Cheat Sheet: The pandas DataFrame Object

Preliminaries

Always start by importing these Python modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas import DataFrame, Series
```

These are the standard import aliases most programmers use with Python pandas.

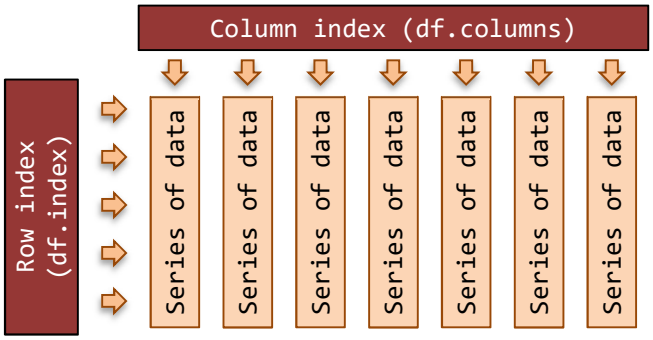
Check which version of pandas you are using

```
print(pd.__version__)
```

This cheat sheet was written for pandas version 0.23 and Python version 3.6.

The conceptual model

The **DataFrame object** is a two-dimensional table of data with column and row indexes (something like a spread sheet). The columns are made up of Series objects (see below).



A DataFrame has two Indexes:

- Typically, the column index (df.columns) is a list of strings (variable names) or (less commonly) integers
- Typically, the row index (df.index) might be:
  - Integers – for case or row numbers;
  - Strings – for case names; or
  - DatetimeIndex or PeriodIndex – for time series

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

Examples of Series Arithmatic

```
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([1, 2, 3], index=[0, 1, 2])
s5 = Series([1, 2, 3], index=[2, 1, 0])
s6 = s4 + s5                # -> 4, 4, 4

s7 = Series([1, 2, 3], index=[1, 2, 3])
s8 = Series([1, 2, 3], index=[0, 1, 2])
s9 = s7 + s8                # NAN, 3, 5, NAN
```

Get your data into a DataFrame

Instantiate an empty DataFrame

```
df = DataFrame()
```

Load a DataFrame from a CSV file

```
df = pd.read_csv('file.csv') # often works
df = pd.read_csv('file.csv', header=0,
    index_col=0, quotechar='"', sep=':',
    na_values = ['na', '-', '.', ''])
```

Note: refer to pandas docs for all arguments

Get data from inline CSV text to a DataFrame

```
from io import StringIO
data = """, Animal, Cuteness, Desirable
row-1,      dog,    8.7,  True
row-2,      cat,    9.5,  True
row-3,      bat,    2.6,  False"""
df = pd.read_csv(StringIO(data), header=0,
    index_col=0, skipinitialspace=True)
```

Note: *skipinitialspace=True* allows for a pretty layout

Load DataFrames from a Microsoft Excel file

```
# Each Excel sheet in a Python dictionary
workbook = pd.ExcelFile('file.xlsx')
d = {} # start with an empty dictionary
for sheet_name in workbook.sheet_names:
    df = workbook.parse(sheet_name)
    d[sheet_name] = df
```

Note: the parse() method takes many arguments like read_csv() above. Refer to the pandas documentation.

Load a DataFrame from a MySQL database

```
import pymysql
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://' +
    'USER:PASSWORD@HOST/DATABASE')
df = pd.read_sql_table('table', engine)
```

DataFrame from Series as rows (using a Python list)

```
s1 = Series(range(6))
s2 = s1 * s1
df = DataFrame([s1, s2])
df.index = ['n', 'n-squared']
```

DataFrame from Series as cols (using a dictionary)

```
s1 = Series({'Tom':1, 'Dick':2, 'Harry':3})
s2 = Series({'Tom':9, 'Dick':8, 'Mary':7})
df = DataFrame({'A': s1, 'B': s2 })
```

Note: misaligned rows; row order can change

DataFrame from Series as cols (using pd.concat())

