Computer Vision

Pandas and data I/O

Why pandas?

- One of the most popular library that data scientists use
- Labeled axes to avoid misalignment of data
  - Data[:, 2] represents weight or weight2?
  - When merge two tables, some rows may be different
- Missing values or special values may need to be removed or replaced

	height	Weight	Weight2	age	Gender
Amy	160	125	126	32	2
Bob	170	167	155	-1	1
Chris	168	143	150	28	1
David	190	182	NA	42	1
Ella	175	133	138	23	2
Frank	172	150	148	45	1

	salary	Credit score
Alice	50000	700
Bob	NA	670
Chris	60000	NA
David	-99999	750
Ella	70000	685
Tom	45000	660

Overview

- Created by Wes McKinney in 2008, now maintained by Jeff Reback and many others.
  - Author of one of the textbooks: Python for Data Analysis
- Powerful and productive Python data analysis and Management Library
- Panel Data System
- Its an open source product.

Overview - 2

- Python Library to provide data analysis features similar to: R, MATLAB, SAS
- Rich data structures and functions to make working with data structure fast, easy and expressive.
- It is built on top of NumPy
- Key components provided by Pandas:
  - Series
  - DataFrame

From now on:

```
In [664]: from pandas import Series, DataFrame
In [665]: import pandas as pd
```

Series

- One dimensional array-like object
- It contains array of data (of any NumPy data type) with associated indexes. (Indexes can be strings or integers or other data types.)
- By default , the series will get indexing from 0 to N where N = size -1

```
In [666]: obj = Series([4, 7, -5, 3])

In [667]: obj
Out[667]:
0 4
1 7
2 -5
3 3
dtype: int64
```

```
In [668]: obj.values
Out[668]: array([ 4, 7, -5, 3], dtype=int64)

In [669]: obj.index
Out[669]: RangeIndex(start=0, stop=4, step=1)
```

Series – referencing elements

```
In [670]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [671]: obj2
Out[671]:
d 4
b 7
a -5
c 3
dtype: int64

In [672]: obj2.index
Out[672]: Index(['d', 'b', 'a', 'c'], dtype='object')

In [673]: obj2.values
Out[673]: array([ 4, 7, -5, 3], dtype=int64)
```

```
In [674]: obj2['a']
Out[674]: -5
```

```
In [818]: obj2.a
Out[818]: -5
```

```
In [675]: obj2['d']=10

In [677]: obj2[['d', 'c', 'a']]
Out[677]:
d 10
c 3
a -5
dtype: int64

In [692]: obj2[:2]
Out[692]:
d 10
b 7
dtype: int64
```

Series – array/dict operations

- numpy array operations can also be applied, which will preserve the index-value link

```
In [694]: obj2[obj2>0]
Out[694]:
d 10
b 7
c 3
dtype: int64
```

```
In [699]: obj2**2
Out[699]:
d 100
b 49
a 25
c 9
dtype: int64
```

- Can be thought of as a dict.
  Can be constructed from a dict directly.

```
In [700]: 'b' in obj2
Out[700]: True
```

```
In [702]: obj3 = Series({'a': 10, 'b': 5, 'c': 30})

In [703]: obj3
Out[703]:
a 10
b 5
c 30
dtype: int64
```

Series – null values

```
In [704]: sdata = {'Texas': 10, 'Ohio': 20, 'Oregon': 15, 'Utah': 18}

In [705]: states = ['Texas', 'Ohio', 'Oregon', 'Iowa']

In [706]: obj4 = Series(sdata, index=states)

In [707]: obj4
Out[707]:
Texas 10.0
Ohio 20.0
Oregon 15.0
Iowa NaN          Missing value
dtype: float64
```

```
In [708]: pd.isnull(obj4)
Out[708]:
Texas False
Ohio False
Oregon False
Iowa True
dtype: bool

In [709]: pd.notnull(obj4)
Out[709]:
Texas True
Ohio True
Oregon True
Iowa False
dtype: bool

In [717]: obj4[obj4.notnull()]
Out[717]:
Texas 10.0
Ohio 20.0
Oregon 15.0
dtype: float64
```

Series – auto alignment

```
In [707]: obj4
Out[707]:
Texas 10.0
Ohio 20.0
Oregon 15.0
Iowa NaN
dtype: float64
```

```
In [714]: obj5
Out[714]:
Ohio 20
Oregon 15
Texas 10
Utah 18
dtype: int64
```

```
In [715]: obj5 + obj4
Out[715]:
Iowa NaN
Ohio 40.0
Oregon 30.0
Texas 20.0
Utah NaN
dtype: float64
```

Series name and index name

```
In [720]: obj4.name = 'population'

In [721]: obj4
Out[721]:
Texas 10.0
Ohio 20.0
Oregon 15.0
Iowa NaN
Name: population, dtype: float64
```

- Index of a series can be changed to a different index.
- Index object itself is immutable.

```
In [1014]: obj4.index[2]='California'
TypeError: Index does not support mutable operations

In [1016]: obj4.index
Out[1016]: Index(['Florida', 'New York', 'Kentucky', 'Georgia'], dtype='object')
```

```
In [722]: obj4.index.name = 'state'

In [723]: obj4
Out[723]:
state
Texas 10.0
Ohio 20.0
Oregon 15.0
Iowa NaN
Name: population, dtype: float64
```

```
In [725]: obj4.index = ['Florida', 'New York', 'Kentucky', 'Georgia']

In [726]: obj4
Out[726]:
Florida 10.0
New York 20.0
Kentucky 15.0
Georgia NaN
Name: population, dtype: float64
```

DataFrame

- A DataFrame is a tabular data structure comprised of rows and columns, akin to a spreadsheet or database table.
- It can be treated as an order collection of columns
  - Each column can be a different data type
  - Have both row and column indices

```
In [727]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
   ...: 'year': [2000, 2001, 2002, 2001, 2002],
   ...: 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

In [728]: frame = DataFrame(data)

In [729]: frame
Out[729]:
  pop state year
0 1.5 Ohio 2000
1 1.7 Ohio 2001
2 3.6 Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
```

reordered

DataFrame – specifying columns and indices

```
In [727]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada',
'Nevada'],
   ...: 'year': [2000, 2001, 2002, 2001, 2002],
   ...: 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

In [730]: frame2 = DataFrame(data, columns=['year', 'state',
'pop', 'debt'], index=['A', 'B', 'C', 'D', 'E'])

In [731]: frame2
Out[731]:
   year state pop debt
A 2000 Ohio 1.5 NaN
B 2001 Ohio 1.7 NaN
C 2002 Ohio 3.6 NaN
D 2001 Nevada 2.4 NaN
E 2002 Nevada 2.9 NaN
```

Same order

Initialized with NaN

- Order of columns/rows can be specified.
- Columns not in data will have NaN.

DataFrame – from nested dict of dicts

- Outer dict keys as columns and inner dict keys as row indices

```
In [838]: pop = {'Nevada': {2001: 2.9, 2002: 2.9}, 'Ohio': {2002: 3.6, 2001: 1.7, 2000: 1.5}}

In [840]: frame3 = DataFrame(pop)

In [841]: frame3
Out[841]:
      Nevada  Ohio
2000  NaN     1.5
2001  2.9     1.7
2002  2.9     3.6
```

Union of inner keys (in sorted order)

Transpose

```
In [842]: frame3.T
Out[842]:
        2000  2001  2002
Nevada  NaN   2.9   2.9
Ohio    1.5   1.7   3.6
```

DataFrame – index, columns, values

```
In [847]: frame3.index
Out[847]: Int64Index([2000, 2001, 2002], dtype='int64')

In [848]: frame3.columns
Out[848]: Index(['Nevada', 'Ohio'], dtype='object')

In [849]: frame3.values
Out[849]:
array([    [ nan, 1.5],
           [ 2.9, 1.7],
           [ 2.9, 3.6]])
```

```
In [850]: frame3.index.name = 'year';
frame3.columns.name='state'

In [851]: frame3
Out[851]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.9 1.7
2002 2.9 3.6
```

(Personal opinion) Bad design: index should be called row label, column should be called column label. Index can be label-based or position-based.

