Cheat Sheet: The Pandas DataFrame Object

Preliminaries

Start by importing these Python modules

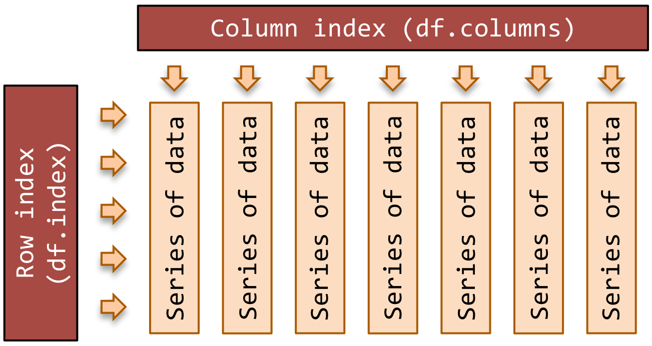
```
import numpy as np                # required
import pandas as pd               # required
from pandas import DataFrame, Series # useful
```

The conceptual model

Series object: an ordered, one-dimensional array of data with an index. All the data in a Series is of the same data type. Series arithmetic is vectorised after first aligning the Series index for each of the operands.

```
s1 = Series(range(0,4))  # --> 0, 1, 2, 3
s2 = Series(range(1,5))  # --> 1, 2, 3, 4
s3 = s1 + s2             # --> 1, 3, 5, 7
s4 = Series(['a','b'])*3 # --> 'aaa', 'bbb'
```

DataFrame object: a two-dimensional table of data with column and row indexes. The columns are made up of pandas Series objects.



Get your data into a DataFrame

Data in Series then combine into a DataFrame

```
# Exmaple 1 ...
s1 = Series(range(6))
s2 = s1 * s1
s2.index = s2.index + 2 # misaligned indexes
df = pd.concat([s1, s2], axis=1)

# Exmaple 2 ...
s3 = Series({'Tom':1, 'Dick':4, 'Harry':9})
s4 = Series({'Tom':3, 'Dick':2, 'Mary':11})
df = pd.concat({'A': s3, 'B': s4 }, axis=1)
```

Note: 1st method has in integer column labels
Note: 2nd method does not guarantee col order
Note: index alignment on DataFrame creation

Load a DataFrame from a CSV file

```
df = pd.read_csv('file.csv')
```

Load a DataFrame from a Microsoft Excel file

```
# put each Excel workbook in a dictionary
workbook = pd.ExcelFile('file.xls')
d = {}
for name in workbook.sheet_names:
    df = workbook.parse(name)
    d[name] = df
```

Load a DataFrame from a MySQL database

```
import MySQLdb as db
import pandas.io.sql as ps
cx = db.connect('localhost', 'username',
                'password', 'database')
df = ps.frame_query('SELECT * FROM data', cx)
```

Get a DataFrame from a Python dictionary

```
# default - assume your data in columns
df = DataFrame({
        'col0' : [1.0, 2.0, 3.0, 4.0],
        'col1' : [100, 200, 300, 400]
    })

# use helper method for data in rows
df = DataFrame.from_dict({ # data by row
        'row0' : {'col0':0, 'col1':'A'},
        'row1' : {'col0':1, 'col1':'B'}
    }, orient='index')

df = DataFrame.from_dict({ # data by row
        'row0' : [1, 1+1j, 'A'],
        'row1' : [2, 2+2j, 'B']
    }, orient='index')
```

Create play data (useful for testing)

```
# created from a 2D numpy array (of randoms)
df = DataFrame(np.random.randn(26,5),
    columns=['col'+str(i) for i in range(5)],
    index=list("ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
df['cat'] = list('aaaabbbccddef' * 2)
```

Saving a DataFrame

Writing DataFrames to CSV

```
df.to_csv('filename.csv', encoding='utf-8')
```

Writing DataFrames to Excel

```
from pandas import ExcelWriter
writer = ExcelWriter('filename.xlsx')
df1.to_excel(writer,'Sheet1')
df2.to_excel(writer,'Sheet2')
writer.save()
```

