**Python (version ≥3.6) for R Users: Stat Modules I**

**CMU MSP 36601, Fall 2017, Howard Seltman**

1. Use the **numpy** module to get vector, matrix, and array functionality as well as linear algebra. The official documentation is at <https://docs.scipy.org/>.
   1. Typically, people use import numpy as np to gain access to these functions.
   2. One key function is **array()** which converts a list of lists (or deeper) to an n-dimensional array (numpy.ndarray) in which all of the data are the same type.

>>> import numpy as np

>>> x = np.array([list(range(i, i+5)) for i in range(4)])

>>> x

array([[0, 1, 2, 3, 4],

[1, 2, 3, 4, 5],

[2, 3, 4, 5, 6],

[3, 4, 5, 6, 7]])

>>> print(type(x), x.ndim, x.shape, sep=", ")

<class 'numpy.ndarray'>, 2, (4, 5)

>>> x.flatten()

array([0, 1, 2, ..., 5, 6, 7])

>>> x.transpose()

array([[0, 1, 2, 3],

...

[4, 5, 6, 7]])

>>> x[:, 0:4]

array([[0, 1, 2, 3],

...

[3, 4, 5, 6]])

>>> x[1]

array([1, 2, 3, 4, 5])

>>> x[:, 1]

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

>>> x[:, 1].shape

(4,)

>>> x[:, 1].ndim

1

* 1. ***Direct array creation***

>>> np.empty((2, 3)) # no guarantee on contents

array([[ 0., 0., 0.],

[ 0., 0., 0.]])

>>> np.zeros((2, 3)) # guaranteed zeros

array([[ 0., 0., 0.],

[ 0., 0., 0.]])

>>> np.identity(3)

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

>>> np.ones((2, 3))

array([[ 1., 1., 1.],

[ 1., 1., 1.]])

>>> np.full((2, 3), 0.5) # fill with your specified value

array([[ 0.5, 0.5, 0.5],

[ 0.5, 0.5, 0.5]])

* 1. Aside: **lambda functions** are anonymous functions that can consist of only a single expression. E.g., lambda x: x\*\*(0.5) is an anonymous square root function.
  2. Array ***from a function***:

>>> np.fromfunction(lambda x, y: 10\*(x+1) + (y+1), (2, 3))

array([[ 11., 12., 13.],

[ 21., 22., 23.]])

* 1. Vector of evenly spaced values and **reshaping**

>>> v = (np.arange(5, 7.6, 0.5))

>>> v

array([ 5. , 5.5, 6. , 6.5, 7., 7.5])

>>> m = v.reshape(2, 3)

>>> m

array([[ 5. , 5.5, 6. ],

[ 6.5, 7. , 7.5]])

>>> m = v.reshape(2, 3, order="F") # Fortran (and R) order

>>> m

array([[ 5. , 6. , 7. ],

[ 5.5, 6.5, 7.5]])

* 1. ***Diagonals*** (like R, extracts a vector from a matrix or constructs a matrix from a vector)

>>> np.diag([1, 3, 4])

array([[1, 0, 0],

[0, 3, 0],

[0, 0, 4]])

>>> np.diag(np.arange(9).reshape(3, 3))

array([0, 4, 8])

* 1. The “@” operator performs ***matrix multiplication*** (also OK for an array and a list):

>>> m1 = np.array([[1, 3], [1, 1], [2, 4]])

>>> m2 = np.array([[2, 2, 2, 2], [4, 4, 4, 4]])

>>> m2 @ m1

ValueError: shapes (2,4) and (3,2) not aligned: 4 (dim 1) != 3 (dim 0)

>>> m1 @ m2

array([[14, 14, 14, 14],

[ 6, 6, 6, 6],

[20, 20, 20, 20]])

* 1. The transpose() method returns the ***transpose of a matrix***, but does not change the object and ***does*** rely on the original data. (.T without parentheses also works.)

