Tufts

CS135
Introduction to Machine Learning

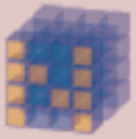
Lecture 2: Numpy and Pandas

Python comes with batteries included

➜ extensive Python standard library

What about batteries for scientists (and others as well)?

➜ scientific Python ecosystem



NumPy

Base N-dimensional array package

SciPy library

Fundamental library for scientific computing

Matplotlib

Comprehensive 2D Plotting

IP[y]: IPython

IPython

Enhanced Interactive Console

Sympy

Symbolic mathematics

pandas

Data structures & analysis

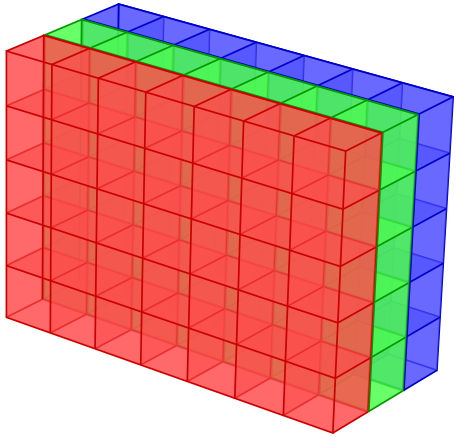
from: www.scipy.org

+ SciKits and many other packages

Wish list

- we want to work with vectors and matrices

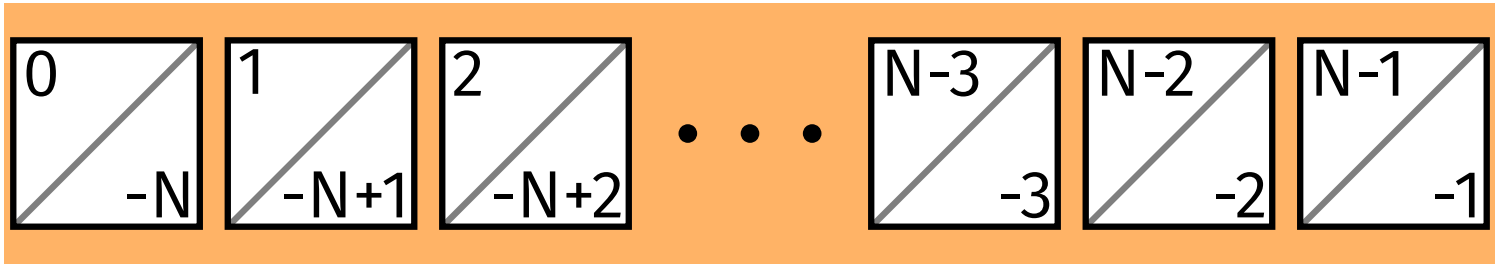
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$



colour image as $N \times M \times 3$-array

- we want our code to run fast
- we want support for linear algebra
- ...

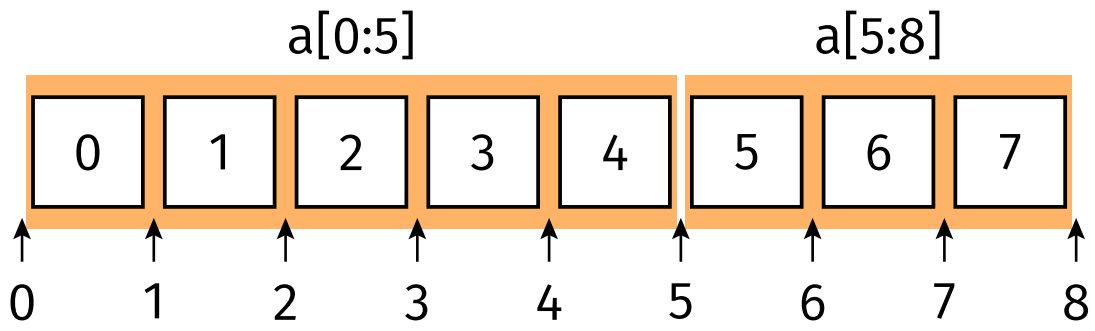
List indexing



- indexing starts at 0
- negative indices count from the end of the list to the beginning

List slicing

basic syntax: `[start:stop:step]`



- if `step=1`
  - slice contains the elements `start` to `stop-1`
  - slice contains `stop-start` elements
- `start`, `stop`, and also `step` can be negative
- default values:
  - `start` 0, i.e. starting from the first element
  - `stop` N, i.e up to and including the last element
  - `step` 1

Matrices and lists of lists

Can we use lists of lists to work with matrices?