```
df = pd.concat([s1, s2], axis=1, sort=False)
```

DataFrame from Series as cols (using transpose)

```
df = DataFrame([s1, s2], index=['A','B']).T
```

Note: data should all be of the same type

DataFrame from row data in a Python dictionary

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

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

DataFrame of fake data – useful for testing

```
df = DataFrame(np.random.rand(500,5),
        columns=list('ABCDE'))
```

DataFrame of fake time-series data

```
df = DataFrame(np.random.rand(500,5)) - 0.5
df = df.cumsum()
df.index = pd.date_range('1/1/2017',
               periods=len(df), freq='D')
```

Fake data with alphabetic index and group variable

```
import string
import random
rows = 52
cols = 5
assert(1 <= rows <= 52) # min/max row count
df = DataFrame(np.random.randn(rows, cols),
   columns=['c'+str(i) for i in range(cols)],
   index=list((string.ascii_uppercase +
         string.ascii_lowercase)[0:rows]))
df['groupable'] = [random.choice('abcde')
    for _ in range(rows)]
```

Saving a DataFrame

Saving a DataFrame to a CSV file

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

Saving DataFrames to an Excel Workbook

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

Saving a DataFrame to MySQL

```
import pymysql
from sqlalchemy import create_engine
e = create_engine('mysql+pymysql://' +
    'USER:PASSWORD@HOST/DATABASE')
df.to_sql('TABLE', e, if_exists='replace')
```

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

Saving to Python objects

```
d = df.to_dict()        # to dictionary
str = df.to_string()    # to string
m = df.as_matrix()      # to numpy matrix
```

Working with the whole DataFrame

Peek at the DataFrame contents/structure

```
df.info()           # index & data types
dfh = df.head(i)    # get first i rows
dft = df.tail(i)    # get last i rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:4, :4]
```

DataFrame non-indexing attributes

```
df = df.T       # transpose rows and cols
l = df.axes     # list row and col indexes
(r_idx, c_idx) = df.axes      # from above
s = df.dtypes   # Series column data types
b = df.empty    # True for empty DataFrame
i = df.ndim     # number of axes (it is 2)
t = df.shape    # (row-count, column-count)
i = df.size     # row-count * column-count
a = df.values   # get a numpy array for df
```

DataFrame utility methods

```
df = df.copy()  # copy a DataFrame
df = df.sort_values(by='col') # axis=1 cols
df = df.sort_values(by=['col1', 'col2'])
df = df.sort_values(by='row', axis=1)
df = df.sort_index()  # axis=1 to sort cols
df = df.astype(dtype) # type conversion
```

DataFrame iteration methods

```
df.iteritems()  # (col-index, Series) pairs
df.iterrows()   # (row-index, Series) pairs
# example ... iterating over columns ...
for (name, series) in df.iteritems():
    print('\nCol name: ' + str(name))
    print('1st value: ' + str(series.iat[0]))
```

Maths on the whole DataFrame (not a complete list)

```
df = df.abs()     # absolute values
df = df.add(o)    # add df, Series or value
s =  df.count()   # non NA/null values
df = df.cummax()  # (cols default axis)
df = df.cummin()  # (cols default axis)
df = df.cumsum()  # (cols default axis)
df = df.diff()    # 1st diff (col def axis)
df = df.div(o)    # div by df, Series, value
df = df.dot(o)    # matrix dot product
s =  df.max()     # max of axis (col def)
s =  df.mean()    # mean (col default axis)
s =  df.median()  # median (col default)
s =  df.min()     # min of axis (col def)
df = df.mul(o)    # mul by df Series val
s =  df.sum()     # sum axis (cols default)
df = df.where(df > 0.5, other=np.nan)
```

Note: methods returning a series default to work on cols

Select/filter rows/cols based on index label values

