**Pandas Cheat Sheet**

**Reading Data**

*#The pandas read\_csv function is well-endowed, with over 30 optional parameters you can specify. For example, you can see in this dataset that the csv file has an in-built index, which pandas did not pick up on automatically. To make pandas use that column for the index (instead of creating a new one from scratch), we may specify and use an index\_col.*

*# csv → comma separated values*

file = pd.read\_csv("../input/wine-reviews/winemag-data\_first150k.csv", index\_col=0)

pd.DataFrame(file)

*#excel*

file = pd.read\_excel('../input/publicassistance/xls\_files\_all/WICAgencies2014ytd.xls', sheet\_name='Pregnant Women Participating')

pd.DataFrame(file)

*# SQL → Structured Query Language*

import sqlite3

conn = sqlite3.connect('../input/pitchfork-data/database.sqlite')

artists = pd.read\_sql\_query('SELECT \* FROM artists', conn)

pd.DataFrame(artists)

**pd.Series**

data1 = pd.Series([0.25, 0.5, 0.75, 1.0])

data1.values

data1.index

*#set index*

data2 = pd.Series([0.25, 0.5, 0.75, 1.0], index=[‘a’, ‘b’, ‘c’, ‘d’], name=’my name’)

*#constructing*

*#1 from python dictionary*

population\_dict = {‘California’: 123456,

‘Texas’:234567,

‘New York’: 123456,

‘Montana’:123456}

population = pd.Series(population\_dict)

population[‘California’]

population[‘California’:’New York’]

*#2 Series object*

pd.Series([0.25, 0.5, 0.75, 1.0])

pd Series(5, index=[100, 200, 300])

pd.Series({1:’a’, 2:’b’, 3:’c’})

pd.Series({1:’a’, 2:’b’, 3:’c’}, index=[3, 2]) *#only index 2 and 3*

**pd.DataFrame**

*#in a tidy data set each variable is saved in its own column and each observation is saved in its own row. Tiday data complements panda’s* ***vectorized operations****.*

area\_dict = {‘California’: 131434525,

‘Texas’:24352467,

‘New York’: 3245256,

‘Montana’:2542456}

area = pd.Series(area\_dict)

states = pd.DataFrame({‘population’:population, ‘area’:area})

states.index

states.columns

states[‘area’]

*#constructing*

*#1 from single Series object*

pd.DataFrame(population, columns=[‘population’])

*#2 from list of dicts*

data = [{‘a’:i, ‘b’: 2\*i} for i in range(3)]

pd.DataFrame(data)

*#3 from a dictionary of Series objects*

pd.DataFrame({‘population’:population, ‘area’:area})

*#4 from 2D NumPy array*

pd.DataFrame(np.random.rand(3, 2),

columns=[‘foo’, ‘bar’],

index=[‘a’, ‘b’, ‘c’])

*#5 directly*

df = pd.DataFrame({"a" : [4 ,5, 6], "b" : [7, 8, 9], "c" : [10, 11, 12]}, index = [1, 2, 3])

df = pd.DataFrame([[4, 7, 10],[5, 8, 11],[6, 9, 12]], index=[1, 2, 3], columns=['a', 'b', 'c'])

**pd.Index**

indA = pd.Index([2,4,5,7,9])

*#one difference between Index objects and NumPy array is that indices are immutable – they cannot by modified via the normal means*

indA[1] = 22 *#error!*

indB = pd.Index([1,4,4, 7,12])

indA & indB *#intersection*

indA | indB *#union*

indA ^ indB *#symetric difference*

**Data Indexing and Selection – pd.Series**

data1[1]

data1[1:3]

data2[‘b’]

‘a’ in data2 *#True*

data2.keys()

list(data2.items())

data2[‘e’] = 22 *#overwriting/append*

data2[‘a’:’c’]

data2[0:2] *#equal to data2[‘a’:’c’]*

data2[(data2> 0.3) & (data2< 0.8)] *#masking*

data2[[‘a’, ‘e’]] *#fancy indexing – double [[]]*

data2.loc[‘b’]

data2.loc[‘b’:’e’] *#via index name, explicit*

data2.iloc[1:3] *#via index position, implicite, python style*

**Data Indexing and Selection – pd.DataFrame**

states[‘area’] *#column name*

states[‘density’] = states[‘population’] / states[‘area’] *#new variable*

states.values

states.T *#transpose*

states.values[0]

states.iloc[:3, :2] *#row, column*

states.loc[:‘NewYork’, : ‘population’]

states.ix[:3, :’population’]