Possible data inputs to DataFrame constructor

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

Indexing, selection and filtering

- Series and DataFrame can be sliced/accessed with label-based indexes, or using position-based indexes similar to Numpy Array

```
In [906]: S = Series(range(4), index=['zero', 'one', 'two', 'three'])
```

```
In [907]: S['two']
Out[907]: 2
```

```
In [908]: S[['zero', 'two']]
Out[908]:
zero 0
two 2
dtype: int32
```

```
In [909]: S[2]
Out[909]: 2
```

```
In [910]: S[[0,2]]
Out[910]:
zero 0
two 2
dtype: int32
```

```
In [911]: S[:2]
Out[911]:
zero 0
one 1
dtype: int32
```

```
In [913]: S['zero':'two']
Out[913]:
zero 0
one 1
two 2
dtype: int32
```

Inclusive

```
In [917]: S[S > 1]
Out[917]:
two 2
three 3
dtype: int32
```

```
In [995]: S[-2:]
Out[995]:
two 2
three 3
dtype: int32
```

DataFrame – retrieving a column

- A column in a DataFrame can be retrieved as a Series by dict-like notation or as attribute
- Series index and name have been kept/set appropriately

```
In [734]: frame['state']
Out[734]:
0 Ohio
1 Ohio
2 Ohio
3 Nevada
4 Nevada
Name: state, dtype: object
```

```
In [733]: frame.state
Out[733]:
0 Ohio
1 Ohio
2 Ohio
3 Nevada
4 Nevada
Name: state, dtype: object
```

```
In [805]: type(frame['state'])
Out[805]: pandas.core.series.Series
```

DataFrame – getting rows

- loc for using indexes and iloc for using positions

```
In [792]: frame2
Out[792]:
   year state pop debt
A 2000 Ohio 1.5 NaN
B 2001 Ohio 1.7 NaN
C 2002 Ohio 3.6 NaN
D 2001 Nevada 2.4 NaN
E 2002 Nevada 2.9 NaN
```

```
In [801]: frame2.loc['A']
Out[801]:
year 2000
state Ohio
pop 1.5
debt NaN
Name: A, dtype: object
```

```
In [804]: type(frame2.loc['A'])
Out[804]: pandas.core.series.Series
```

```
In [819]: frame2.loc[['A', 'B']]
Out[819]:
   year state pop debt
A 2000 Ohio 1.5 NaN
B 2001 Ohio 1.7 NaN
```

```
In [820]: type(frame2.loc[['A', 'B']])
Out[820]: pandas.core.frame.DataFrame
```

DataFrame – modifying columns

```
In [829]: frame2['debt'] = 0

In [830]: frame2
Out[830]:
   year state pop debt
A 2000 Ohio 1.5 0
B 2001 Ohio 1.7 0
C 2002 Ohio 3.6 0
D 2001 Nevada 2.4 0
E 2002 Nevada 2.9 0
```

```
In [831]: frame2['debt'] = range(5)

In [832]: frame2
Out[832]:
   year state pop debt
A 2000 Ohio 1.5 0
B 2001 Ohio 1.7 1
C 2002 Ohio 3.6 2
D 2001 Nevada 2.4 3
E 2002 Nevada 2.9 4
```

```
In [833]: val = Series([10, 10, 10],
index = ['A', 'C', 'D'])

In [834]: frame2['debt'] = val

In [835]: frame2
Out[835]:
   year state pop debt
A 2000 Ohio 1.5 10.0
B 2001 Ohio 1.7 NaN
C 2002 Ohio 3.6 10.0
D 2001 Nevada 2.4 10.0
E 2002 Nevada 2.9 NaN
```

Rows or individual elements can be modified similarly. Using loc or iloc.

DataFrame – removing columns

```
In [836]: del frame2['debt']

In [837]: frame2
Out[837]:
     year  state   pop
A    2000  Ohio    1.5
B    2001  Ohio    1.7
C    2002  Ohio    3.6
D    2001  Nevada  2.4
E    2002  Nevada  2.9
```

More on DataFrame indexing

```
In [855]: data = np.arange(9).reshape(3,-1)

In [856]: data
Out[856]:
array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])

In [868]: frame = DataFrame(data,
index=['r1', 'r2', 'r3'],
columns=['c1', 'c2', 'c3'])

In [869]: frame
Out[869]:
c1 c2 c3
r1 0 1 2
r2 3 4 5
r3 6 7 8
```

```
In [870]: frame['c1']
Out[870]:
r1 0
r2 3
r3 6
Name: c1, dtype: int32

In [878]: frame.loc['r1']
Out[878]:
c1 0
c2 1
c3 2
Name: r1, dtype: int32

In [952]: frame['c1']['r1']
Out[952]: 0
```

```
In [871]: frame[['c1', 'c3']]
Out[871]:
c1 c3
r1 0 2
r2 3 5
r3 6 8

In [879]: frame.loc[['r1','r3']]
Out[879]:
c1 c2 c3
r1 0 1 2
r3 6 7 8

In [885]: frame.iloc[:2]
Out[885]:
c1 c2 c3
r1 0 1 2
r2 3 4 5
```

Row slices

```
In [954]: frame[:2]
Out[954]:
c1 c2 c3
r1 0 1 2
r2 3 4 5
```

Row slices

More on DataFrame indexing - 2

```
In [1027]: frame.loc[['r1', 'r2'], ['c1', 'c2']]
Out[1027]:
    c1 c2
r1 0 1
r2 3 4

In [1034]: frame.loc['r1':'r3', 'c1':'c3']
Out[1034]:
    c1 c2 c3
r1 0 1 2
r2 3 4 5
r3 6 7 8

In [1036]: frame.iloc[:2,:2]
Out[1036]:
    c1 c2
r1 0 1
r2 3 4
```

```
In [1140]: v =
DataFrame(np.arange(9).reshape(3,3),
index=['a', 'a', 'b'], columns=['c1','c2','c3'])
```

Duplicated keys

```
In [1141]: v
Out[1141]:
c1 c2 c3
a 0 1 2
a 3 4 5
b 6 7 8

In [1142]: v.loc['a']
Out[1142]:
c1 c2 c3
a 0 1 2
a 3 4 5
```

More on DataFrame indexing - 3

```
In [980]: frame
Out[980]:
c1 c2 c3
r1 0 1 2
r2 3 4 5
r3 6 7 8

In [981]: frame[frame['c1']>0]
Out[981]:
c1 c2 c3
r2 3 4 5
r3 6 7 8

In [982]: frame['c1']>0
Out[982]:
r1 False
r2 True
r3 True
Name: c1, dtype: bool
```

```
In [1038]: frame < 3
Out[1038]:
c1 c2 c3
r1 True True True
r2 False False False
r3 False False False

In [987]: frame[frame<3] = 3

In [988]: frame
Out[988]:
c1 c2 c3
r1 3 3 3
r2 3 4 5
r3 6 7 8
```

Removing rows/columns

This returns a new object (MATLAB-like).

```
In [899]: frame.drop(['r1'])
Out[899]:
    c1 c2 c3
r2 3 4 5
r3 6 7 8

In [900]: frame.drop(['r1','r3'])
Out[900]:
    c1 c2 c3
r2 3 4 5

In [901]: frame.drop(['c1'], axis=1)
Out[901]:
    c2 c3
r1 1 2
r2 4 5
r3 7 8
```

```
In [1050]: frame
Out[1050]:
    c1 c2 c3
r1 0 1 2
r2 3 10 5
r3 6 7 8
```

Reindexing

- Alter the order of rows/columns of a DataFrame or order of a series according to new index

```
In [887]: frame
Out[887]:
    c1 c2 c3
r1 0 1 2
r2 3 4 5
r3 6 7 8

In [888]: frame.reindex(['r1', 'r3', 'r2', 'r4'])
Out[889]:
    c1 c2 c3
r1 0.0 1.0 2.0
r3 6.0 7.0 8.0
r2 3.0 4.0 5.0
r4 NaN NaN NaN
```

```
In [892]: frame.reindex(columns=['c2', 'c3', 'c1'])
Out[892]:
    c2 c3 c1
r1 1 2 0
r2 4 5 3
r3 7 8 6
```

This returns a new object (MATLAB-like).