```
df = df.filter(items=['a', 'b'])  # by col
df = df.filter(items=[5], axis=0) # by row
df = df.filter(like='x') # keep x in col
df = df.filter(regex='x') # regex in col
df = df.select(lambda x: not x%5) # 5th rows
```

Note: select takes a Boolean function, for cols: axis=1
Note: filter defaults to cols; select defaults to rows

Working with Columns (and pandas Series)

Peek at the column/Series structure/contents

```
sh = df['col'].head(i) # get first i elements
st = df['col'].tail(i) # get last i elements
s = df['col'].describe() # summary stats
```

Get column index and labels

```
idx = df.columns        # get col index
label = df.columns[0]   # first col label
l = df.columns.tolist() # list of col labels
a = df.columns.values   # array of col labels
```

Change column labels

```
df = df.rename(columns={'old':'new','a':'1'})
df.columns = ['new1', 'new2', 'new3'] # etc.
```

Selecting columns

```
s =  df['colName']     # select col to Series
df = df[['colName']]   # select col to df
df = df[['a','b']]     # select 2-plus cols
df = df[['c','a','b']] # change col order
s =  df[df.columns[0]] # select by number
df = df[df.columns[[0, 3, 4]]] # by numbers
df = [df.columns[:-1]] # all but last col
s =  df.pop('c')       # get & drop from df
```

Selecting columns with Python attributes

```
s = df.a    # same as s = df['a']
df.existing_column = df.a / df.b
df['new_column'] = df.a / df.b
```

Trap: column names must be valid Python identifiers.
Trap: cannot create new columns by attribute.

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']]
```

Trap: When adding a new column, only items from the new column series that have a corresponding index in the DataFrame will be added. The index of the receiving DataFrame is not extended to accommodate all of the new series.
Trap: when adding a python list or numpy array, the column will be added by integer position.

Add a mismatched column with an extended index

```
df = DataFrame([1, 2, 3], index=[1, 2, 3])
s = Series([2, 3, 4], index=[2, 3, 4])
df = df.reindex(df.index.union(s.index))
df['s'] = s # with NaNs where no data
```

Note: assumes unique index values

Dropping (deleting) columns (mostly by label)

```
df = df.drop('col1', axis=1)
df = df.drop(['col1', 'col2'], axis=1)
del df['col'] # even classic python works
df = df.drop(df.columns[0], axis=1)   #first
df = df.drop(df.columns[-1:], axis=1) #last
```

Swap column contents

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

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'])
```

Note: many many more numpy math functions
Hint: Prefer pandas math over numpy where you can.

Set column values set based on criteria

```
df['b'] = df['a'].where(df['a']>0, other=0)
df['d'] = df['a'].where(df.b!=0, other=df.c)
```

Note: where other can be a Series or a scalar

Data type conversions

```
df['float'] =  df['col'].astype(float)
df['int'] =  df['col'].astype(int)
s = df['col'].astype(str) # Series dtype
a = df['col'].values      # numpy array
l = df['col'].tolist()    # python list
```

Trap: index lost in conversion from Series to array or list

Common column-wide methods/attributes

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

Find first index label for min/max values in column

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

Common column element-wise methods

```
s = df['col'].isnull()
s = df['col'].notnull()    # not isnull()
s = df['col'].astype(float)
s = df['col'].abs()
s = df['col'].round(decimals=0)
s = df['col'].diff(periods=1)
s = df['col'].shift(periods=1)
s = df['col'].to_datetime()
s = df['col'].fillna(0)    # replace NaN w 0
s = df['col'].cumsum()
s = df['col'].cumprod()
s = df['col'].pct_change(periods=4)
s = df['col'].rolling(window=4,
    min_periods=4, center=False).sum()
```

Append a column of row sums to a DataFrame

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

Note: also means, mins, maxs, etc.

Multiply every column in DataFrame by a Series

```
df = df.mul(s, axis=0)  # on matched rows
```

Note: also add, sub, div, etc.

Selecting columns with .loc, .iloc

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

Get the integer position of a column index label

```
i = df.columns.get_loc('col_name')
```

Test if column index values are unique/monotonic