states.iloc[0,2] = 90 *#overwriting*

*#while indexing refers to columns, slicing refers to rows (first parametr)*

states[‘California’:’Montana’] *#row (slicing)*

states[1:3] *#as to above*

states[states.density > 100] *#masking, row-wise*

reviews['description'][0]

reviews[['country', 'points', 'price', 'variety']]

reviews['points'][[0,2,5,8]]

reviews.iloc[ :10, [1,3,6,9]]

reviews.loc[[0,1,10,100],['country', 'province', 'region\_1', 'region\_2']]

reviews.loc[2:30, 'points':'region\_2']

reviews.loc[reviews.region\_2.notnull()]

reviews.loc[wine.country.isin(['France', 'Albania']), ['description', 'points', 'price']]

reviews.points[reviews.country == 'Italy']

reviews.country[(reviews.country.isin(['France', 'Italy']) & (reviews.points >= 90))]

reviews.set\_index("title") *#setting index*

**Operating on Data in Pandas**

area = pd.DataFrame({‘Alaska’:1234455, ‘Texas’:123456, ‘Montana’:12334556}, name=’area’)

population = pd.DataFrame({‘New York’:12344455, ‘Texas’:5456456, ‘Montana’:34334556}, name=population’)

population / area *#contain NaN value*

area.index | population.index

*# if using NaN is not the desired behavior:*

A.add(B, fill\_value=0)

A – A[0]

df – df.iloc[0]

df.substract(df[‘R’], axis=0)

*#maping between python operators and pandas methods*

+ add()

- sub(), substract()

\* mul(), multiply()

/ truediv(), div(), divide()

// floordiv()

% mod()

\*\* pow()

**Summary functions and maps**

reviews.points.describe() *#.mean(), .median(), .value\_counts()*

median\_price = reviews.price.median()

reviews.price.map(lambda x: x – median\_price) *#map + lambda*

reviews.loc[(reviews.points / reviews.price).idxmax()].title *#idxmax*

a = reviews.loc[(reviews.country.notnull()) & (reviews.variety.notnull())]

a = a.apply(lambda srs: srs.country + ' - ' + srs.variety, axis='columns') *#apply*

a.value\_counts()

ratio = wine.loc[(wine.points.notnull() & wine.price.notnull())]

ratio.apply(lambda srs: srs.points / srs.price, axis='columns')

**Handling Missing Data**

data = pd.Series([1, np.nan, 2, None]) *#dtype float64, pandas automatically converts the None to NaN value*

data.isnull()

data[data.notnull()]

data.dropna() *#will drop all rows in which any null value is present*

df.dropna(axis=’columns’)

df.dropna(axis=’columns’, how=’all’) *#when all is NaN*

df.dropna(axis=’rows’, thresh=3) *#keep when at least 3 values are positive*

data.fillna(0)

data.fillna(method=’ffill’) *#forward-fill*

data.fillna(method=’bfill’) *#back-fill*

data.fillna(method=’ffill’, axis=1) *#notice that if a previous value is not available during a forward fill, the NA value remains*

**Hierarchical Indexing - MultiIndex**

index = [(‘California’, 2000), (‘California’, 2010), (‘Montana’, 2000), (‘Montana’, 2010)]

population = [12345, 23456, 345676, 231556]

pop = pd.Series(population, index=index) *#the bad way*

pop[(‘California’, 2010) : (‘Montana’, 2010)]

index = pd.MultiIndex.from\_tuples(index) *#the better way*

pop = pop.reindex(index)

pop[ :, :2010] *#index1, index2*

*#unstack() method convert a multiply-indexed Series into conventionally indexed DataFrame*

pop\_df = pop.unstack()

pop\_df.stack()

pop\_df = pd.DataFrame({‘total’:pop, ‘something’:[123,234,3243,24234})

*#creation*

df = pd.DataFrame(np.random.rand(4,2),

index=[[‘a’, ‘a’, ‘b’, ‘b’],[1,2,1,2]],

columns=[‘data1’, ‘data2’])

data = {(‘California’, 2000): 32434554,

(‘California’, 2010):23525456,

(‘Montana’, 2000):32414252,

(‘Montana’, 2010):32452552}

*#explicit MultiIndex constructor*

pd.MultiIndex.from\_arrays([[‘a’, ‘a’, ‘b’, ‘b’], [1,2,1,2]])

pd.MultiIndex.from\_tuples([(‘a’, 1), (‘a’, 2), (‘b’, 1), (‘b’, 2)])

pd.MultiIndex.from\_product([[‘a’, ‘b’],[1,2]])

pd.MultiIndex(levels=[[‘a’, ‘b’], [1,2]],

labels=[[0,0,1,1], [0,1,0,1]])

pop.index.names = [‘state’, ‘year’] *#setting indexes’ names*

*#MultiIndex for columns*

index = pd.MultiIndex.from\_product([[2013, 2014], [1,2]], names=[‘year’, ‘visit’])

columns = pd.MultiIndex.from\_product([[‘Bob’, ‘Elen’, ‘Olga’], [‘HR’,’Temp’]], names=[‘subject’, ‘type’])

my\_data = pd.DataFrame(data, index=index, columns=columns)