>>> m1.transpose()

array([[1, 1, 2],

[3, 1, 4]])

>>> m1.T @ m1

array([[ 6, 12],

[12, 26]])

>>> m1t = m1.transpose()

>>> m1[0, 0] = 0

>>> m1t[0, 0]

0

* 1. ***Example: row and column means and sums***

>>> [sum(row) for row in m1]

[3, 2, 6]

>>> m1.sum(axis=1)

array([3, 2, 6])

>>> [np.mean(col) for col in m1.T]

[1.0, 2.6666666666666665]

>>> m1.mean(axis=0)

array([ 1. , 2.66666667])

* 1. ***Elementwise math*** on arrays works like in R. Examples include the usual operators plus add(), subtract(), multiply(), divide(), mod(), exp(), log(), reciprocal(), trigonometric functions, maximum(), minimum(), isfinite(), etc. Elementwise comparisons return an array of logical values. Warning: the usual “and” and “or” do not perform elementwise comparisons, but “&” and “|” do (but as usual may be misleading for non-logicals).

>>> np.diag([1, 2, 3]) + 10

array([[11, 10, 10],

[10, 12, 10],

[10, 10, 13]])

>>> np.diag([1, 2, 3]) + np.ones((3, 3))

array([[ 2., 1., 1.],

[ 1., 3., 1.],

[ 1., 1., 4.]])

>>> np.divide(np.diag([1, 2, 3]), np.ones((3, 3))) # or use "/"

>>> new\_arr = np.empty((3, 3))

>>> np.divide(np.diag([1, 2, 3]), np.ones((3, 3)), new\_arr)

array([[ 1., 0., 0.],

[ 0., 2., 0.],

[ 0., 0., 3.]])

>>> new\_arr

array([[ 1., 0., 0.],

[ 0., 2., 0.],

[ 0., 0., 3.]])

>>> new\_arr != 0

array([[ True, False, False],

[False, True, False],

[False, False, True]], dtype=bool)

>>> np.greater(new\_arr, np.identity(3))

array([[False, False, False],

[False, True, False],

[False, False, True]], dtype=bool)

>>> a1 = np.array([[True, True], [True, False]])

>>> a2 = np.array([[False, False], [True, False]])

>>> a1 and a2 # fails

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

>>> a1 & a2

array([[False, False],

[ True, False]], dtype=bool)

>>> np.array([[5, 5], [5, 0]]) & np.array([[2, 2], [2, 0]])

array([[0, 0],

[0, 0]], dtype=int32)

* 1. ***Other linear algebra functions*** are available:

>>> (m1 @ m1.T).trace()

31

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

32

>>> mat = np.array([[1, 2], [3, 4]])

>>> mat @ mat @ mat

array([[ 37, 54],

[ 81, 118]])

>>> np.linalg.matrix\_power(mat, 3)

array([[ 37, 54],

[ 81, 118]])

1. The **pandas** module provides functionality similar to R’s *data frames***.**  See <http://pandas.pydata.org/>.
   1. There is a cheatsheet at <https://github.com/pandas-dev/pandas/raw/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf>.
   2. The basic **DataFrame** object is collection of columns of type **Series**.
   3. A Series can be directly created with the “class constructor”, with or without item names:

prices = pd.Series([12.34, 5.10, 18.60, 2.50],

index=['A54', 'C17', 'B23', 'M17'])

prices

A54 12.34

C17 5.10

B23 18.60

M17 2.50

dtype: float64

* 1. ***Create a DataFrame using the class constructor***

>>> import pandas as pd

>>> names = ["Pooya", "Ralph", "Jihae", "Ling"]

>>> ages = [28, 31, 24, 22]

>>> MSP = [True, False, False, True]

>>> pd.DataFrame([names, ages, MSP])

0 1 2 3

0 Pooya Ralph Jihae Ling

1 28 31 24 22

2 True False False True

>>> dtf = pd.DataFrame(list(zip(names, ages, MSP)),

columns = ["name", "age", "MSP"])

>>> dtf

name age MSP

0 Pooya 28 True

1 Ralph 31 False

2 Jihae 28 False

3 Ling 22 True

>>> pd.DataFrame({'name': names, 'age': ages, 'MSP': MSP})

MSP age name

0 True 28 Pooya

1 False 31 Ralph

2 False 24 Jihae

3 True 22 Ling

>>> pd.DataFrame((names, ages, MSP))

PandasError: DataFrame constructor not properly called!