Function application and mapping

- DataFrame.applymap(f) applies f to every entry
- DataFrame.apply(f) applies f to every column (default) or row

```
In [1087]: frame
Out[1087]:
c1 c2 c3
r1 0 1 2
r2 3 4 5
r3 6 7 8
```

```
In [1077]: def square(x): return x**2
In [1078]: frame.applymap(square)
Out[1078]:
c1 c2 c3
r1 0 1 4
r2 9 16 25
r3 36 49 64
```

```
In [1084]: def max_minus_min(x): return max(x)-min(x)
In [1085]: frame.apply(max_minus_min)
Out[1085]:
c1 6
c2 6
c3 6
dtype: int64
```

```
In [1086]: frame.apply(max_minus_min, axis=1)
Out[1086]:
r1 2
r2 2
r3 2
dtype: int64
```

Function application and mapping - 2

In [**1088**]: def max_min(x): return Series([max(x), min(x)], index=['max', 'min'])

In [**1089**]: frame.apply(max_min)
Out[**1089**]:
c1 c2 c3
max 6 7 8
min 0 1 2

Other DataFrame functions

- sort_index()

In [1090]: frame.index=['A', 'C', 'B']; frame.columns=['b','a','c'];

In [1091]: frame.sort_index()
Out[1091]:
b a c
A 0 1 2
B 6 7 8
C 3 4 5

In [1092]: frame.sort_index(axis=1)
Out[1092]:
a b c
A 1 0 2
C 4 3 5
B 7 6 8

- sort_values()

In [1094]: frame = DataFrame(np.random.randint(0, 10, 9).reshape(3,-1), index=['r1', 'r2', 'r3'], columns=['c1', 'c2', 'c3'])

In [1095]: frame
Out[1095]:
c1 c2 c3
r1 8 3 9
r2 2 5 0
r3 4 4 8

In [1101]: frame.sort_values(by='c1')
Out[1101]:
c1 c2 c3
r2 2 5 0
r3 4 4 8
r1 8 3 9

In [1102]: frame.sort_values(axis=1, by=['r3','r1'])
Out[1102]:
c2 c1 c3
r1 3 8 9
r2 5 2 0
r3 4 4 8

Other DataFrame functions - 2

- Rank()

```
In [1106]: frame
Out[1106]:
    c1 c2 c3
r1 8 3 9
r2 2 5 0
r3 4 4 8

In [1107]: frame.rank(axis=1)
Out[1107]:
    c1 c2 c3
r1 2.0 1.0 3.0
r2 2.0 3.0 1.0
r3 1.5 1.5 3.0
```

Frame[‘c1’][‘r1’] is the second smallest in r1
Frame[‘c1’][‘r3’] and Frame[‘c2’][‘r3’] is tied for the smallest in r3

Other DataFrame functions

- mean()
  - Mean(axis=0, skipna=True)
- sum()
- cumsum()
- describe(): return summary statistics of each column
  - for numeric data: mean, std, max, min, 25%, 50%, 75%, etc.
  - For non-numeric data: count, uniq, most-frequent item, etc.
- corr(): correlation between two Series, or between columns of a DataFrame
- corr_with(): correlation between columns of DataFram and a series or between the columns of another DataFrame

Handling missing data

- Filtering out missing values

```
In [1198]: from numpy import nan as NaN

In [1199]: data = Series([1, NaN, 2.5, NaN, 6])

In [1200]: data.dropna()
Out[1200]:
0 1.0
2 2.5
4 6.0
dtype: float64

In [1201]: data
Out[1201]:
0 1.0
1 NaN
2 2.5
3 NaN
4 6.0
dtype: float64
```

```
In [1204]: data.notnull()
Out[1204]:
0 True
1 False
2 True
3 False
4 True
dtype: bool

In [1205]: data[data.notnull()]
Out[1205]:
0 1.0
2 2.5
4 6.0
dtype: float64
```

Handling missing data - 2

```
In [1206]: data = DataFrame([[1, 2, 3],
[1, NaN, NaN], [NaN, NaN, NaN],
[NaN, 4, 5]])

In [1207]: data
Out[1207]:
0 1 2
0 1.0 2.0 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 4.0 5.0

In [1208]: data.dropna()
Out[1208]:
0 1 2
0 1.0 2.0 3.0
```

```
In [1209]: data.dropna(how='all')
Out[1209]:
0 1 2
0 1.0 2.0 3.0
1 1.0 NaN NaN
3 NaN 4.0 5.0

In [1210]: data.dropna(axis=1, how='all')
Out[1210]:
0 1 2
0 1.0 2.0 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 4.0 5.0
```

```
In [1215]: data[4]=NaN

In [1216]: data
Out[1216]:
0 1 2 4
0 1.0 2.0 3.0 NaN
1 1.0 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 4.0 5.0 NaN

In [1217]: data.dropna(axis=1,
how='all')
Out[1217]:
0 1 2
0 1.0 2.0 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 4.0 5.0
```

Filling in missing data

```
In [1218]: data
Out[1218]:
0 1 2 4
0 1.0 2.0 3.0 NaN
1 1.0 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 4.0 5.0 NaN

In [1219]: data.fillna(0)
Out[1219]:
0 1 2 4
0 1.0 2.0 3.0 0.0
1 1.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0
3 0.0 4.0 5.0 0.0
```

Modify the dataframe instead of retunring a new object (default)

```
In [1220]: data.fillna(0, inplace=True)

In [1221]: data
Out[1221]:
0 1 2 4
0 1.0 2.0 3.0 0.0
1 1.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0
3 0.0 4.0 5.0 0.0
```

```
In [1227]: data
Out[1227]:
0 1 2
0 NaN 9 9.0
1 NaN 7 2.0
2 4.0 8 9.0
3 3.0 4 NaN
```

replace nan with column mean

```
In [1228]:
data.fillna(data.mean(skipna=True))
Out[1228]:
0 1 2
0 3.5 9 9.000000
1 3.5 7 2.000000
2 4.0 8 9.000000
3 3.0 4 6.666667
```

Hierarchical indexing

```
In [1229]: dataSeries = Series(np.arange(10),
index=[['a']*3+['b']*3+['c']*4, [‘i’,’ii’,’iii’]*3+[‘iv’]])

In [1230]: dataSeries
Out[1230]:
a    i     0
     ii    1
     iii   2
b    i     3
     ii    4
     iii   5
c    i     6
     ii    7
     iii   8
     iv    9
dtype: int32
```

MultiIndex

```
In [1240]: dataSeries.index
Out[1240]:
MultiIndex(levels=[['a', 'b', 'c'], ['i', 'ii', 'iii', 'iv']],
labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2, 3]])
```

```
In [1242]: dataSeries['b']
Out[1242]:
i 3
ii 4
iii 5
dtype: int32
```

```
In [1243]: dataSeries[:, 'ii']
Out[1243]:
a 1
b 4
c 7
dtype: int32
```

Hierarchical indexing and DataFrame

- Unstack and stack

```
In [1245]: dataSeries.unstack()
Out[1245]:
     i    ii   iii   iv
a  0.0  1.0  2.0  NaN
b  3.0  4.0  5.0  NaN
c  6.0  7.0  8.0  9.0
```

```
In [1246]: dataSeries.unstack().T.stack()
Out[1246]:
i    a    0.0
     b    3.0
     c    6.0
ii   a    1.0
     b    4.0
     c    7.0
iii  a    2.0
     b    5.0
     c    8.0
iv   c    9.0
dtype: float64
```