Writing DataFrames to MySQL

```
import MySQLdb as db
cx = db.connect('localhost', 'username',
                'password', 'database')
df.to_sql(name='tablename', con=cx,
    flavour='mysql', if_exists='replace')
```

Note: if_exists - 'fail', 'replace', 'append'

Working with row and column indexes

DataFrames have two Indexes

Typically, the column index is a list of strings (observed variable names) or (less commonly) integers. The row index might be

- Integers - for case or row numbers (default is numbered from 0 to length-1)
- Strings – for case names
- DatetimeIndex or PeriodIndex – for time series data (more on these indexes below)

Get column index and labels

```
idx = df.columns                  # get col index
label = df.column[0]              # 1st col label
lst = df.columns.tolist()         # get as a list
```

Change column labels

```
df.rename(columns={'old':'new'},inplace=True)
df = df.rename(columns = {'a':'a1','b':'b2'})
```

Get the row index and labels

```
idx = df.index                    # get row index
label = df.index[0]               # 1st row label
lst = df.index.tolist()           # get as a list
```

Change the (row) index

```
df.index = idx                 # new ad hoc index
df.index = range(len(df))      # set with list
df = df.reset_index() # replace old with new
     # note: old index stored as a col in df
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['col1','col2','etc'])
df.rename(index={'old':'new'}, inplace=True)
```

Get the integer position of a row index label

```
i = df.index.get_loc('label') # also columns
```

Sort DataFrame by its row or column index

```
df.sort_index(inplace=True) # sort by rows
df = df.sort_index(axis=1)  # sort by cols
```

Test if the index values are unique/monotonic

```
if df.index.is_unique: pass # do something
if df.columns.is_unique: pass # do something
if df.index.is_monotonic: pass # do something
if df. columns.is_monotonic: pass # something
```

For a monotonic index, each element is greater than or equal to the previous element

Drop duplicates in the row index

```
df['index'] = df.index       # 1 create new col
df = df.drop_duplicates(cols='index',
        take_last=True)      # 2 use new col
del df['index']              # 3 del the col
df.sort_index(inplace=True)# 4 tidy up
```

Test if two DataFrames/Series have same index

```
len(a) == len(b) and all(a.index == b.index)
```

Working with columns of data (axis=1)

A DataFrame column is a pandas Series object

Selecting columns

```
s = df['colName']      # select column by name
df = df[['a','b']]     # select 2 or more cols
df = df[['c','a','b']]# change column order
s = df[df.columns[0]] # select column by num
```

Selecting columns with Python attributes

```
s = df.a               # same as s = df['a']
df.existing_col = df.a / df.b
# cannot create new columns by attribute ...
df['new_col'] = df.a / df.b
```

Trap: column names must be valid identifiers.

Adding new columns to a DataFrame

```
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan, len(df))
df['random'] = np.random.rand(len(df))
df['index_as_col'] = df.index
df1[['b','c']] = df2[['e','f']]   # multi add
df3 = df1.append(other=df2)       # multi add
```

Swap column contents

```
df[['B', 'A']] = df[['A', 'B']]
```

Dropping columns (by label)

```
df = df.drop('col1', axis=1)
df.drop('col1', axis=1, inplace=True)
df = df.drop(['col1','col2'], axis=1) # multi
s = df.pop('col') # get col; drop from frame
del df['col']      # even classic python works
```

Selecting columns with .loc, .iloc and .ix

```
df = df.loc[:, 'col1':'col2'] #inclusive "to"
df = df.iloc[:, 0:2]          #exclusive "to"
```

A slice of columns can be selected by label (using df.loc[rows, cols]); by integer position (using df.iloc[rows, cols]); or a hybrid of the two (using df.ix[rows, cols])

Note: the row slice object : copies all rows

Note: For .loc, the indexes can be:

- A single label (eg. 'A')
- A list/array of labels (eg. ['A', 'B'])
- A slice object of labels (eg. 'A':'C')
- A Boolean array

Note: For .iloc, the indexes can be

- A single integer (eg. 27)
- A list/array of integers (eg. [1, 2, 6])
- A slice object with integers (eg. 1:9)