**Indexing and Slicing a MultiIndex**

*#Series*

pop[‘California’, 2000]

pop[‘California’]

pop.loc[‘California’:’Montana’]

pop[ :, 2010]

pop[pop > 220000]

pop[[‘California’, ‘Montana’]]

*#DataFrame*

my\_data[‘Olga’, ‘HR’] *#col, col*

my\_data.iloc[:2, :2] *#row, col*

my\_data.loc[ :, (‘Olga’, ‘Elen’)]

*#Rearranging Multi-Indices*

data = data.sort\_index()

pop.unstack(level=0)

pop\_flat = pop.reset\_index(name=’population’) *#move former index to DataFrame cells*

pop\_flat.set\_index(‘state’, ‘year’)

my\_data = my\_data.mean(level=’year’)

my\_data.mean(axis=1, level=’type’)

**Combining Dataset: Concat and Append**

pd.concat([data1, data2]) *#row-wise*

pd.concat([data1, data2], axis=’col’)

*#pandas concatenation preserve indices by default*

pd.concat([data1, data2], ignore\_index=True) *#new indices*

pd.concat([data1, data2], keys=[‘x’, ‘y’]) *#adding MultiIndex keys*

*#by default, the entries for which no data is available are filled with NA value*

pd.concat([data1, data2], join=’inner’)

pd.concat([data1, data2], join\_axes=[data1.columns])

data1.append(data2) *#as a option but better to use pd.concat([data1, data2])*

**Combining Datasets: Merge and Join**

*#one-to-one joins – the same values in merged key-column*

*#many-to-one joins – more values in merged key-column*

*#many-to-many joins – more values in both sides (mix)*

pd.merge([data1, data2])

pd.merge([data1, data2], on=’employee’) *#selecting key column*

pd.merge([data1, data2], left\_on=’employee’, right\_on=’name’).drop(‘name’, axis=1)

pd.merge([data1, data2], left\_index=True, right\_index=True) *#via index*

data1.join(data2) *#by default joining on indicies*

pd.merge([data1, data2], left\_on=’employee’, right\_index=True)

pd.merge([data1, data2], how=’inner’) *#outer, left, right*

pd.merge([data1, data2], on=’employee’, suffixes=[‘\_L’, ‘\_R’]) *#adding L and R to conflicting columns*

**Aggregation and Grouping**

data.mean() *#via columns by default*

data.mean(axis=’columns’)

*#pandas aggregation methods*

count() *#total number of items*

first(), last()

mean(), median()

min(), max()

std(), var()

mad() *#mean absolute deviation*

prod() *#product od all items*

sum()

*#split – apply – combine*

df.groupby(‘key’).sum()

planets.groupby(‘method’)[‘orbitral\_period’].median()

planets.groupby(‘method’)[‘year‘].unstack()

df.groupby(‘key’).aggregate(‘min’, np.median, max)

df.groupby(‘key’).aggregate({‘data1’: ‘min’, ‘data2’:’max’})

def filter\_func(x):

return x[‘data2’].std() > 4

df.groupby(‘key’).filter(filter\_func)

df.groupby(‘key’).transform(lambda x: x – x.mean())

def norm\_by\_data2(x):

x[‘data1’] /= x[‘data2’].sum()

return x

df.groupby(‘key’).apply(norm\_by\_data2)

L = [0, 1, 0, 1, 2, 0]

df.groupby(L).sum()

df.groupby(df[‘key’]).sum()

df2 = df.set\_index(’key’)

mapping = {‘A’:’vowel’, ‘B’: ‘consanat’, ‘C’: ‘consanat’}

df2.groupby(maping).sum()

df2.groupby(str.lower).mean()

df2.groupby([str.lower, mapping]).mean()

*#exemple*

decade = 10 \* (planets[‘year’] // 10)

decade = decade.astype(str) + ‘s’

decade.name = ‘decade’

planets.groupby([‘method’, decade])[‘number’].sum().unstack().fillna(0)