Hierarchical indexing for DataFrame

```
In [1256]: frame2 = DataFrame(np.arange(16).reshape(4,4), index=[['a', 'a', 'b', 'b'], ['i','ii']*2],
columns=[['c1', 'c1', 'c2', 'c2'], ['.1', '.2']*2])
```

```
In [1257]: frame2
Out[1257]:
      c1      c2
     .1  .2  .1  .2
a i   0   1   2   3
  ii  4   5   6   7
b i   8   9  10  11
  ii 12  13  14  15
```

```
In [1274]: frame2.swaplevel(-2, -1)
Out[1274]:
      c1      c2
     .1  .2  .1  .2
i  a  0   1   2   3
ii a  4   5   6   7
i  b  8   9  10  11
ii b 12  13  14  15
```

```
In [1275]: frame2.swaplevel(-2, -1, axis=1)
Out[1275]:
     .1  .2  .1  .2
     c1  c1  c2  c2
a i   0   1   2   3
  ii  4   5   6   7
b i   8   9  10  11
  ii 12  13  14  15
```

Use DataFrame columns as indices

- set_index

```
In [1281]: df = DataFrame({'a':range(7),
'b':range(7,0,-1), 'c':['one']*3+['two']*4,
'd':[0,1,2]*2+[3]})

In [1282]: df
Out[1282]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

```
In [1283]: df2=df.set_index(['c', 'a'])

In [1284]: df2
Out[1284]:
       b  d
c   a
one 0  7  0
    1  6  1
    2  5  2
two 3  4  0
    4  3  1
    5  2  2
    6  1  3
```

```
In [1285]: df2.loc['one']
Out[1285]:
   b  d
a
0  7  0
1  6  1
2  5  2
```

Data loading, storage and file formats

- We’ll mainly talk about pandas data input/output
- Other options are available

Examples are available at:

http://cs.utsa.edu/~jruan/cs5163f17/ch06.ipynb.zip

Text format

- read_csv
- read_table

Essentially the same. Use different delimiter by default, but can supply delimiter as a parameter.

Table 6-1. Parsing functions in pandas

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
read_table	Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter
read_fwf	Read data in fixed-width column format (that is, no delimiters)
read_clipboard	Version of read_table that reads data from the clipboard. Useful for converting tables from web pages

Features

- Indexing: can treat one or more columns as indexes of the returned DataFrame, and whether to get column names from the file, the user or not at all
- **Type inference** and data conversion. Includes user-defined value conversion and custom list of missing value markers
  - No need to specify between float, int, str, and bool
- Datetime parsing. Combining date and time info from multiple columns into a single column.
- Iterating: support for iterating over chunks of very large files.
- Unclean data issue: skipping header rows or footer, comments, etc.

Table 6-2. read_csv /read_table function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row
`index_col`	Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result, combine with `header`=None
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
`na_values`	Sequence of values to replace with NA
`comment`	Character or characters to split comments off the end of lines
`parse_dates`	Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns)

keep_date_col	If joining columns to parse date, drop the joined columns. Default True
converters	Dict containing column number of name mapping to functions. For example {'foo': f} would apply the function f to all values in the 'foo' column
dayfirst	When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False
date_parser	Function to use to parse dates
nrows	Number of rows to read from beginning of file
iterator	Return a TextParser object for reading file piecemeal
chunksize	For iteration, size of file chunks
skip_footer	Number of lines to ignore at end of file
verbose	Print various parser output information, like the number of missing values placed in non-numeric columns
encoding	Text encoding for unicode. For example 'utf-8' for UTF-8 encoded text
squeeze	If the parsed data only contains one column return a Series
thousands	Separator for thousands, e.g. ',' or '.'

Examples

Demo using jupyter notebook

Most examples are taken from:

https://github.com/wesm/pydata-book

ch06.ipynb

Writing data to text format

- to_csv(path)

```
Signature: to_csv(path_or_buf=None, sep=',', na_rep='',
float_format=None, columns=None, header=True,
index=True, index_label=None, mode='w',
encoding=None, compression=None, quoting=None,
quotechar='"', line_terminator='\n', chunksize=None,
tupleize_cols=False, date_format=None,
doublequote=True, escapechar=None, decimal='.')
```

JSON format

- JSON: JavaScript Object Notation
  - One of the standard formats for sending data by HTTP requests
  - Eg:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30,
                                        "pets": ["Zeus", "Zuko"]},
                         {"name": "Katie", "age": 38,
                                        "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

- Very similar to python syntax. However, strings must be enclosed in “double quotes” instead of ‘single quotes’.
- Can have dicts, lists, strings, numbers, booleans, and nulls
- json.loads() converts a json-format string to a python object (e.g, dict or list)
- json.dump() converts a python object to a json-format string
- pandas.read_json() read json format file to DataFrame
- data.to_json(): converts a DataFrame to a json string

BeautifulSoup html parser

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://en.wikipedia.org/wiki/Main_Page").text
soup = BeautifulSoup(html, 'html5lib')
for anchor in soup.find_all('a'):
   print(anchor.get('href', '/'))
```

http://cs.utsa.edu/~jruan/cs5163f17/ch06.ipynb

More examples on DSS Ch 9 page 108-113

XML and HTML parsing

- lxml library
- lxml.html for html
- lxml.objectify for xml

- pandas.read_html(path): read html tables into a list of DataFrames

- http://cs.utsa.edu/~jruan/cs5163f17/ch06.ipynb.zip

- More examples on PDA (page 166-170)

Binary data format

- “pickle” format
  - dataframe.to_pickle(path) saves a DataFrame into binary format
  - pandas.read_pickle(picklefile) reads a pickle file into a DataFrame
- HDF5 format
  - store = pd.HDFStore(path)
  - store[‘key’] = obj
  - store.close() # save objects into file
  - store.open()
  - store.select(objName, start=0, stop = n)
  - pd.read_hdf(path, objName, start=0, stop = n)

Interacting with Database

- sqlite3 to create a light-weight database
- sqlalchemy to access database and retrieve records as python objects
- pandas.read_sql to read table into DataFrame

Data transformation and normalization

- Use boxplot to take a quick book
- Transform data to obtain a certain distribution
  - e.g. from lognormal to normal
  - Normalize data so different columns became comparable / compatible
- Typical normalization approach:
  - Z-score transformation
  - Scale to between 0 and 1
  - Trimmed mean normalization
  - Vector length transformation
  - Quantilenorm

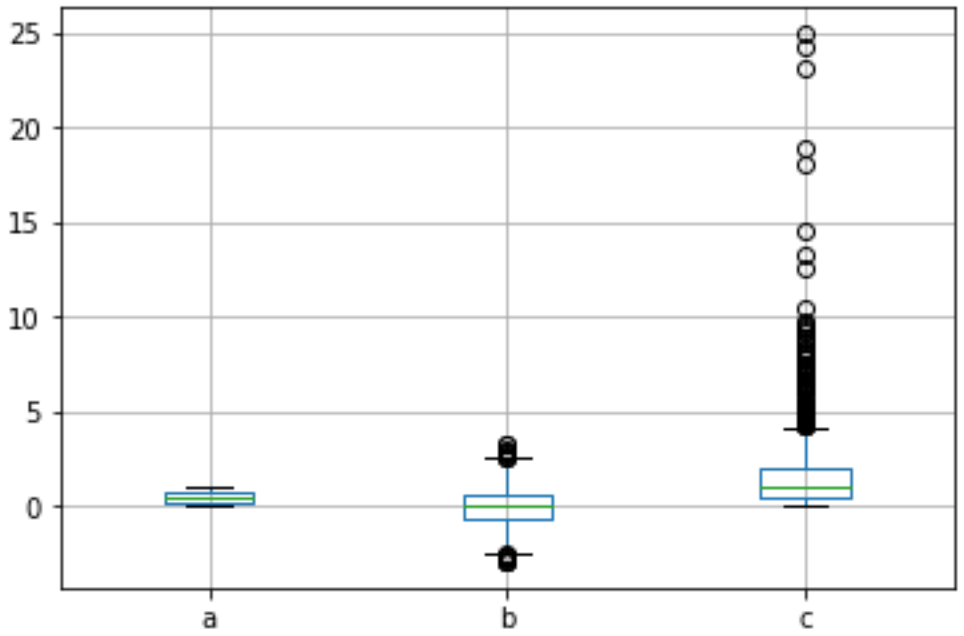
Boxplot example

In [**1867**]: df=DataFrame({'a': np.random.rand(1000),
'b': np.random.randn(1000,),
'c': np.random.lognormal(size=(1000,))})