Vectorised arithmetic on columns

```
df['proportion'] = df['count'] / df['total']
df['percent'] = df['proportion'] * 100.0
```

Apply numpy mathematical functions to columns

```
df['log_data'] = np.log(df['col1'])
df['rounded'] = np.round(df['col2'], 2)
```

Columns value set based on criteria

```
# Option 1: using a mask
df['new'] = 0
df[df['c'] > 0]['new'] = df['c']
# Option 2: using the where statement
df['new'] = df['c'].where(df['c']>0, other=0)
```

Note: Multiple conditions can be combined using & and | with conditions in parentheses.
Note: where other can be a Series or a scalar
Note: ~ the boolean not operator for pandas

Iterating over the Dataframe cols

```
for (column, series) in df.iteritems():
    # do something ...
```

Where column is the label and series is a pandas Series that contains the column data.

Common column-wide methods/attributes

```
value = df['col1'].dtype      # type of data
value = df['col1'].size       # col dimensions
value = df['col1'].count()    # non-NA count
value = df['col1'].sum()
value = df['col1'].prod()
value = df['col1'].min()
value = df['col1'].max()
value = df['col1'].mean()
value = df['col1'].median()
value = df['col1'].cov(df['col2'])
s =     df['col1'].describe()
s =     df['col1'].value_counts()
```

Find index for min/max values in column

```
value =  df['col1'].idxmin() # returns label
value =  df['col1'].idxmax() # returns label
```

Common column element-wise methods

```
s = df['col'].to_datetime()
s = df['col1'].isnull()
s = df['col1'].notnull() # not isnull()
s = df['col1'].astype('float') # type convert
s = df['col1'].round(decimals=0)
s = df['col1'].diff(periods=1)
s = df['col1'].shift(periods=1)
s = df['col1'].fillna(0) # replace NaN with 0
s = df['col1']. pct_change(periods=4)
s = df['c'].rolling_min(periods=4, window=4)
s = df['c'].rolling_max(periods=4, window=4)
s = df['c'].rolling_sum(periods=4, window=4)
```

Append a column of row totals to a DataFrame

```
df['Total'] = df.sum(axis=1)
```

Note: can do row means, mins, maxs, etc. in a similar manner.

Group by a column

```
s = df.groupby('cat')['col1'].sum()
dfg = df.groupby('cat').sum()
```

Group by a row index (non-hierarchical index)

```
df = df.set_index(keys='cat')
s = df.groupby(level=0)['col1'].sum()
dfg = df.groupby(level=0).sum()
```

Working with rows (axis=0)

Adding rows

```
df = original_df.append(more_rows_in_df)
```

Hint: convert to a DataFrame and then append. Both DataFrames should have same col labels.

Dropping rows (by name)

```
df = df.drop('row_label')
df = df.drop(['row1','row2'])      # multi-row
```

Boolean row selection by values in a column

```
df = df[df['col2'] >= 0.0]
df = df[(df['col3']>=1.0) | (df['col1']<0.0)]
df = df[df['col'].isin([1,2,5,7,11])]
df = df[~df['col'].isin([1,2,5,7,11])] # not
df = df[df['col'].str.contains('hello')]
```

Trap: bitwise "or" and "and" co-opted to be Boolean operators on a Series of Boolean --> also note parentheses around comparisons.

Select a slice of rows by integer position
[inclusive-from : exclusive-to]
[inclusive-from : exclusive-to : step]
default start is 0; default end is len(df)

```
df = df[:]       # copy DataFrame
df = df[0:2]     # rows 0 and 1
df = df[-1:]     # the last row
df = df[2:3]     # row 2 (the third row)
df = df[:-1]     # all but the last row
df = df[::2]     # every 2nd row (0 2 ..)
```

Trap: a single integer without a colon is a column index for numbered columns.

Select a slice of rows by label/index
[inclusive-from : inclusive–to [: step]]

```
df = df['a':'c'] # rows 'a' through 'c'
```

Trap: doesn't work on integer labelled rows

Append a row of column totals to a DataFrame

```
# Option 1: using a dictionary comprehension
sums = {col: df[col].sum() for col in df}
sums_df = DataFrame(sums, index=['Total'])
df = df.append(sums_df)
# Option 2: All done with pandas
df = df.append(DataFrame(df.sum(),
     columns=['Total']).T) # .T is transpose
```

Iterating over DataFrame rows

```
for (index, row) in df.iterrows():
```

Trap: row data type may be coerced.

