NUMPY import **numpy** as np

a=np.**array**([..]) a=np.**array**([[…],[…],[…]])

a[row, column]….a[row]….a[:,column]→only if matrix

a[a cond]= ret val of elem with cond True *using bool arrays as masks*

b=a<20 →b is np.array with T/F values

a.**where(a cond)** = ret positions (row, column)

a.**shape**=dimensions of array

a.**dtype**= array type (only if regular matrix)

np.**arange**([start,] stop[, step])=numpy array of num

a.**reshape**(raws,columns)

a=**np.NaN**

a.**transpose**()

np.**eye**(N)= 2-D array with 1 on diagonal & rest 0

a.**dot**(b)=dot product of 2 arryas *(shape may be req; ret scalar for 2 vectors)*

Linear Algebra import **numpy.linalg** as la

la.**inv**(a) = matriz inversa

$$A \cdot X = B$$

$$A^{-1} \cdot A \cdot X = A^{-1} \cdot B$$

$$X = A^{-1} \cdot B$$

linear equation system

x-2y=15

x+y=30

A = np.array([[1,1],[-2,1]])

B = np.array([30,15])

Solution=la.inv(A).dot(B)

PANDAS import pandas as pd *combination of numpy array and dictionary*

pd.**to_datetime**('2015-01-15 08:30').hour

Creating: pd.**Series**() pd.**DataFrame**() *axis **rows=0** **col=1***

S=pd.Series([1,2,3]*, index=list('abc')*) =pd.Series({'a':1, 'b' : 2, 'c':3})

S[index_name]→ ret element … same as for Dict

S['new_index']= values → adds new row to S

df=S.**to_frame**(name='col1') *and to add S to df:* df['col2']=S2

d=**{'a' : [1, 5],**'b' : [2,10], 'c':[np.NaN,20]**}** → dict →elem=col

d={'col' : [vals], 'col' : [vals],…} *dict (of narrays/lists)*

d= pd.Series([1,2,3]*, index=list('abc')*) *Series(=dict)*

d={'o' : S, 't' : pd.Series([1., 2.], index=['a', 'b'])} *dict of Series*

d=**[{'a': 1, 'b': 2},** {'a': 5, 'b': 10, 'c': 20}**]** → list →elem=row *list of dict*

d=[(1,2.,'Hello'), (2,3.,"World")] *list of tuples*

d=[S1,S2…] → each S is a row; S index=DF col *list of Series*

df=pd.DataFrame(d, index=['a','b'],**columns**=['c','d']) → **col acts as filter**

df= **pd.DataFrame.from_items**([('A', [1, 2, 3]), ('B', [4, 5, 6])])

df = **pd.concat**(list_S, axis=1).**transpose()** ***Series with different indices***

*list_S=[pd.Series([1,2],**index=['a','b']**), pd.Series([3,4],**index=['a','c']**)]*

df.**loc**[len(df)]=[vals] → add new row **label**

df['new_col']=[vals] → add new column

df2=**df.copy()** make a new DF not just a new label to object!

df1==df2 → test elements one by one

df2=**df1 operation df2** = df1-1 → operation on element level

Reading file/ZipExtFile/json into DF *default sep=","*

pd.**read_csv**(path, sep='^', index_col='name', **usecols**=['col1','col2'],

jhjtrfhjk'ltrh;ltrkhyg';ltrek

w;tor

pd.**read_json**('path or buffer') Convert a JSON string to pandas object

Row (**index**)/ **Column** labels ***immutable***

df.**set_index**('Col', inplace=True, drop=True)→use col as index

df.set_index(keys=[[vals]], inplace=True)→use set of keys as index

idx=**df.index**/**df.columns** = ret Index type with row/col labels

idx.**tolist**() = ret list of labels

idx.**values** = ret np.array of all index values *df.index.values[:10]*

idx.**name** =return index col name

idx.**unique**() = ret array of unique index values

df.**reindex**([old (and new) labels]) = might produce NaN

df.**reset_index()**=ret df with label info in columns(col_name, index, levels,..)

df.**rename**([index={},]**columns**={'Date2':'Date'})

Selecting elements: **df['col']** *df.col =probl numbers as prefix*

df_**col_filter**=**df[['col1',** 'col5', 'col17',…**]]** *filter columns*

df_**row_filter**=**df[[True,False…]].copy()** *filter rows*

#pd.options.mode.chained_assignment = None #default='warn'

filterd_rows =**~** (df['manufacturer'] == 'Airbus**')** & **(**df['nb_engines'] == 2**)**

df.ix['label'] =df.**loc**['label']= ret **all** 'label' row(s) → ret DF or S

df.**iloc**[N] = ret row at **position** N → ret S

df[N:N+1]=extract row N *df[N] will not work since it does not exist*

df**['col']['label']**= ret element → df.iloc[N]['col']=df['col'].iloc[num]

df.**info()** df.**head**(N) df.**tail**(N) df.**sample**()