In [**1868**]: df.head()
Out[**1868**]:

	a	b	c
0	0.356627	1.406655	3.288161
1	0.472792	-1.247858	2.499727
2	0.467848	0.406503	2.215045
3	0.341257	1.457440	0.390666
4	0.236013	0.026771	1.295106

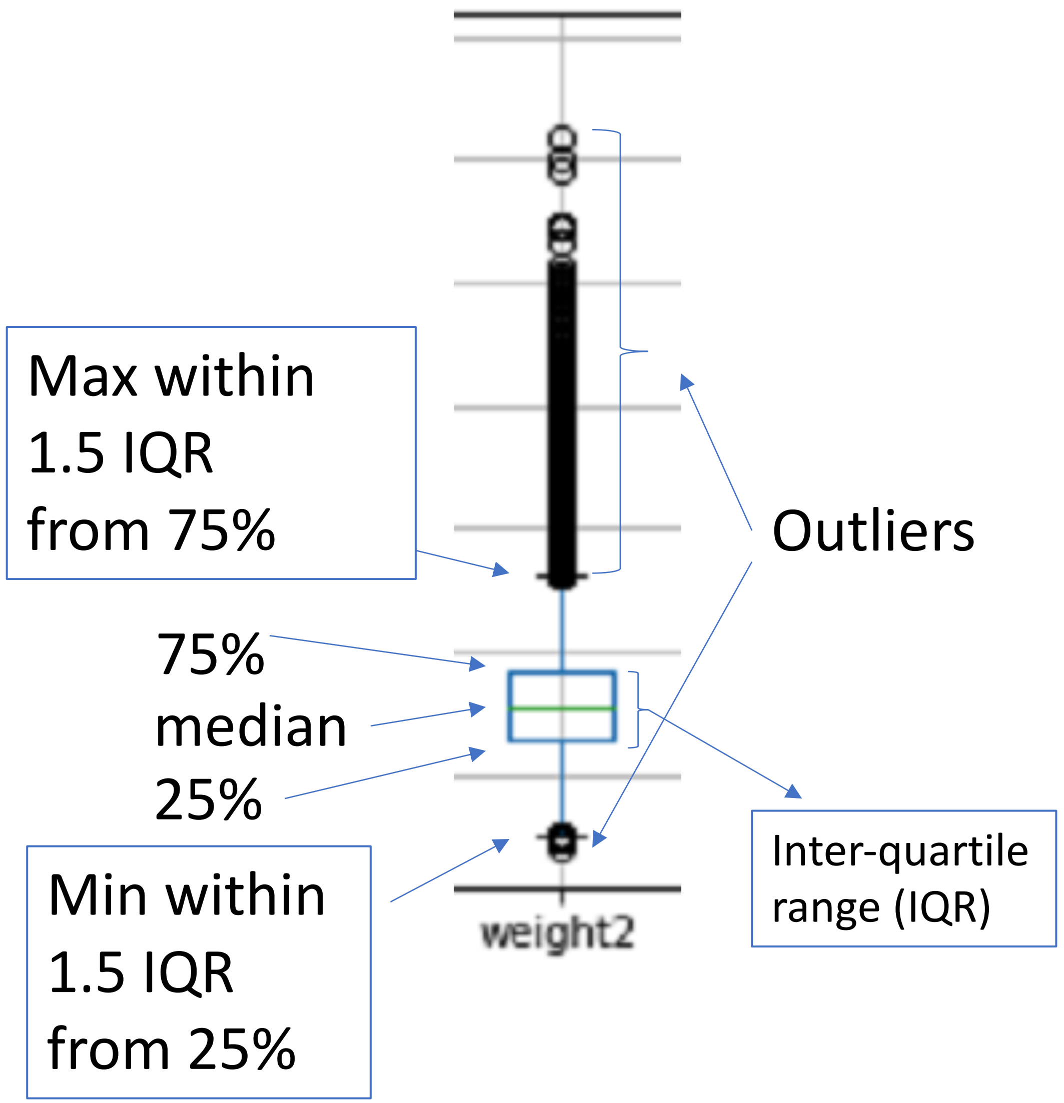
In [**1869**]: df.boxplot()
Out[**1869**]: <matplotlib.axes._subplots.AxesSubplot at 0x4dfb0f28>

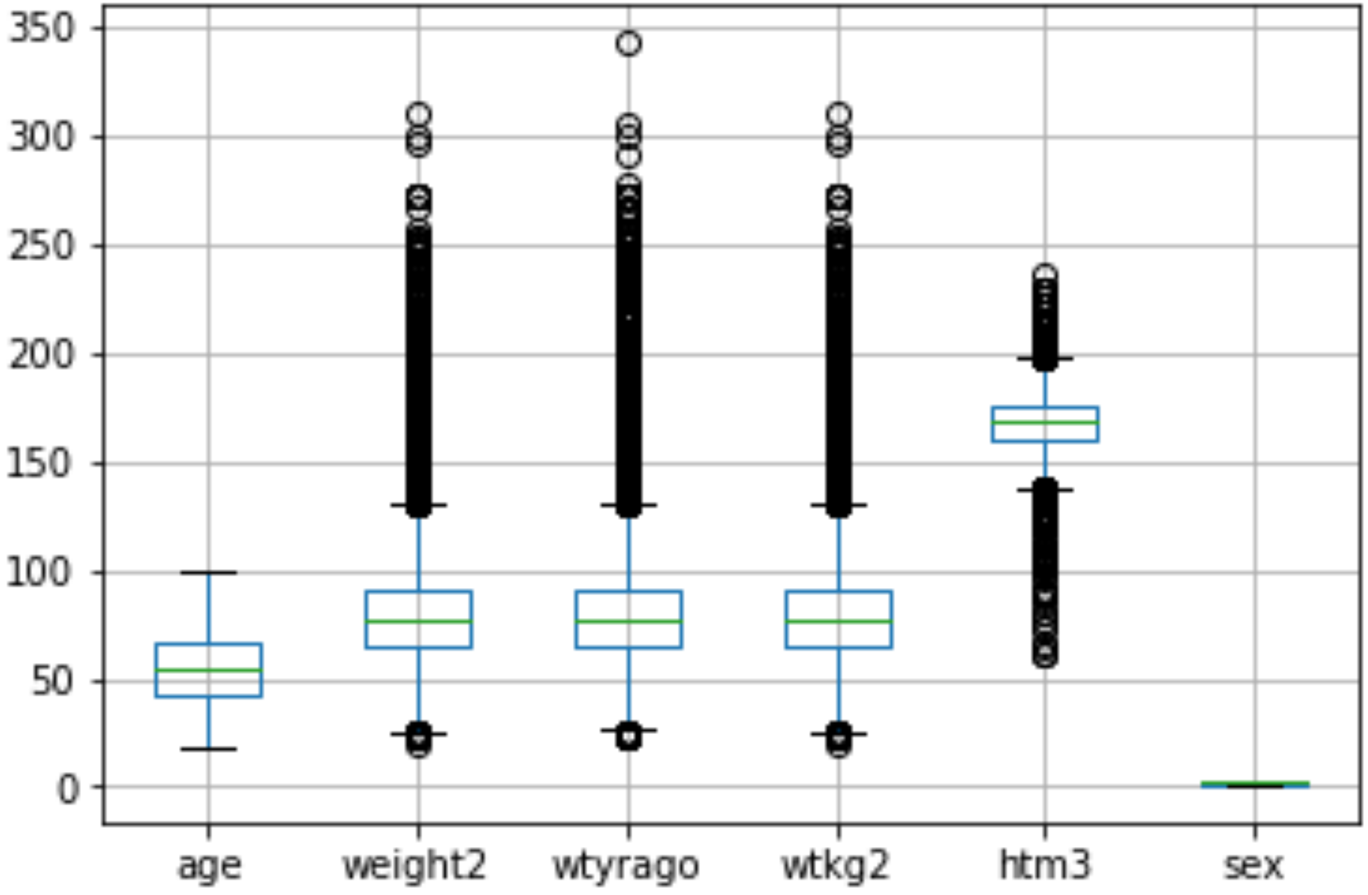


Boxplot example 2

In [1876]: df2 = pd.read_csv('brfss.csv', index_col=0)