Sorting DataFrame rows by column values

```
df = df.sort(df.columns[0], ascending=False)
df.sort(['col1', 'col2'], inplace=True)
```

Remember!

```
w = df['label']          # a selected column
x = df[['L1', 'L2']]     # selected columns
y = df['label':'label']  # selected rows
z = df[i:j] # where i & j are ints, → rows
```

Working with cells

Selecting a cell by row and column labels

```
value = df.at['row', 'col']
value = df.loc['row', 'col']
value = df['col']['row']                # tricky
```

Note: .at[] fastest label based scalar lookup

Setting a cell by row and column labels

```
df.at['row, 'col'] = value
df.loc['row, 'col'] = value
df['col']['row'] = value                # tricky
```

Selecting and slicing on labels

```
df = df.loc['row1':'row3', 'col1':'col3']
```

Note: the "to" on this slice is inclusive.

Setting a cross-section by labels

```
df.loc['A':'C', 'col1':'col3'] = np.nan
df.loc[1:2, 'col1':'col2'] = np.zeros((2,2))
df.loc[1:2, 'A':'C'] = other.loc[1:2,'A':'C']
```

Remember: inclusive from:to in the slice

Selecting a cell by integer position

```
value = df.iat[9, 3]              # [row, col]
value = df.iloc[0, 0]             # [row, col]
value = df.iloc[len(df)-1, len(df.columns)-1]
```

Selecting a range of cells by int position

```
df = df.iloc[2:4, 2:4]# a subset of the df
df = df.iloc[:5, :5]  # top left corner
s = df.iloc[5, :]     # returns row as Series
df = df.iloc[5:6, :]  # returns row as a row
```

Note: exclusive "to" – same as list slicing.

Setting cell by integer position

```
df.iloc[0, 0] = value             # [row, col]
df.iat[7, 8] = value
```

Setting cell range by integer position

```
df.iloc[0:3, 0:5] = value
df.iloc[1:3, 1:4] = np.ones((2,3))
```

Remember: exclusive from:to in the slice

Operate on the whole DataFrame

```
# replace np.nan with 0
df.fillna(0, inplace=True)
# replace white space with np.nan
df = df.replace(r'\s+', np.nan, regex=True)
```

Views and copies

From the manual: The rules about when a view on the data is returned are dependent on NumPy. Whenever an array of labels or a boolean vector are involved in the indexing operation, the result will be a copy. A single label/scalar indexing & slicing, e.g. df.ix[3:6] or df.ix[:, 'A'], returns a view.

Joining/Combining DataFrames

Three ways to join two DataFrames:

- merge (a database/SQL-like join operation)
- concat (stack side by side or stack one on top of the other)
- combine_first (splice the two togther, choosing values from one over the other)

Merge on indexes

```
df_new = pd.merge(left=df1, right=df2,
  how='outer', left_index=True,
  right_index=True)
```

How: 'left', 'right', 'outer', 'inner'
How: outer=union/all; inner=intersection

Merge on columns

```
df_new = pd.merge(left=df1, right=df2,
how='left', left_on='col1', right_on='col2')
```

Trap: When joining on columns, the indexes on the passed DataFrames are ignored.
Trap: many-to-many merges on a column can result in an explosion of associated data.

Join on indexes (another way of merging)

```
df_new = df1.join(other=df2, on='col1',
                   how='outer')
df_new = df1.join(other=df2, on=['a', 'b'],
                   how='outer')
```

Note: DataFrame.join() joins on indexes by default. DataFrame.merge() joins on common columns by default.

Simple concatenation is often the best

```
df=pd.concat([df1,df2],axis=0) # top/bottom
df = df1.append([df2, df3])    # top/bottom
df=pd.concat([df1,df2],axis=1) # left/right
```

Trap: can end up with duplicate rows or cols
Note: concat has an ignore_index parameter

Combine_first

```
df = df1.combine_first(other=df2)
df = reduce(lambda x, y: x.combine_first(y),
       [df1, df2, df3, df4, df5])
```

Uses the non-null values from df1. The index of the combined DataFrame will be the union of the indexes from df1 and df2.