```
if df.columns.is_unique: pass # ...
b = df.columns.is_monotonic_increasing
b = df.columns.is_monotonic_decreasing
```

Mapping a DataFrame column or Series

```
map = Series(['red', 'green', 'blue'],
        index=['r', 'g', 'b'])
s = Series(['r', 'g', 'r', 'b']).map(map)
# s contains: ['red', 'green', 'red', 'blue']

m = Series([True, False], index=['Y', 'N'])
df = DataFrame(np.random.choice(list('YN'),
        500, replace=True), columns=['col'])
df['col'] = df['col'].map(m)
```

Note: Useful for decoding data before plotting
Note: Sometimes referred to as a lookup function
Note: Indexes can also be mapped if needed.

Find the largest and smallest values in a column

```
s = df['A'].nlargest(5)
s = df['A'].nsmallest(5)
```

Sorting the columns of a DataFrame

```
df = df.sort_index(axis=1, ascending=False)
```

Working with rows

Get the row index and labels

```
idx = df.index          # get row index
label = df.index[0]     # first row label
label = df.index[-1]    # last row label
l = df.index.tolist()   # get as a list
a = df.index.values     # get as an array
```

Change the (row) index

```
df.index = idx            # new ad hoc index
df = df.set_index('A')    # col A new index
df = df.set_index(['A', 'B']) # MultiIndex
df = df.reset_index()     # replace old w new
# note: old index stored as a col in df
df.index = range(len(df)) # set with list
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['r1','r2','etc'])
```

Adding rows

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

Hint: convert row to a DataFrame and then append.
Both DataFrames must have same column 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])]
df = df[df['col'].str.contains('hello')]
```

Trap: bitwise "or", "and" “not; (ie. | & ~) co-opted to be Boolean operators on a Series of Boolean
Trap: need parentheses around comparisons.

Selecting rows using isin over multiple columns

```
# fake up some data
data = {1:[1,2,3], 2:[1,4,9], 3:[1,8,27]}
df = DataFrame(data)

# multi-column isin
lf = {1:[1, 3], 3:[8, 27]} # look for
f = df[df[list(lf)].isin(lf).all(axis=1)]
```

Selecting rows using an index

```
idx = df[df['col'] >= 2].index
print(df.loc[idx])
```

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

```
df = df[:]       # copy entire DataFrame
df = df[0:2]     # rows 0 and 1
df = df[2:3]     # row 2 (the third row)
df = df[-1:]     # the last 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 label for integer numbered columns.

Select a slice of rows by label/index

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

Note: [inclusive-from : inclusive–to [: step]]
Trap: cannot work for integer labelled rows – see previous code snippet on integer position slicing.

Append a row of column totals to a DataFrame

```
# Option 1: use 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)
```

Iterating over DataFrame rows

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

Trap: row data type may be coerced.

Sorting the rows of a DataFrame by the row index

```
df = df.sort_index(ascending=False)
```

Sorting DataFrame rows based on column values

```
df = df.sort_values(by=df.columns[0],
                    ascending=False)
df = df.sort_values(by=['col1', 'col2'])
```

Random selection of rows

```
import random as r
k = 20 # pick a number
selection = r.sample(range(len(df)), k)
df_sample = df.iloc[selection, :] # get copy
```

Note: this randomly selected sample is not sorted

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 = df.sort_index()      # 4 tidy up
```

Test if two DataFrames have same row index

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

Get the integer position of a row or col index label

```
i = df.index.get_loc('row_label')
```

Trap: index.get_loc() returns an integer for a unique match. If not a unique match, may return a slice/mask.

Get integer position of rows that meet condition

```
a = np.where(df['col'] >= 2) #numpy array
```

Test if the row index values are unique/monotonic

```
if df.index.is_unique: pass # ...
b = df.index.is_monotonic_increasing
b = df.index.is_monotonic_decreasing
```

Find row index duplicates

```
if df.index.has_duplicates:
    print(df.index.duplicated())
```

Note: also similar for column label duplicates.