In [1877]: df2.boxplot()
Out[1877]: <matplotlib.axes._subplots.AxesSubplot at 0x4ebcf588>



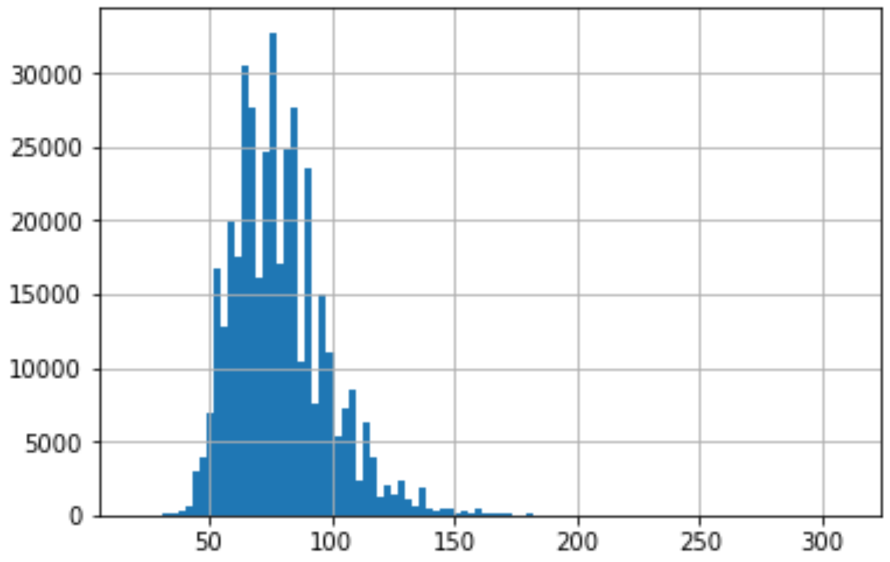


Other useful pandas plotting functions

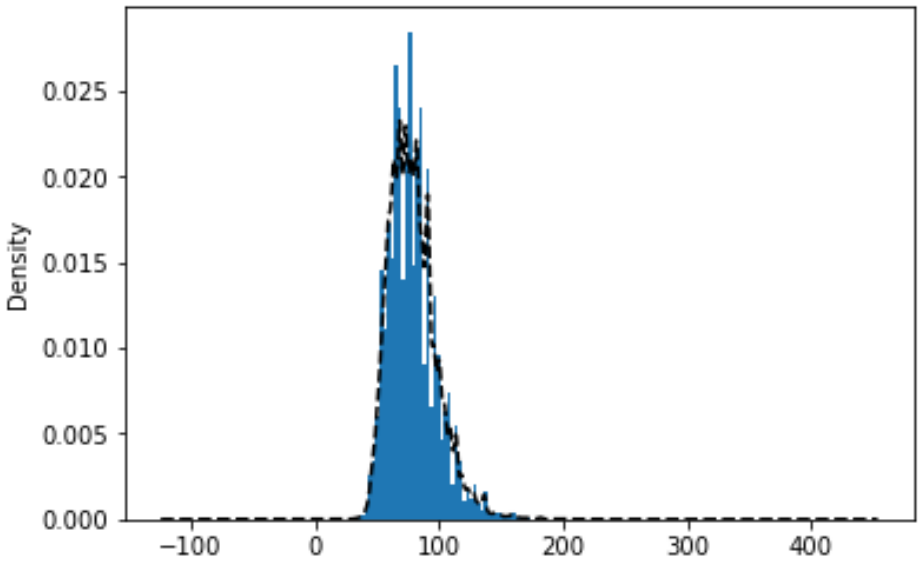
- hist, plot, scatter, etc.

Use kernel density estimate to approximate
the distribution with a mixture of normal distributions

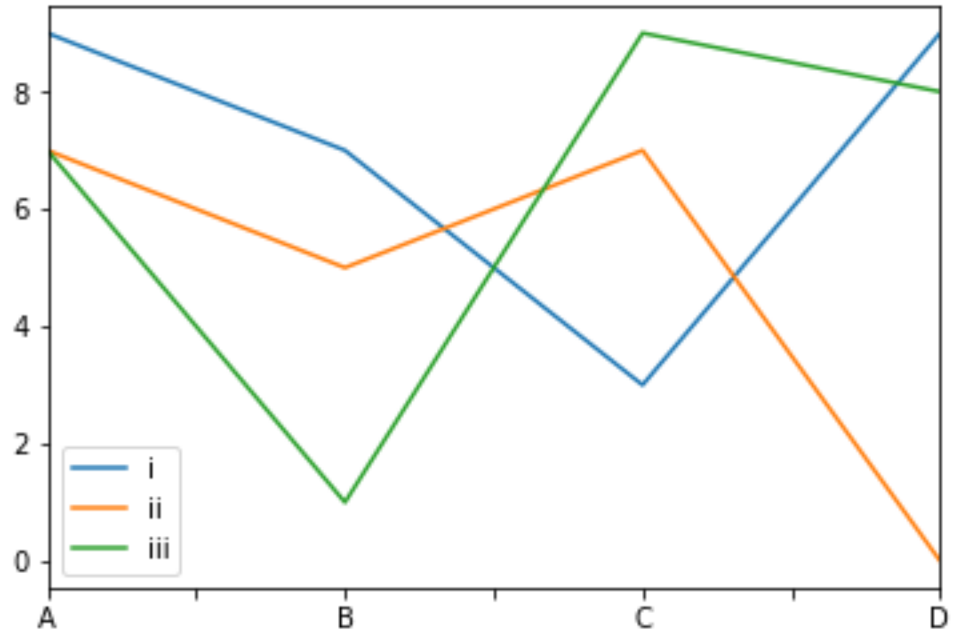
```
In [1891]: df2['weight2'].hist(bins=100)
Out[1891]: <matplotlib.axes._subplots.AxesSubplot at
0x52197fd0>
```



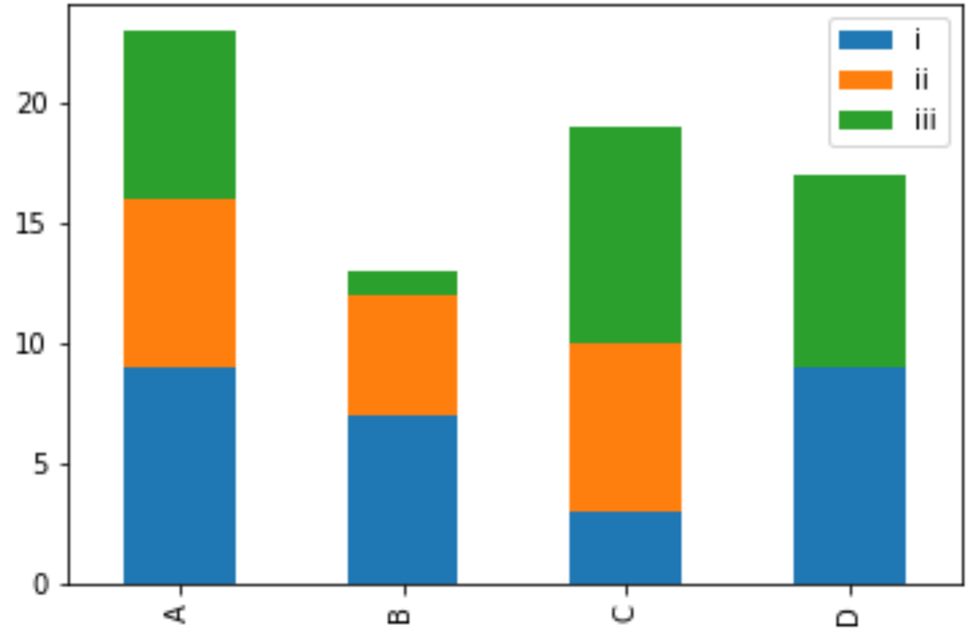

```
In [1893]: df2['weight2'].hist(bins=100,normed=True);
df2['weight2'].plot(kind='kde', style='k--')
Out[1893]: <matplotlib.axes._subplots.AxesSubplot at
0x53ddc828>
```



```
In [1911]: df3 = DataFrame(np.random.randint(0, 10, (4, 3)),
index=['A', 'B', 'C', 'D'], columns=['i', 'ii', 'iii'])

In [1912]: df3.plot()
Out[1912]: <matplotlib.axes._subplots.AxesSubplot at
0x519a07b8>
```




```
In [1913]: df3.plot(kind='bar', stacked=True)
Out[1913]: <matplotlib.axes._subplots.AxesSubplot at
0x51afad68>
```