Working with the whole DataFrame

Peek at the DataFrame

```
summary_df = df.describe()
head_df = df.head(); tail_df = df.tail()
top_left_corner_df = df.iloc[:5, :5]
```

Other useful

```
df = df.T          # transpose rows and columns
df2 = df.copy()    # copy a DataFrame
```

Working with dates, times and their indexes

Dates and time – points and spans

With its focus on time-series data, pandas provides a suite of tools for managing dates and time: either as a point in time (a Timestamp) or as a span of time (a Period).

```
timestamp = pd.Timestamp('2013-01-01')
period = pd.Period('2013-01-01', freq='M')
```

Dates and time – stamps and spans as indexes

An index of Timestamps is a DatetimeIndex; and an index of Periods is a PeriodIndex. These can be constructed as follows:

```
date_strs = ('2013-10-01', '2013-11-01',
             '2013-12-01', '2014-01-01')
tstamp = pd.to_datetime(pd.Series(date_strs))
dt_idx = pd.DatetimeIndex(tstamp, freq='MS')
prd_idx = pd.PeriodIndex(tstamp, freq='M')
spi = Series([1,2,3,4], index=prd_idx)
sdi = Series([1,2,3,4], index=dt_idx)
# Also: index changed through its attribute
spi.index = dt_idx # change to time stamps
spi.index = range(len(spi)) # to integers
```

From DatetimeIndex and PeriodIndex and back

```
spi = sdi.to_period(freq='M')# to PeriodIndex
sdi = spi.to_timestamp()   # to DatetimeIndex
```

Note: from period to timestamp defaults to the point in time at the start of the period.

Frequency constants (not a complete list)

Name	Description
U	Microsecond
L	Millisecond
S	Second
T	Minute
H	Hour
D	Calendar day
B	Business day
W-{MON, TUE, …}	Week ending on …
MS	Calendar start of month
M	Calendar end of month
QS-{JAN, FEB, …}	Quarter start with year starting (QS – December)
Q-{JAN, FEB, …}	Quarter end with year ending (Q – December)
AS-{JAN, FEB, …}	Year start (AS - December)
A-{JAN, FEB, …}	Year end (A - December)

More examples on working with dates/times

```
d = pd.to_datetime(['04-01-2012'],
      dayfirst=True) # Australian date format
t = pd.to_datetime(['2013-04-01 15:14:13.1'])
```

DatetimeIndex can be converted to an array of Python native datetime.datetime objects using the to_pydatetime() method.

Error handling with dates

```
# first example returns string not Timestamp
s = pd.to_datetime('2014-02-30')
# second example returns NaT (not a time)
n = pd.to_datetime('2014-02-30', coerce=True)
# NaT is like NaN ... tests True for isnull()
b = pd.isnull(n) # --> True
```

Creating date/period indexes from scratch

```
dt_idx = pd.DatetimeIndex(pd.date_range(
    start='1/1/2011',  periods=12, freq='M'))
p_idx = pd.period_range('1960-01-01',
      '2010-12-31', freq='M')
```

Row selection with a time-series index

```
# play data ... start with play data above
idx = pd.period_range('2013-01',
      periods=len(df), freq='M')
df.index = idx

february_selector = (df.index.month == 2)
february_data = df[february_selector]

q1_data = df[(df.index.month >= 1) &
   (df.index.month <= 3)] # note: & not "and"

mayornov_data = df[(df.index.month == 5) |
   (df.index.month == 11)] # note: | not "or"

annual_tot = df.groupby(df.index.year).sum()
```

Also: year, month, day [of month], hour, minute, second, dayofweek [Mon=0 .. Sun=6], weekofmonth, weekofyear [numbered from 1], week starts on Monday], dayofyear [from 1], … Note: this method works with both Series and DataFrame objects.

