually/yearly

Resample: mean



Resample: median

```
AO_data_mm = AO_data.resample("A", how='median')
```



Final thoughts

- Pandas is very powerful data analysis tool.
- We covered most of the building blocks but still there is a lot…..
- To master Pandas -> practice, practice and practice

More information: http://pandas.pydata.org/pandas-docs/stable/