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
matrix = [[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]
```

- How can we extract a row?
- How can we extract a column?

Lists of lists do not work like matrices

Problems with matrices

- different axes are not treated on equal footing
- lists can contain arbitrary objects
  matrices have a homogeneous structure
- list elements can be scattered in memory

Applied to matrices …

…lists are conceptually inappropriate

…lists have less performance than possible

We need a new object

ndarray

multidimensional, homogeneous array of fixed-size items

Numpy

- fast and space-efficient multidimensional array
- Standard mathematical functions for fast operations on entire array avoiding for loops
- Linear algebra, random number generation
- Tools for integrating code written in C, C++, and Fortran

Creating Arrays

```
>>> import numpy as np
>>> data1 = [6, 7.5, 8, 0, 1]  # create data in a list
>>> arr1 = np.array(data1)     # create numpy object.
array([ 6. ,  7.5,  8. ,  0. ,  1. ])
>>> data2 = [[1, 2, 3, 4],[5, 6, 7, 8]]
>>> arr2 = np.array(data2)
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> arr2.shape
(2, 4)
>>> arr1.dtype
dtype('float64')
>>> arr2.dtype
dtype('int64')
```

Creating Arrays (2)

```
>>> np.zeros(10)
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> np.zeros((3,6))
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

>>> np.empty((2,3,2))
array([[[  0.00000000e+000,   0.00000000e+000],
        [ 1.27319748e-313,   1.27319747e-313],
        [ 1.27319747e-313,   1.27319747e-313]],


       [[ 8.39911598e-323,   4.44659081e-323],
        [ 2.65130035e-313,   2.12199580e-314],
        [ 3.40183879e-313,   1.25222910e-308]]])
```

Creating Arrays (3)

```
>>> np.ones((3,6))
array([[ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.]])

>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Data Types of ndarrays

```
>>> arr1 = np.array([1,2,3],dtype=np.float64)
>>> arr2 = np.array([4,5,6],dtype=np.int32)
>>> arr1.dtype
dtype('float64')
>>> arr2.dtype
dtype('int32')
>>> arr1/2
array([ 0.5,  1. ,  1.5])
>>> arr2/2
array([2, 2, 3], dtype=int32)
```

- Integer division in Numpy arrays.

Changing Data Types

- array.astype() method

```
>>> arr2
array([4, 5, 6], dtype=int32)
>>> float_arr = arr2.astype(np.float64)
>>> float_arr.dtype
dtype('float64')
```

Element-wise Array Operations

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

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

>>> arr * arr              # This is not matrix multiplication
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

>>> arr - arr
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>> arr + arr
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]])
```

Array and Scalars

```
>>> 1/arr
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667]])
>>> arr ** 0.5
array([[ 1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974]])
```

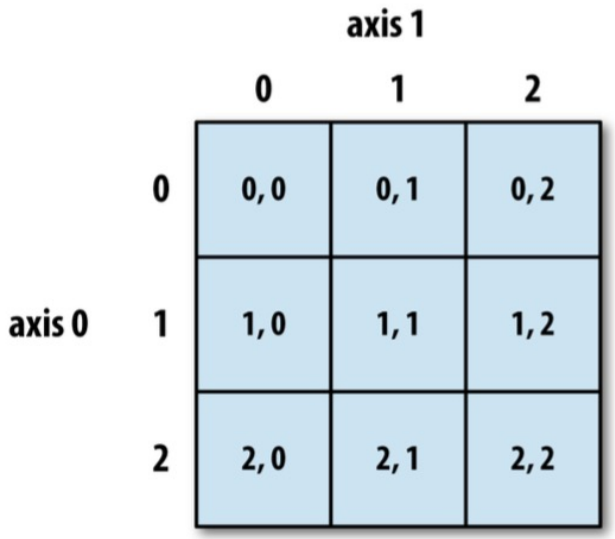
Basic Index and Slicing

```
>>> arr = np.arange(10)

>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> arr[5]
5

>>> arr[5:8]
array([5, 6, 7])