The tail of a time-series DataFrame

```
df = df.last("5M")     # the last five months
```

Working with strings

Working with strings

```
# assume that df['col'] is series of strings
s = df['col'].str.lower()
s = df['col'].str.upper()
s = df['col'].str.len()
df['col'] += 'suffix' # add text to each row
df['col'] *= 2        # repeat text
s = df['col1'] + df['col2'] # concatenate
```

Most python string functions are replicated in the pandas DataFrame and Series objects.

Regular expressions

```
s = df['col'].str.contains('regex')
s = df['col'].str.startswith('regex')
s = df['col'].str.endswith('regex')
s = df['col'].str.replace('old', 'new')
```

Note: pandas has many more regex methods

Working with missing and non-finite data

Working with missing data
Pandas uses the not-a-number construct (np.nan and float('nan')) to indicate missing data. The Python None can arise in data as well. It is also treated as missing data; as is the pandas not-a-time (pd.NaT) construct.

Missing data in a Series
```
s = pd.Series([8,None,float('nan'),np.nan])
# -->            [8,     NaN,   NaN,    NaN]
s.isnull() # --> [False, True, True,  True]
s.notnull()# --> [True, False, False, False]
```

Missing data in a DataFrame
```
df = df.dropna()  # drop all rows with a NaN
df = df.dropna(axis=1) # as above for cols
df=df.dropna(how='all') # only if all in row
df=df.dropna(thresh=2) # at least 2 NaN in r
# only drop row if NaN in a specified 'col'
df = df.dropna(df['col'].notnull())
```

Non-finite numbers
With floating point numbers, pandas provides for positive and negative infinity.
```
s = Series([float('inf'), float('-inf'),
     np.inf, -np.inf]) # inf, -inf, inf, -inf
```
Pandas treats integer comparisons with plus or minus infinity as expected.

Testing for finite numbers
(using the data from the previous example)
```
np.isfinite(s) # False, False, False, False
```

Working with Categorical Data

Categorical data
The pandas Series has an R factors-like data type for encoding categorical data into integers.
```
c = pd.Categorical.from_array(list)
c.levels   # --> the coding frame
c.labels   # --> the encoded integer array
c.describe # --> the values and levels
```

Indexing categorical data
The categorical data can be indexed in a manner conceptually similar to that for Series.iloc[] above:
```
listy = ['a', 'b', 'a', 'b', 'b', 'c']
c = pd.Categorical.from_array(listy)
c.levels       # --> ['a', 'b', 'c']
c.labels       # --> [0, 1, 0, 1, 1, 2]
x = c[1]       # --> 'b'
x = c[[0,1]]   # --> ['a', 'b']
x = c[0:2]     # --> ['a', 'b']
```

Categorical into DataFrame
You can put a column of encoded Categorical data in the DataFrame, but in the process the factor information will be lost; so you will need to hold this factor information outside of the DataFrame.
```
factor = pd.Categorical.from_array(df['cat'])
df['labels'] = factor.labels # integers only
df['cat2'] = factor # converts back to string
```

Basic Statistics

Summary statistics
```
s = df['col1'].describe()
df1 = df.describe()
```

Value counts
```
s = df['col1'].value_counts()
```

Cross-tabulation (frequency count)
```
ct = pd.crosstab(index=df['a'], cols=df['b'])
```

Quantiles and ranking
```
q = df.quantile(q=[0.05,0.25,0.5,0.75,0.95])
r = df.rank()
```

Histogram binning
```
count, bins = np.histogram(df['col1'])
count, bins = np.histogram(df['col'], bins=5)
count, bins = np.histogram(df['col1'],
                   bins=[-3,-2,-1,0,1,2,3,4])
```

Correlation and covariance
```
df_cm = df.corr()
df_cv = df.cov()
```

Regression
```
import statsmodels.formula.api as sm
result = sm.ols(formula="col1 ~ col2 + col3",
                data=df).fit()
print (result.params)
print (result.summary())
```

Smoothing example using rolling_apply
```
k3x5 = np.array([1,2,3,3,3,2,1]) / 15.0
s = pd.rolling_apply(df['col1'], window=7,
         func=lambda x: (x * k3x5).sum(),
          min_periods=7, center=True)
```