>>> type(dtf)

pandas.core.frame.DataFrame

* 1. ***Save as csv***

>>> fileLoc = r"data\fakeData.csv" # raw string

>>> dtf.to\_csv(fileLoc, index=False, header=True)

* 1. ***Read from csv***

>>> dtf2 = pd.read\_csv(fileLoc)

You can add header=False if there is no header with or without names=[] to set the column names. See documentation for many other options.

See also read\_table(), read\_clipboard(), read\_excel(), read\_sas, and read\_sql().

* 1. ***Check Series and DataFrame info***

>>> type(prices)

pandas.core.series.Series

>>> len(prices)

4

>>> prices.index

Index(['A54', 'C17', 'B23', 'M17'], dtype='object')

>>> prices.shape

>>> prices.get\_values()

array([ 12.34, 5.1 , 18.6 , 2.5 ])

>>> prices.values

array([ 12.34, 5.1 , 18.6 , 2.5 ])

>>> prices.dtype

dtype('float64')

>>> type(dtf)

pandas.core.frame.DataFrame

>>> dtf.dtypes

name object

age int64

MSP bool

dtype: object

>>> dtf.get\_values()

array([['Pooya', 28, True],

['Ralph', 31, False],

['Jihae', 28, False],

['Ling', 22, True]], dtype=object)

>>> dtf.axes

[RangeIndex(start=0, stop=4, step=1),

Index(['name', 'age', 'MSP'], dtype='object')]

>>> dtf.ndim

2

>>> dtf.size

12

>>> dtf.shape

(4, 3)

>>> dtf.head(2)

name age MSP

0 Pooya 28 True

1 Ralph 31 False

>>> dtf.tail(2)

name age MSP

2 Jihae 28 False

3 Ling 22 True

>>> dtf.index # R’s rownames()

RangeIndex(start=0, stop=4, step=1)

>>> dtf.index = [chr(ord('A')+i) for i in range(len(dtf))]

>>> dtf

name age MSP

A Pooya 28 True

B Ralph 31 False

C Jihae 28 False

D Ling 22 True ♠

* 1. ***Subsetting*** occurs in two forms: use of the “indexing operators ([])” is more familiar. An important alternative is the “data access methods”, which have been implemented with greater optimizations. These methods use the standard object.method notation, but that is followed by square brackets rather than parenthesis (see below).

Indexing operator method for Series

>>> prices[3]

2.5

>>> prices['M17']

2.5

>>> prices['C17':]

C17 5.1

B23 18.6

M17 2.5

>>> prices[:'C17'] # includes C17!!!

A54 12.34

C17 5.10

>>> prices[1:3] # does not include prices[3]

C17 5.1

B23 18.6

Indexing operator method for DataFrame

>>> dtf['age']

A 28

B 31

C 28

D 22

Name: age, dtype: int64

>>> type(dtf['age'])

pandas.core.series.Series

>>> dtf['age']['C'] # or dtf['age'][2]

24

>>> dtf[1] # fails!!!

>>> dtf[[1]]

age

A 29

B 31

C 24

D 22

>>> dtf[[1, 0]]

age name

A 29 Pooya

B 31 Ralph

C 24 Jihae

D 22 Ling

>>> dtf[['MSP', 'name']]

MSP name

A True Pooya

B False Ralph

C False Jihae

D True Ling

>>> dtf[1:3] # slices **rows**; excludes row 3

name age MSP

B Ralph 31 False

C Jihae 24 False

>>> dtf['B':'C'] # includes row 'C'

name age MSP

B Ralph 31 False

C Jihae 24 False

>>> dtf[[True, False, True, False]]

name age MSP

A Pooya 28 True

C Jihae 28 False

>>> dtf['age'].max()