>>> arr[5:8]=12

>>> arr
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Basic Index and Slicing (2)

```
>>> arr_slice = arr[5:8]

>>> arr_slice[1] = 12345

>>> arr
array([    0,      1,      2,      3,      4,     12,
12345,     12,      8,      9])

>>> arr_slice[:]=64

>>> arr
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Basic Index and Slicing (3)



Expression	Shape
`arr[:2, 1:]`	`(2, 2)`
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1, 3)`
`arr[:, :2]`	`(3, 2)`
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1, 2)`

Indexing and slicing in higher dimensions

- usual slicing syntax
- difference to lists:
  slices for the various axes separated by comma

a[2, -3]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

- usual slicing syntax
- difference to lists:
  slices for the various axes separated by comma

a[:3, :5]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexing and slicing in higher dimensions

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Boolean Indexing

```
>>> names = np.array(['Bob','Joe','Will','Bob','Will','Joe','Joe'])
>>> data = np.random.randn(7,4)
>>> data
array([[-0.12343604,  2.04685658,  0.83095751, -1.05873731],
       [-1.21982411,  0.01785438, -0.57337724,  0.50338352],
       [ 1.1539617 ,  0.71137968, -1.15053635,  0.57132576],
       [-0.53783602,  0.24067336, -0.81966656, -0.25569241],
       [-1.28358017, -2.21748936,  0.40209673,  0.04342643],
       [-2.05112225, -1.31258861,  0.42707355, -0.95093388],
       [ 2.47354912,  1.7091805 ,  0.05354025,  0.27015468]])
>>> names == 'Bob'
array([ True, False, False,  True, False, False, False], dtype=bool)
>>> data[names=='Bob']
array([[-0.12343604,  2.04685658,  0.83095751, -1.05873731],
       [-0.53783602,  0.24067336, -0.81966656, -0.25569241]])
>>> data[0]
array([-0.12343604,  2.04685658,  0.83095751, -1.05873731])
```

Boolean Indexing (2)

```
>>> data[names=='Bob',2:]
array([[ 0.83095751, -1.05873731],
       [-0.81966656, -0.25569241]])
>>> data[names=='Bob',2:]
array([[ 0.83095751, -1.05873731],
       [-0.81966656, -0.25569241]])
>>> names != 'Bob'
array([False,  True,  True, False,  True,  True,  True], dtype=bool)
>>> mask = (names == 'Bob') | (names == 'Will')
>>> mask
array([ True, False,  True,  True,  True, False, False], dtype=bool)
```

Boolean Indexing (3)

```
>>> data[data<0] = 0
>>> data
array([[ 0.        ,  2.04685658,  0.83095751,  0.        ],
       [0.        ,  0.01785438,  0.        ,  0.50338352],
       [1.1539617 ,  0.71137968,  0.        ,  0.57132576],
       [ 0.        ,  0.24067336,  0.        ,  0.        ],
       [0.        ,  0.        ,  0.40209673,  0.04342643],
       [0.        ,  0.        ,  0.42707355,  0.        ],
       [2.47354912,  1.7091805 ,  0.05354025,  0.27015468]])
```

Transpose

```
>>> arr
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> arr.T
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Matrix Multiplication

- np.dot() Method
- * is used for elementwise multiplication

```
>>> u = np.array([[5],[-4],[8]])
>>> u
array([[ 5],
       [-4],
       [ 8]])
>>> v = np.array([[-3,2,-3]]).T
>>> v
array([[-3],
       [ 2],
       [-3]])
```

Matrix Multiplication (2)

```
>>> np.dot(u,v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
>>> np.dot(u.T,v)
array([[-47]])
```

Matrix Multiplication (3)

```
>>> A = np.array([[0,1,2],[1,2,3],[2,3,0]])
>>> A
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 0]])
>>> B = np.array([[3,0,4],[-2,2,1],[4,-1,2]])
>>> B
array([[ 3,  0,  4],
       [-2,  2,  1],
       [ 4, -1,  2]])
>>> np.dot(A,B)
array([[ 6,  0,  5],
       [11,  1, 12],
       [ 0,  6, 11]])
```

Fast Element-wise Array Functions

```
>>> arr = np.arange(10)
>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.sqrt(arr)
array([
0.        ,  1.        ,  1.41421356,  1.73205081,  2.
,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.
       ])
>>> np.exp(arr)
array([ 1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,
         8.10308393e+03])
```