Working with cells

Getting a cell by row and column labels

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

Note: .at[] fastest label based scalar lookup
Note: at[] does not take slices as an argument

Setting a cell by row and column labels

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

Getting 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'] = othr.loc[1:2,'A':'C']
```

Remember: inclusive "to" in the slice

Getting 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]
```

Getting a range of cells by int position

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

Note: exclusive "to" – same as python 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))
df.iloc[1:3, 1:4] = np.zeros((2, 3))
df.iloc[1:3, 1:4] = np.array([[1, 1, 1],
                              [2, 2, 2]])
```

Remember: exclusive-to in the slice

Views and copies

From the manual: Setting a copy can cause subtle errors. 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.

Summary: selecting using the DataFrame index

Using the DataFrame index to select columns

```
s =  df['col_label']         # returns Series
df = df[['col_label']]    # returns DataFrame
df = df[['L1', 'L2']] # select cols with list
df = df[index]    # select cols with an index
df = df[s]     # select with col label Series
```

Note: scalar returns Series; list &c returns a DataFrame.

Using the DataFrame index to select rows

```
df = df['from':'inc_to']  # label slice
df = df[3:7]              # integer slice
df = df[df['col'] > 0.5]  # Boolean Series
df = df.loc['label']      # single label
df = df.loc[container]    # lab list/Series
df = df.loc['from':'to']  # inclusive slice
df = df.loc[bs]           # Boolean Series
df = df.iloc[0]           # single integer
df = df.iloc[container]   # int list/Series
df = df.iloc[0:5]         # exclusive slice
```

Trap: Boolean Series gets rows, label Series gets cols.

Using the DataFrame index to select a cross-section

```
# r and c can be scalar, list, slice
df.loc[r, c]   # label accessor (row, col)
df.iloc[r, c]  # integer accessor
df[c].iloc[r]  # chained – also for .loc
```

Using the DataFrame index to select a cell

```
# r and c must be label or integer
df.at[r, c]    # fast scalar label accessor
df.iat[r, c]   # fast scalar int accessor
df[c].iat[r]   # chained – also for .at
```

DataFrame indexing methods

```
v = df.get_value(r, c)    # get by row, col
df = df.set_value(r,c,v)  # set by row, col
df = df.xs(key, axis)     # get cross-section
df = df.filter(items, like, regex, axis)
df = df.select(crit, axis)
```

Note: the indexing attributes (.loc, .iloc, .at .iat) can be used to get and set values in the DataFrame.
Note: the .loc, and iloc indexing attributes can accept python slice objects. But .at and .iat do not.
Note: .loc can also accept Boolean Series arguments
Avoid: chaining in the form df[col_indexer][row_indexer]
Trap: label slices are inclusive, integer slices exclusive.

Some index attributes and methods

```
b = idx.is_monotonic_decreasing
b = idx.is_monotonic_increasing
b = idx.has_duplicates
i = idx.nlevels        # num of index levels
idx = idx.astype(dtype)# change data type
b = idx.equals(o)      # check for equality
idx = idx.union(o)     # union of two indexes
i = idx.nunique()      # number unique labels
label = idx.min()      # minimum label
label = idx.max()      # maximum label
```

Joining/Combining DataFrames

Three ways to join two DataFrames:

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

Merge on (row) 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)

# multi-combine with python reduce()
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.

Groupby: Split-Apply-Combine

Grouping

```
gb = df.groupby('cat') # by one columns
gb = df.groupby(['c1','c2']) # by 2 cols
gb = df.groupby(level=0) # multi-index gb
gb = df.groupby(level=['a','b']) # mi gb
print(gb.groups)
```

Note: groupby() returns a pandas groupby object

Note: the groupby object attribute .groups contains a dictionary mapping of the groups.
Trap: NaN values in the group key are automatically dropped – there will never be a NA group.

The pandas "groupby" mechanism allows us to split the data into groups, apply a function to each group independently and then combine the results.

Iterating groups – usually not needed