31

>>> dtf['age'] < dtf['age'].max()

A True

B False

C True

D True

>>> dtf[dtf['age'] < dtf['age'].max()]

name age MSP

A Pooya 28 True

C Jihae 28 False

D Ling 22 True

# note: parentheses required and must use "&" instead of "and"

>>> dtf[(dtf['age'] > 22) & (dtf['age'] < 30)]

name age MSP

A Pooya 28 True

C Jihae 28 False

>>> dtf.age[dtf['age'] <= 28]

A 28

C 28

D 22

>>> dtf[(dtf['age'] <= 28) & dtf.MSP] [['age', 'MSP']]

age MSP

A 28 True

D 22 True

# Overloaded use of "~" complement operator in pandas:

>>> dtf[['age', 'MSP']][~((dtf['age'] <= 28) & dtf.MSP)]

age MSP

B 31 False

C 28 False

The above examples use the “indexing operators ([])”. An alternative is to use the data access methods, which have been implemented with greater optimizations:

* .column\_name is convenient but limited (e.g., silently does the wrong thing during assignment when a method of the same name exists)
* loc works on *labels* in the index.
* iloc works on the *positions* in the index (so it only takes integers).
* ix usually tries to behave like loc but falls back to behaving like iloc if the label is not in the index. It is on the deprecation list!

>>> dtf.name[0:2] # not robust!

A Pooya

B Ralph

>>> dtf.loc['B':'C'] # based on index label and inclusive!

name age MSP

B Ralph 31 False

C Jihae 24 False

>>> dtf.loc['B':'C', 'age':'MSP']

age MSP

B 31 False

C 24 False

>>> dtf.loc['B':'C', ['age', 'MSP']]

age MSP

B 31 False

C 24 False

>>> dtf.loc[['B', 'D'], 'age':'MSP']

age MSP

B 31 False

D 22 True

>>> dtf.loc[:, 'age':'MSP']

age MSP

A 28 True

B 31 False

C 24 False

D 22 True

>>> dtf.loc[[True, False, False, True]]

name age MSP

A Pooya 28 True

D Ling 22 True

>>> dtf.loc[dtf.age > 28]

name age MSP

B Ralph 31 False

>>> dtf[dtf.age > 28][['name', 'MSP']] # two steps

name MSP

B Ralph False

>>> dtf.loc[dtf.age > 28, ['name', 'MSP']] # one step

name MSP

B Ralph False

>>> dtf.iloc[[0, 3]]

name age MSP

A Pooya 28 True

D Ling 22 True

>>> dtf.iloc[dtf.age > 28] # fails

>>> dtf.iloc[[0, 3], 0:2] # exclusive like [0, 2)

name age

A Pooya 28

D Ling 22

>>> dtf.iloc[[i for (i, age) in zip(range(len(dtf)), dtf.age)

if age < 28], 0:2]

name age

C Jihae 24

D Ling 22

>>> list(enumerate(dtf['age']))

[(0, 28), (1, 31), (2, 24), (3, 22)]

>>> dtf.iloc[[i for (i, age) in enumerate(dtf['age'])

if age < 28], 0:2]

>>> dtf.iloc[[1, 3], ['MSP', 'age']] # fails

>>> dtf.iloc[[1, 3], dtf.axes[1].isin(['MSP', 'age'])]

Deprecated method .ix[] works like .loc[] but can revert to .iloc[] if needed.