**Pivot Tables**

*#it helps to think of pivot tables as essentialy a multidimensional version of groupby aggregation*

titanic.pivot\_table(‘survived’, index=’sex’, columns=’class’) *#groub by ‘sex’ and ‘class’ and count mean for ‘survived’ (mean is set by default)*

age = pd.cut(titanic[‘age’], [0, 18, 80])

titanic.pivot\_table(‘survived’, [’sex’, age], ’class’)

fare = pd.qcut(titanic[‘fare’], 2)

titanic.pivot\_table(‘survived’, [’sex’, age], [fare, ’class’])

titanic.pivot\_table(‘survived’, index=’sex’, columns=’class’,

aggfunc{‘survived’:sum, ‘fare’:’mean’})

titanic.pivot\_table(‘survived’, index=’sex’, columns=’class’, margins=True)

**Vectorized String Operations**

*#delarujemy str aby nie zwracało błedu w przypadku napotkania NaN lub int/float.*

names.str.capitalize() *#for pd.Series*

names[‘surname’].str.capitalize() *#for pd.DataFrame*

len() lower() translate() islower()

ljust() upper() startswith() isupper()

rjust() find() endswith() isnumeric()

center() rfind() isalnum() isdecimal()

zfill() index() isalpha() split()

strip() rindex() isdigit() rsplit()

rstrip() capitalize() isspace() partition()

lstrip() swapcase() istitle() rpartition()

*#pandas method*

match() extract() findall() replace()

contains() count() split() rsplit()

get() slice() slice\_replace() cat()

repeat() normalize() pad() wrap()

join() get\_dummies()

names.str.extract(‘([A-Za-z]+)’)

names.str.finall(r’^[AEIOU.\*[^aeiou]]$’)

names.str[0:3]

names.str.slit().str.get(-1)

**Working with Time Series**

index = pd.DatetimeIndex([‘2014-07-04’, ‘2014-08-04’, ‘2015-07-04’, ‘2015-08-04’])

data = pd.Series([0,1,2,3], index=index)

data[‘2014-07-04’: ‘2015-07-04’] *#slicing*

data[‘2015’] *#by year*

dates = pd.to\_datetime([datetime(2015, 7, 3), ‘4th of July, 2015’, ‘2015-Jul-6’, ‘20150804’])

dates.to\_period(‘D’) *#to daily frequency*

dates – dates[0]

pd.date\_range(‘2015-07-03’, ‘2015-07-10’) *#default frequency is day (‘D’)*

pd.date\_range(‘2015-07-03’, periods=8) *#8 next days*

pd.date\_range(‘2015-07-03’, periods=8, freq=’H’) *#8 next hours*

pd.period\_range(‘2015-07’, periods=8, freq=’M’) *#8 next month, period without part of full date*

pd.timedelta\_range(0, periods=10, freq=’H’) *#10 next hours after midnight*

*#listing of pandas frequency codes*

D calendar day B buisness day

W weekly

M month end BM buisness month end

Q quarter end BQ buisness quarter end

A year end BA buisness year end

H hours BH buisenss hours

T minutes

S seconds

L milliseconds

U microseconds

N nanoseconds

*#listing of start –indexed frequence codes*

MS month start BQS buisness quarter start

BMS buisness month start AS year start

QS quarter start BAS buisness year start

pd.timedelta\_range(0, periods=9, freq=’2H30T’)

*#we can import financial data from a number of available sources, including Yahoo finance, Google Finance.*

from pandas\_datarender import data

goog = data.DataReader(‘GOOG’, start=’2004’, end=’2016’, data\_source=’google’)

**High-Performance Pandas: eval() and query()**

*#pandas includes some experimental tools that allow you to directly access C-speedoperations without costly alocationof intermediate arrays. Those are the eval() and query() functions, which rely on the Numexpr package. The benefit of eval/query is mainly in the saved memory.*

%timeit df1 + df2 + df3 + df4

%timeit pd.eval(‘df1 + df2 + df3 + df4’) *#much faster!*

pd.eval(‘df1 < df2 <= df3 != df4’)

pd.eval(‘(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)’)

pd.eval(‘(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)’)

pd.eval(‘df2.T[0] + df3.iloc[1]’)

pd.eval(‘(df.A + df.B) / (df.C - )’) *#columns without []*

pd.eval(‘(A + B) / (C - 1)’) *#columns in DataFrame*

pd.eval(‘D = (A + B) / C’, inplace=True)

*#query function works in the same way*