```
for name, group in gb:
    print (name, group)
```

Selecting a group

```
dfa = df.groupby('cat').get_group('a')
dfb = df.groupby('cat').get_group('b')
```

Applying an aggregating function

```
# apply to a column ...
s = df.groupby('cat')['col1'].sum()
s = df.groupby('cat')['col1'].agg(np.sum)
# apply to the every column in DataFrame
s = df.groupby('cat').agg(np.sum)
df_summary = df.groupby('cat').describe()
df_row_1s = df.groupby('cat').head(1)
```

Note: aggregating functions reduce the dimension by one – they include: mean, sum, size, count, std, var, sem, describe, first, last, min, max

Applying multiple aggregating functions

```
gb = df.groupby('cat')
# apply multiple functions to one column
dfx = gb['col2'].agg([np.sum, np.mean])
# apply to multiple fns to multiple cols
dfy = gb.agg({
    'cat': np.count_nonzero,
    'col1': [np.sum, np.mean, np.std],
    'col2': [np.min, np.max]
})
```

Note: gb['col2'] above is shorthand for df.groupby('cat')['col2'], without the need for regrouping.

Transforming functions

```
# transform to group z-scores, which have
# a group mean of 0, and a std dev of 1.
zscore = lambda x: (x-x.mean())/x.std()
dfz = df.groupby('cat').transform(zscore)
# replace missing data with group mean
mean_r = lambda x: x.fillna(x.mean())
dfm = df.groupby('cat').transform(mean_r)
```

Note: can apply multiple transforming functions in a manner similar to multiple aggregating functions above,

Applying filtering functions
Filtering functions allow you to make selections based on whether each group meets specified criteria

```
# select groups with more than 10 members
eleven = lambda x: (len(x['col1']) >= 11)
df11 = df.groupby('cat').filter(eleven)
```

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()
```

Pivot Tables: working with long and wide data

These features work with and often create hierarchical or multi-level Indexes; (the pandas MultiIndex is powerful and complex).

Pivot, unstack, stack and melt
Pivot tables move from long format to wide format data

```
# Let's start with data in long format
from StringIO import StringIO # python2.7
#from io import StringIO      # python 3
data = """Date,Pollster,State,Party,Est
13/03/2014, Newspoll, NSW,   red,   25
13/03/2014, Newspoll, NSW,   blue,  28
13/03/2014, Newspoll, Vic,   red,   24
13/03/2014, Newspoll, Vic,   blue,  23
13/03/2014, Galaxy,   NSW,   red,   23
13/03/2014, Galaxy,   NSW,   blue,  24
13/03/2014, Galaxy,   Vic,   red,   26
13/03/2014, Galaxy,   Vic,   blue,  25
13/03/2014, Galaxy,   Qld,   red,   21
13/03/2014, Galaxy,   Qld,   blue,  27"""
df = pd.read_csv(StringIO(data),
      header=0, skipinitialspace=True)

# pivot to wide format on 'Party' column
# 1st: set up a MultiIndex for other cols
df1 = df.set_index(['Date', 'Pollster',
    'State'])
# 2nd: do the pivot
wide1 = df1.pivot(columns='Party')

# unstack to wide format on State / Party
# 1st: MultiIndex all but the Values col
df2 = df.set_index(['Date', 'Pollster',
    'State', 'Party'])
# 2nd: unstack a column to go wide on it
wide2 = df2.unstack('State')
wide3 = df2.unstack() # pop last index

# Use stack() to get back to long format
long1 = wide1.stack()
# Then use reset_index() to remove the
# MultiIndex.
long2 = long1.reset_index()

# Or melt() back to long format
# 1st: flatten the column index
wide1.columns = ['_'.join(col).strip()
    for col in wide1.columns.values]
# 2nd: remove the MultiIndex
wdf = wide1.reset_index()
# 3rd: melt away
long3 = pd.melt(wdf, value_vars=
    ['Est_blue', 'Est_red'],
    var_name='Party', id_vars=['Date',
    'Pollster', 'State'])
```

Note: See documentation, there are many arguments to these methods.