>>> dtf.ix[[0, 3], ['MSP', 'age']]

MSP age

A True 28

D True 22

Other useful methods:

>>> dtf.drop("MSP", axis=1) # axis=0 drops columns by index id

name age

A Pooya 28

B Ralph 31

C Jihae 24

D Ling 22

>>> dtf.filter(regex = "^[a-z]") # add axis=0 to filter rownames

name age

A Pooya 28

B Ralph 31

C Jihae 24

D Ling 22

>>> temp = pd.DataFrame([[math.nan, 2, 3], [4, 5, 6]])

>>> temp

0 1 2

0 NaN 2 3

1 4.0 5 6

>>> temp.dropna(axis=0) # return rows with no nan’s

0 1 2

1 4.0 5 6

>>> temp.dropna(axis=1)

1 2

0 2 3

1. 5 6

>>> temp = temp.fillna(999) # perhaps useful before an export

>>> temp

0 1 2

0 999.0 2 3

1 4.0 5 6

* 1. ***Logic testing***

>>> dtf.isnull()

name age MSP

A False False False

B False False False

C False False False

D False False False

>>> dtf > 28

name age MSP

A True False False

B True True False

C True False False

D True False False

>>> dtf.age > 28

A False

B True

C False

D False

Name: age, dtype: bool

>>> (dtf.age == 31) | dtf.MSP # parentheses required; "or" fails

A True

B True

C False

D True

dtype: bool

* 1. ***Descriptive statistics***

>>> prices.sum() # ignores nan’s

38.54

>>> prices.count() # ignores nan’s

4

>>> dtf.count()

name 4

age 4

MSP 4

dtype: int64

>>> dtf.count(axis=1) # 0=rows (for each column) vs. 1 columns

A 3

B 3

C 3

D 3

dtype: int64

>>> dtf.mean() # runs DataFrame.mean(axis=0)

age 27.25

MSP 0.50

dtype: float64

>>> dtf['age'].mean(skipna=False) # runs Series.mean()

27.25

Other functions include sum(), min(), max(), mean(), std(), var(), rank(), sem(), and describe().

* 1. ***Plotting*** of Series or DataFrame objects

dtf['age'].plot.hist()

dtf['age'].plot.hist(bins=20)

dtf.MSP.plot.pie()

dtf.age.plot.box(title='Ages')

dtf.age.plot.kde()

dtf.boxplot('age', by='MSP')

dtf.plot.scatter('age', 'MSP')

♠

* 1. ***Restructuring***

>>> wide = pd.DataFrame([["A", 3, 4, 5],

["B", 6, 7, 8],

["C", 9, 10, 11]],

columns=["id","v1","v2","v3"])

>>> wide

id v1 v2 v3

0 A 3 4 5.0

1 B 6 7 8.0

2 C 9 10 NaN

>>> tall = pd.melt(wide, 'id')

>>> tall

id variable value

0 A v1 3.0

1 B v1 6.0

2 C v1 9.0

3 A v2 4.0

4 B v2 7.0

5 C v2 10.0

6 A v3 5.0

7 B v3 8.0

8 C v3 NaN

>>> tall.pivot('id', 'variable', 'value')

variable v1 v2 v3

id

A 3 4 5

B 6 7 8

C 9 10 11

* 1. ***Adding new columns and rows***

>>> dtf['age2'] = dtf['age'] \* 2

>>> dtf['ratio'] = dtf.age2 / dtf.age

>>> dtf

name age MSP age2 ratio

A Pooya 28 True 56 2.0

B Ralph 31 False 62 2.0

C Jihae 28 False 56 2.0

D Ling 22 True 44 2.0

>>> dtf.insert(2, 'score', [12,15,22,11])

dtf

name age score MSP age2 ratio

A Pooya 28 12 True 56 2.0

B Ralph 31 15 False 62 2.0

C Jihae 28 22 False 56 2.0

D Ling 22 11 True 44 2.0

>>> dtf.rename(columns={'age': 'Age'}, inplace=True)

>>> dtf.rename(columns={'Age': 'age'}, inplace=True)

>>> dtf['rx'] = [1, 2, 3, 1]

>>> codes = {1: 'Placebo', 2: 'Drug A', 3: 'Drug B'}

>>> dtf['rxc'] = dtf[['rx']].applymap(codes.get)