#pd.set_option('display.max_columns', 100/None)

df.**describe**([include='all'])=ret df with statistics, excl NaN; incl strings

df.**dtypes**= ret dtypes of the object

df.**astype**(**dtype**, **copy=True**, raise_on_error=True) *cast S/df/row/col*

df.**shape**=(rows, columns) *size=df.shape[0] * df.shape[1]*

df.**size**= num of elem in df → *len(df) = number of records/rows*

df.**count(**[axis=0/1]**)**=ret S with label & number of **not-null** values

df.**isnull().sum**(axis=0/1)=ret S with label & number of **null** values

df.**fillna**(**value**=def_vals, **method**=None, **axis**=None, **inplace**=False)

*method:**None/ffill/ bfill** propagate forward/backward last/next valid value*

***def_vals** = {'c1' : 'v1', c2' : v2,…'}; df.fillna(def_vals)*

df.**dropna**(axis=**0/1/[0,1]**, **how**='any', **thresh**=N, **inplace**=False)

*how: drop label if '**all**' or at least one ('**any**') element is NA*

df_new= df.dropna(axis=1, thresh=1000)

df.**drop** (**labels**, **axis=0**, **inplace**=False, errors='raise')

*cols_drop = df.columns[df.**count**() < 1000]; df_new=df.drop(col_drop, axis=1)*

df.**drop_duplicates()**

df['column'].map(function) = apply to elements of columns or rows

*df['by_hour']=**map**(lambda x:pd.to_datetime(x).hour,df['TimeStr'])*

df.apply(func, axis=0/1)= apply to series over row/col

df.applymap(func)=apply to all elements

groups=df.**groupby**('gr_col') *group elems=DF with all cols & same gr_col val*

for group_name, group_elements in groups:

NaN groups in GroupBy are automatically excluded. This behavior is consistent with R, for example. convert via .astype(str) to string before grouping. That will conserve the NaN's. df.astype(str).groupby(['b']).size()

groups.**groups**= ret dict with group name & row labels

When grouping, we specify col(s) to group, then col to 'select' then agg/ops on select

groups**['sel_col'].agg**(['mean', 'count','std', 'nunique']) *column stats*

sel_col_groups = **groups['sel_col']** ret (Series Name, Series elems)

avg_of_col_per_group = sel_col_groups.mean()

groups.**agg**({'col1':'fun1', 'col2':'fun2'}) *apply only on selected col*

groups.**agg**(['**mean**', '**count**','**std**']) *apply alls func to all col except gr_col*

df.**sort_values**(**by**=['col1', 'col2'], **ascending**=False)

delays=df.groupby('TailNum')['DepDelay'].agg(['mean','count']).sort_values(by='mean')

df.**corr()**=correlation between the numerical variables

df.**sum(axis=0/1)** = 0→def=sum over rows of a column

df.**mean**([**axis=0/1**,skipna=True, level=None, numeric_only=None])

df.**std**([**axis=0/1**,skipna=True,level=None,ddof=1]) ddof=degrees of freedom

df.**join**(df2/S, on='col', **how='left'**, lsuffix=' ', rsuffix=' ', sort=False)

*df.join([df2,df3]) no options → **left join on indexes***

pd.**concat**(objs, **axis=0**, **join='outer'**, join_axes=None,copy= T/**F**, keys=, ignore_index=T/**F**, levels=, names=, verify_integrity= T/**F**,)

pd.concat([df1, df2, df3]) *no options → **outer join on indexes***

df.**merge**(df2, **how='inner'**, on=, left_on=, right_on=, left_index=T/**F**, right_index=T/**F**, sort=T/**F**, suffixes=('_x', '_y'), indicator=T/**F**)

.str= vectorized string functions for Index and Series (always use .str.func)

.str.split() = split each element to list

.str.startswith('P').unique()

df[col_name.str.startswith('P').fillna(False)].count()

df.columns=df.columns.**str.strip()** = remove blank spaces

filter_cols = df2.columns[df2.columns.**str.contains**('Origin')]

pd.set_option('display.max_colwidth', -1)

pd.set_option('display.max_columns', 100/None)