Fast Element-wise Array Functions

```
>>> x = np.random.randn(8)
>>> x
array([-0.50678535,  1.08875418,  0.08067502, -0.24385984, -0.75510501,
        2.03938358, -0.6377626 ,  0.77175103])
>>> y = np.random.randn(8)
>>> y
array([ 0.54281294, -0.77852043,  1.03648467, -1.02540904,  0.01098809,
        0.83743411,  2.3423178 ,  0.87976044])
>>> np.maximum(x,y) # Element-wise maximum
array([ 0.54281294,  1.08875418,  1.03648467, -0.24385984,  0.01098809,
        2.03938358,  2.3423178 ,  0.87976044])
```

Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if condition else y

```
>>> xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
>>> yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
>>> cond = np.array([True, False, True, True, False])
>>> result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]
>>> zip(xarr,yarr,cond)
[(1.1000000000000001, 2.1000000000000001, True), (1.2, 2.2000000000000002,
False), (1.3, 2.2999999999999998, True), (1.3999999999999999,
2.3999999999999999, True), (1.5, 2.5, False)]
>>> result
[1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

- It will not be very fast for large arrays.
- It will not work with multidimensional arrays.

Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if condition else y

```
>>> result = np.where(cond,xarr,yarr)

>>> result
array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

- The second and third arguments to np.where don't need to be arrays; one or both of them can be scalars.

```
>>> arr = np.random.randn(4,4)

>>> arr
array([[ 1.39752677, -0.19462025, -0.14560082, -1.31851477],
       [ 2.25289253, -0.04246312, -3.09833434,  2.2046344 ],
       [-2.19293197, -0.40341835, -0.62235675,  0.35305701],
       [ 0.91293091,  0.13963139, -0.21237659, -0.72933765]])
```

Expressing Conditional Logic as Array Operations

```
>>> np.where(arr>0,2,-2)
array([[ 2, -2, -2, -2],
       [ 2, -2, -2,  2],
       [-2, -2, -2,  2],
       [ 2,  2, -2, -2]])

>>> np.where(arr>0,2,arr)
array([[ 2.        , -0.19462025, -0.14560082, -1.31851477],
       [ 2.        , -0.04246312, -3.09833434,  2.        ],
       [-2.19293197, -0.40341835, -0.62235675,  2.        ],
       [ 2.        ,  2.        , -0.21237659, -0.72933765]])
```

np.where

```
>>> arr = np.arange(9).reshape([3,3])

>>> arr
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.where(arr<=4)
(array([0, 0, 0, 1, 1]), array([0, 1, 2, 0, 1]))

>>> index = np.where(arr<=4)

>>> index
(array([0, 0, 0, 1, 1]), array([0, 1, 2, 0, 1]))

>>> result = arr[index]

>>> result
array([0, 1, 2, 3, 4])
```

Numpy Functions

- Find additional Numpy function at:
  https://docs.scipy.org/doc/numpy/reference/routines.html

PANDAS

Reading data using pandas

In [ ]:
```
#Read csv file
df = pd.read_csv("data/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

k

Exploring data frames

In [3]:
```
#List first 5 records
df.head()
```

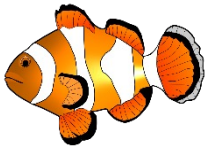
Out[3]:

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

Hands-on exercises

- ✓ Try to read the first 10, 20, 50 records;
- ✓ Can you guess how to view the last few records;

Hint:



Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frame data types

```
In [4]: #Check a particular column type
        df['salary'].dtype
```

```
Out[4]: dtype('int64')
```

```
In [5]: #Check types for all the columns
        df.dtypes
```

```
Out[4]: rank
        discipline
        phd
        service
        sex
        salary
        dtype: object
```

Data Frames attributes

Python objects have *attributes* and *methods*.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

Hands-on exercises

- ✓ Find how many records this data frame has;
- ✓ How many elements are there?
- ✓ What are the column names?
- ✓ What types of columns we have in this data frame?

Data Frames methods

Unlike attributes, python methods have *parenthesis.*
All attributes and methods can be listed with a *dir()* function: `dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Hands-on exercises

- ✓ Give the summary for the numeric columns in the dataset
- ✓ Calculate standard deviation for all numeric columns;
- ✓ What are the mean values of the first 50 records in the dataset?

Hint: use head() method to subset the first 50 records and then calculate the mean

Selecting a column in a Data Frame

Method 1: Subset the data frame using column name:

```
df['sex']
```

Method 2: Use the column name as an attribute:

```
df.sex
```

Note: there is an attribute *rank* for pandas data frames, so to select a column with a name "rank" we should use method 1.

Hands-on exercises

- ✓ Calculate the basic statistics for the *salary* column;
- ✓ Find how many values in the *salary* column (use *count* method);
- ✓ Calculate the average salary;

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

In [ ]:
```
#Group data using rank
df_rank = df.groupby(['rank'])
```

In [ ]:
```
#Calculate mean value for each numeric column per each group
df_rank.mean()
```

rank	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Data Frames *groupby* method

Once groupby object is create we can calculate various statistics for each group:

```
In [ ]: #Calculate mean salary for each professor rank:
        df.groupby('rank')[['salary']].mean()
```

	salary
rank	
AssocProf	91786.230769
AsstProf	81362.789474
Prof	123624.804348

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

Data Frames *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the *groupby* operation. You may want to pass sort=False for potential speedup:

In [ ]:
```
#Calculate mean salary for each professor rank:
df.groupby(['rank'], sort=False)[['salary']].mean()
```

Data Frame: filtering

To subset the data, we can apply Boolean indexing. This indexing is commonly known as a filter. For example, if we want to subset the rows in which the salary value is greater than $120K:

In [ ]:
```
#Calculate mean salary for each professor rank:
df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:
> greater; >= greater or equal;
< less; <= less or equal;
== equal; != not equal;

In [ ]:
```
#Select only those rows that contain female professors:
df_f = df[ df['sex'] == 'Female' ]
```

Data Frames: Slicing

- There are a number of ways to subset the Data Frame:
  - one or more columns
  - one or more rows
  - a subset of rows and columns

  – Rows and columns can be selected by their position or label

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

In [ ]:
```
#Select column salary:
df['salary']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

In [ ]:
```
#Select column salary:
df[['rank','salary']]
```

Data Frames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

In [ ]:
```
#Select rows by their position:
df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted:
So for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

In [ ]:
```
#Select rows by their labels:
df_sub.loc[10:20,['rank','sex','salary']]
```

Out[ ]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

In [ ]:
```
#Select rows by their labels:
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

In [ ]:
```
# Create a new data frame from the original sorted by the column Salary
df_sorted = df.sort_values( by ='service')
df_sorted.head()
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Data Frames: method iloc (summary)

```
df.iloc[0]  # First row of a data frame
df.iloc[i]  #(i+1)th row
df.iloc[-1] # Last row
```

```
df.iloc[:, 0]  # First column
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]        #First 7 rows
df.iloc[:, 0:2]     #First 2 columns
df.iloc[1:3, 0:2]   #Second through third rows and first 2 columns
df.iloc[[0,5], [1,3]]  #1st and 6th rows and 2nd and 4th columns
```

Data Frames: Sorting

We can sort the data using 2 or more columns:

In [ ]:
```
df_sorted = df.sort_values( by =['service', 'salary'], ascending = [True, False])
df_sorted.head(10)
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
52	Prof	A	12	0	Female	105000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
55	AsstProf	A	2	0	Female	72500
57	AsstProf	A	3	1	Female	72500
28	AsstProf	B	7	2	Male	91300
42	AsstProf	B	4	2	Female	80225
68	AsstProf	A	4	2	Female	77500

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights = pd.read_csv("data/flights.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default (unlike R)

Aggregation Functions in Pandas

- Aggregation - computing a summary statistic about each group, i.e.
  - compute group sums or means
  - compute group sizes/counts

- Common aggregation functions:

  – min, max
  – count, sum, prod
  – mean, median, mode, mad
  – std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

In [ ]:
```
flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Graphics to explore the data

Seaborn package is built on matplotlib but provides high level interface for drawing attractive statistical graphics, similar to ggplot2 library in R. It specifically targets statistical data visualization

To show graphs within Python notebook include inline directive:

```
In [ ]: %matplotlib inline
```

Graphics

	description
distplot	histogram
barplot	estimate of central tendency for a numeric variable
violinplot	similar to boxplot, also shows the probability density of the data
jointplot	Scatterplot
regplot	Regression plot
pairplot	Pairplot
boxplot	boxplot
swarmplot	categorical scatterplot
factorplot	General categorical plot

Basic statistical Analysis

- statsmodel and scikit-learn - both have a number of function for statistical analysis
- The first one is mostly used for regular analysis using R style formulas, while scikit-learn is more tailored for Machine Learning.
- statsmodels:
  - linear regressions
  - ANOVA tests
  - hypothesis testings
  - many more ...
- scikit-learn:
  - kmeans
  - support vector machines
  - random forests
  - many more ...