Why normalization (re-scaling)

	Height (inches)	Heights (feet)	Heights (cm)	Weight (LB)
A	63	5.25	160.0	150
B	64	5.33	162.6	155
C	72	6.00	182.9	156

In [1961]: def distance(ser1, ser2): return ((ser1-ser2)**2).sum()**0.5

A-B

In [1963]:
distance(df6.loc['A',['foot','lb']],df6.loc['B',['foot','lb']])
Out[1963]: 5.000639959045242

A-C

In [1964]:
distance(df6.loc['A',['foot','lb']],df6.loc['C',['foot','lb']])
Out[1964]: 6.046693311223912

B-C

In [1965]:
distance(df6.loc['B',['foot','lb']],df6.loc['C',['foot','lb']])
Out[1965]: 1.2037026210821342

A-B

In [1958]:
distance(df6.loc['A',['inch','lb']],df6.loc['B',['inch','lb']]
Out[1958]: 5.0990195135927845

A-C

In [1959]:
distance(df6.loc['A',['inch','lb']],df6.loc['C',['inch','lb']]
Out[1959]: 10.816653826391969

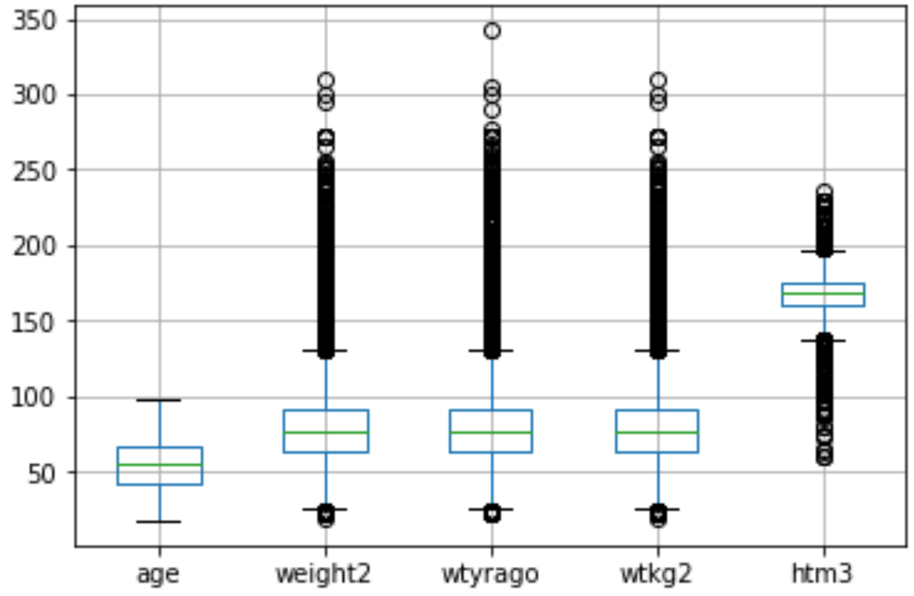
B-C

In [1960]:
distance(df6.loc['B',['inch','lb']],df6.loc['C',['inch','lb']]
Out[1960]: 8.06225774829855

Z-score transformation

In [1929]: df4 = df.drop('sex', axis=1)

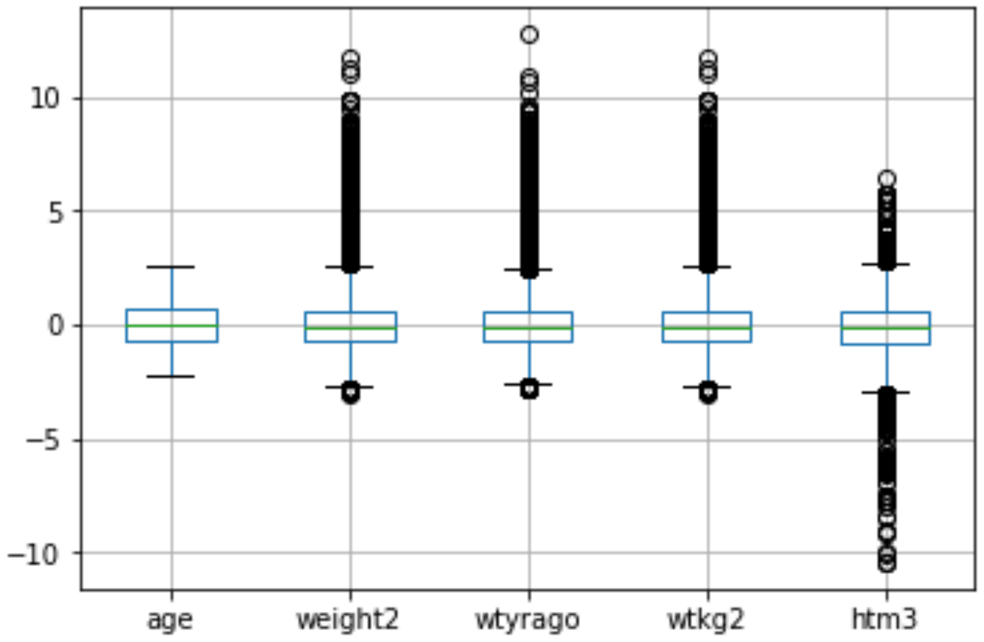
In [1930]: df4.boxplot()
Out[1930]: <matplotlib.axes._subplots.AxesSubplot at 0x51e9cb00>



In [1931]: def zscore(series): return (series - series.mean(skipna=True)) / series.std(skipna=True);

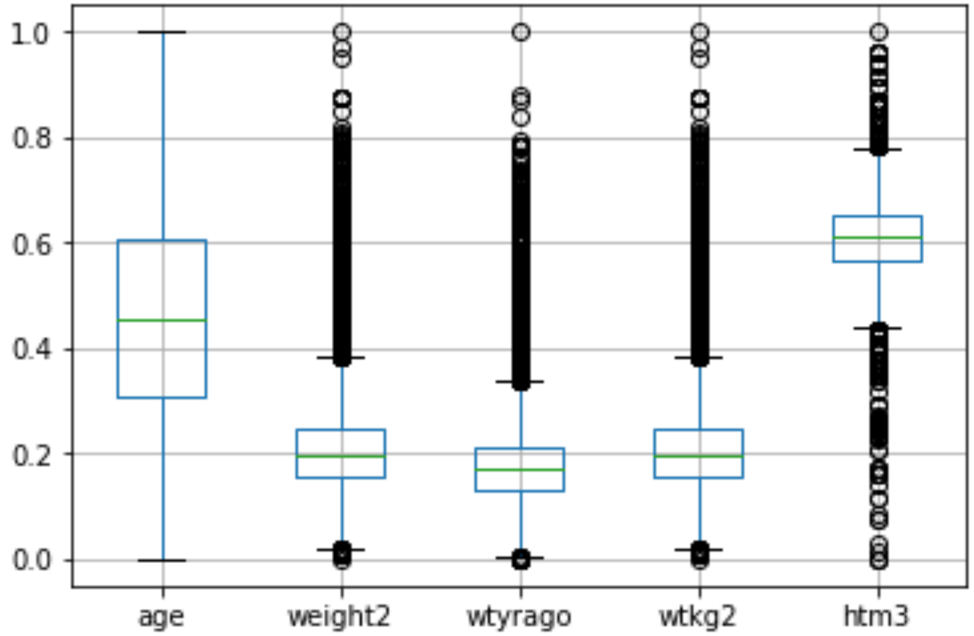
In [1932]: df5 = df4.apply(zscore)