>>> dtf

name age score MSP age2 ratio rx rxc

A Pooya 28 12 True 56 2.0 1 Placebo

B Ralph 31 15 False 62 2.0 2 Drug A

C Jihae 28 22 False 56 2.0 3 Drug B

D Ling 22 11 True 44 2.0 1 Placebo

# equivalents to R’s cbind() and rbind()

>>> D1 = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],

columns=['c1', 'c2', 'c3'])

>>> D2 = pd.DataFrame([[11, 12, 13], [14, 15, 16], [17, 18, 19]],

columns=['c1', 'c2', 'c3'])

>>> pd.concat([D1, D2])

c1 c2 c3

0 1 2 3

1 4 5 6

2 7 8 9

0 11 12 13

1 14 15 16

2 17 18 19

>>> pd.concat([D1, D2], axis=1)

c1 c2 c3 c1 c2 c3

0 1 2 3 11 12 13

1 4 5 6 14 15 16

2 7 8 9 17 18 19

* 1. The **groupby() method** of a DataFrame takes a column name or a list of column names and returns “pandas.core.groupby.DataFrameGroupBy” object. This is somewhat similar to split() in R. The object reflects groups of observations for all values or combinations of values of the arguments to groupby(). Trying to print the object is uninformative. The len() of the object is the number of groups. The .groups attribute is a dictionary with keys equal to the values defining the groups and values equal to a vector of DataFrame indices (“row names”). The get\_group() method takes a key and returns a group. There are many ways to use the groupby object. The simplest is in a “for” loop. Each value is a tuple containing the key and the DataFrame. Here is an example:

>>> N = 10

>>> dat = pd.DataFrame({'id': ["S" + str(ii) for ii in range(N)],

'age': [22 + random.choice(range(3)) for

ii in range(N)],

'male': [random.random() < 0.5 for ii in

range(N)],

'score': [round(random.random(), 2) for

ii in range(N)]})

>>> dat

age id male score

0 23 S0 True 0.67

1 23 S1 False 0.59

2 23 S2 True 0.96

3 22 S3 True 0.21

4 22 S4 True 0.98

5 24 S5 True 0.70

6 24 S6 False 0.72

7 24 S7 False 0.68

8 23 S8 False 0.45

9 23 S9 False 0.32

>>> gm = dat.groupby('male')

>>> gm

<pandas.core.groupby.DataFrameGroupBy object at 0x000000000B2F8630>

>>> len(gm)

2

>>> gm.groups

{False: [1, 6, 7, 8, 9], True: [0, 2, 3, 4, 5]}

>>> for (male, grp) in gm:

print('-- male --' if male else '-- female --')

print(grp)

print("mean score is", grp.score.mean())

print()

-- female --

age id male score

1 23 S1 False 0.59

6 24 S6 False 0.72

7 24 S7 False 0.68

8 23 S8 False 0.45

9 23 S9 False 0.32

mean score is 0.552

-- male --

age id male score

0 23 S0 True 0.67

2 23 S2 True 0.96

3 22 S3 True 0.21

4 22 S4 True 0.98

5 24 S5 True 0.70

mean score is 0.704

>>> gm.get\_group(False)

age id male score

0 23 S0 False 0.80

2 23 S2 False 0.80

3 22 S3 False 0.46

4 22 S4 False 0.77

8 22 S8 False 0.39

9 24 S9 False 0.69

>>> gma = dat.groupby(['male','age'])

>>> gma.groups

{(False, 23): [1, 8, 9],

(False, 24): [6, 7],

(True, 22): [3, 4],

(True, 23): [0, 2],

(True, 24): [5]}

>>> gma.get\_group((False, 24))

age id male score

6 24 S6 False 0.72

7 24 S7 False 0.68

groupby() objects have methods for statistics that automatically apply the statistics to each applicable column in each group:

>>> gm.mean()

age score

male

False 23.4 0.552

True 22.8 0.704

>>> gm['age'].mean()

male

False 23.4

True 22.8

>>> gm.max()

age id score

male

False 24 S9 0.72

True 24 S5 0.98

>>> gma.mean()

score

male age

False 23 0.453333

24 0.700000

True 22 0.595000

23 0.815000

24 0.700000

The agg() method runs each of several selected functions on each group column:

>>> gma.agg([np.sum, np.mean, np.std])

score

sum mean std

male age

False 23 1.36 0.453333 0.135031

24 1.40 0.700000 0.028284

True 22 1.19 0.595000 0.544472

23 1.63 0.815000 0.205061

24 0.70 0.700000 NaN

The transform() method applies a function to each column by group. E.g., we can construct z-scores *by group* with:

>>> gm.transform(lambda x: (x-x.mean())/x.std())

age score

0 0.239046 -0.109320

1 -0.730297 0.228858

2 0.239046 0.823113

3 -0.956183 -1.588351

4 -0.956183 0.887419

5 1.434274 -0.012861

6 1.095445 1.011791

7 1.095445 0.770889

8 -0.730297 -0.614302

9 -0.730297 -1.397235

>>> pd.concat([dat, gm.transform(lambda x:

(x-x.mean())/x.std())],

axis=2)

Note the following confusing issue:

>>> np.std(dat.age) # numpy default is ddof=0 (pop. sd)

0.8306623862918074

>>> np.std(dat.age, ddof=1) # sample sd

0.8755950357709131

>>> dat.age.std() # pandas changes the ddof default to 1

0.8755950357709131

Aside: computing z-scores for all numeric data (ungrouped; and jumping ahead):

>>> def z(obj):

... if obj.dtype in (np.float64, np.int64):

... return (obj - np.mean(obj)) / np.std(obj)

... return(obj)

>>> print(dtf.apply(z))

See <http://pandas.pydata.org/pandas-docs/stable/groupby.html> for more groupby() methods.

* 1. **map()** for Series and **apply()** for DataFrame

>>> dtf['age'].map(math.log)

A 3.332205

B 3.433987

C 3.178054

D 3.091042

>>> dtf['age'].map(lambda x: x\*\*2)

A 784

B 961

C 576

D 484

>>> dtf['name'].map(str.lower)

A pooya

B ralph

C jihae

D ling

>>> dtf[['age', 'score', 'age2', 'ratio']].apply(np.median)

age 26.0

score 13.5

age2 52.0

ratio 2.0

* 1. **Convert integer columns to float** so that DataFrames will play nice with some other modules:

>>> dtf.dtypes

name object

age int64

score int64

MSP bool

age2 int64

ratio float64

rx int64

rxc object

dtype: object

>>> list(dtf.dtypes == 'int64')

[False, True, True, False, True, False, True, False]

>>> dtf.loc[:, dtf.dtypes == 'int64'] = \

dtf.loc[:, dtf.dtypes == 'int64'].astype('float64')

>>> dtf

name age score MSP age2 ratio rx rxc

A Pooya 28.0 12.0 True 56.0 2.0 1.0 Placebo

B Ralph 31.0 15.0 False 62.0 2.0 2.0 Drug A

C Jihae 24.0 22.0 False 48.0 2.0 3.0 Drug B

D Ling 22.0 11.0 True 44.0 2.0 1.0 Placebo

1. **matplotlib** (<http://matplotlib.org/>)

Default settings of Spyder show plots immediately, which prevents adding legends, etc.

Try changing Tools / Preferences / IPython Console / Graphics to "Backend: Qt5. (Requires IPython reset, e.g., with control-period.)

Use: import matplotlib.pyplot as plt

Then plots show in a separate window, but all plt command work together.

# evenly sampled time at 200ms intervals

t = np.arange(0., 5., 0.2) # like R's seq(0, 4.8, 0.2)

# Show red dashes, blue squares and green triangles.

# The 'format string' specifies color and line type using codes

# found in ?plt.plot.

plt.plot(t, t, 'r--', t, t\*\*2, 'bs', t, t\*\*3, 'g^')

plt.ylabel('score')

plt.xlabel('time')

plt.title("Powers")

plt.savefig("power.png")

plt.close() # optional