In [1933]: df5.boxplot()
Out[1933]: <matplotlib.axes._subplots.AxesSubplot at 0x51e52ac8>



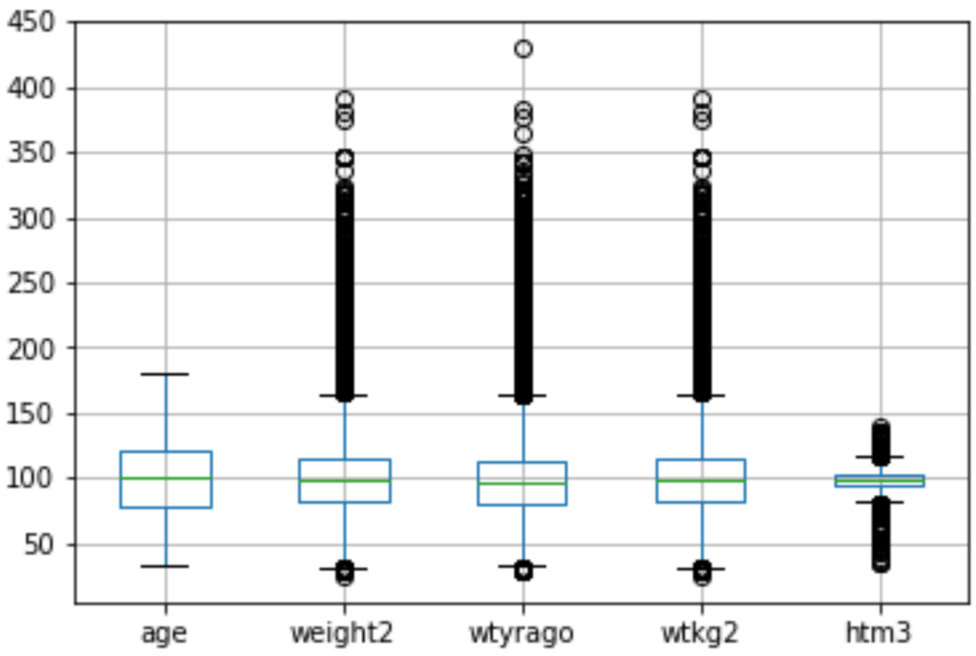
Scaling to between 0 and 1

```
In [2181]: def scaling(series):
       return (series - series.min()) / (series.max() - series.min())

In [2182]: df7 = df4.apply(scaling)

In [2183]: boxplot(df7)
```



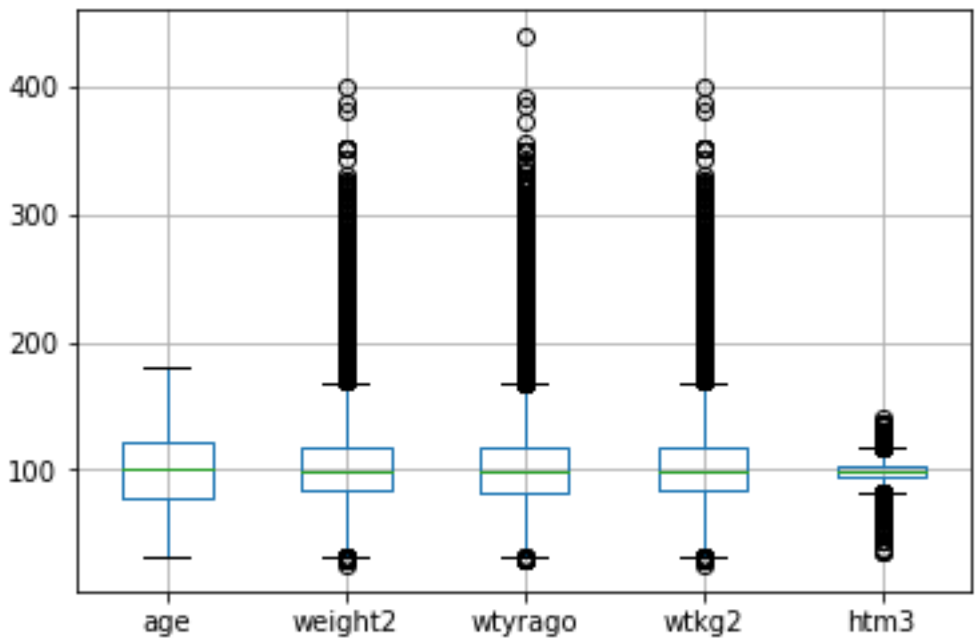
Mean-based scaling

In [2188]: def meanScaling(series):
...: return series / series.mean()
...: df8 = df4.apply(meanScaling) * 100
...: df8.boxplot()
...:



In [2229]: def trimMeanScale(series, proportionToCut=0):
...: return series / stats.trim_mean(series.dropna(), proportionToCut)
In [2230]: df8 = df4.apply(trimMeanScale, proportionToCut=0.1)*100
In [2231]: df8.boxplot()

Mean after removing largest and smallest proportionToCut data



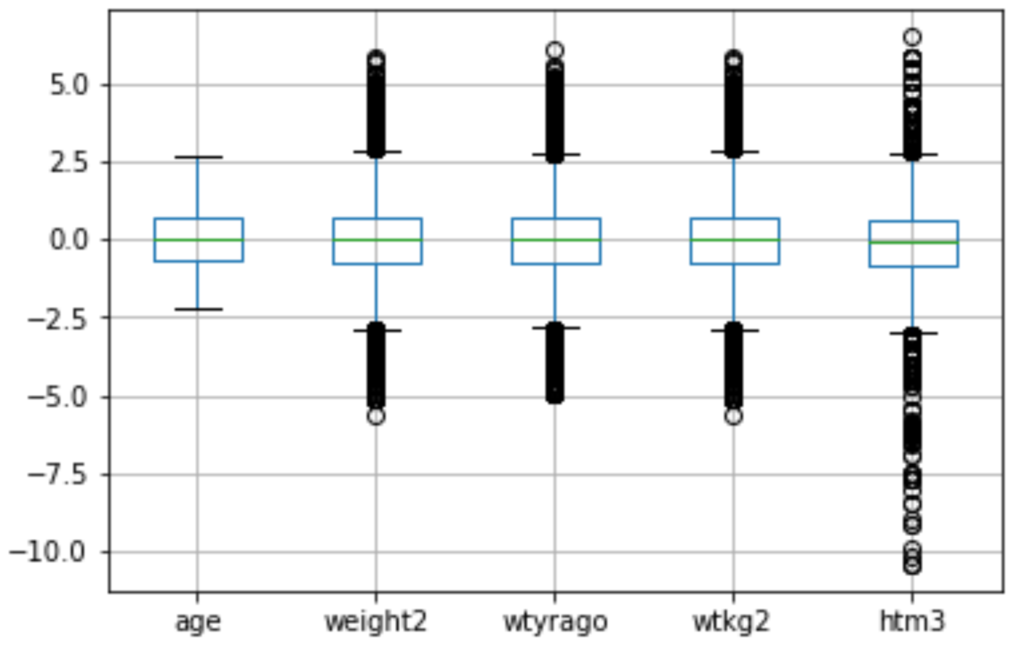
Transform and normalize

```
In [2242]: df9 = df4.transform({'age': np.copy, 'weight2': np.log, 'wtyrago': np.log, 'wtkg2': np.log, 'htm3': np.copy})
```

```
In [2243]: df10 = df9.apply(zscore);
```

```
In [2244]: df10.boxplot()
```

Transform each column with a different function



Additional materials

- https://www.geeksforgeeks.org/pandas-tutorial/?ref=lbp
- https://pandas.pydata.org/pandas-docs/stable/getting_started/intro_tutorials/index.html