Working with dates, times and their indexes

Dates and time – points and spans
With its focus on time-series data, pandas has 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).

```
t = pd.Timestamp('2013-01-01')
t = pd.Timestamp('2013-01-01 21:15:06')
t = pd.Timestamp('2013-01-01 21:15:06.7')
p = pd.Period('2013-01-01', freq='M')
```

Note: Timestamps can range from 1678 to 2261. (Check pd.Timestamp.max and pd.Timestamp.min).

A Series of Timestamps or Periods

```
ts = ['2015-04-01', '2014-04-02']

# Series of Timestamps
s = pd.to_datetime(pd.Series(ts))

# Series of Periods
s = s.dt.to_period('M') # from Timestamps
```

Note: While Periods make a very useful index; they may be less useful in a Series.
Trap: pd.to_datetime(list_of_timestamp_strings) returns a pandas DatetimeIndex object.

From non-standard strings to Timestamps

```
t = ['09:08:55.7654-JAN092002',
        '15:42:02.6589-FEB082016']
s = pd.Series(pd.to_datetime(t,
        format="%H:%M:%S.%f-%b%d%Y"))
```

Also: %B = full month name; %m = numeric month; %y = year without century; and more …

Dates and time – stamps and spans as indexes
An index of Timestamps is a DatetimeIndex.
An index of Periods is a PeriodIndex.

```
date_strs = ['2018-01-01', '2018-04-01',
             '2018-07-01', '2018-10-01']

dti = pd.DatetimeIndex(date_strs)

pid = pd.PeriodIndex(date_strs, freq='D')
pim = pd.PeriodIndex(date_strs, freq='M')
piq = pd.PeriodIndex(date_strs, freq='Q')

print (pid[1] - pid[0])  # 90  [days]
print (pim[1] - pim[0])  #  3  [months]
print (piq[1] - piq[0])  #  1  [quarter]

time_strs = ['2015-01-01 02:10:40.12345',
             '2015-01-01 02:10:50.67890']
pis = pd.PeriodIndex(time_strs, freq='U')

df.index = pd.period_range('2015-01',
    periods=len(df), freq='M')

dti = pd.to_datetime(['04-01-2012'],
    dayfirst=True) # Australian date format
pi = pd.period_range('1960-01-01',
    '2015-12-31', freq='M')
```

Hint: unless you are working in less than seconds, prefer PeriodIndex over DatetimeImdex.

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)

DatetimeIndex from DataFrame columns

```
datecols = ['year', 'month', 'day']
df.index = pd.to_datetime(df[datecols])
```

From DatetimeIndex to Python datetime objects

```
dti = pd.DatetimeIndex(pd.date_range(
  start='1/1/2011', periods=4, freq='M'))
s = Series([1,2,3,4], index=dti)
a = dti.to_pydatetime()       # numpy array
a = s.index.to_pydatetime()   # numpy array
```

From Timestamps to Python dates or times

```
df['py_date'] = [x.date() for x in df['TS']]
df['py_time'] = [x.time() for x in df['TS']]
```

Note: converts to datatime.date or datetime.time. But does not convert to datetime.datetime.

From DatetimeIndex to PeriodIndex and back

```
df = DataFrame(np.random.randn(20,3))
df.index = pd.date_range('2015-01-01',
       periods=len(df), freq='M')
dfp = df.to_period(freq='M')
dft = dfp.to_timestamp()
```

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

Working with a PeriodIndex

```
pi = pd.period_range('1960-01','2015-12',
                     freq='M')
a = pi.values   # numpy array of integers
p = pi.tolist() # python list of Periods
sp = Series(pi)  # pandas Series of Periods
s = Series(pi).astype(str) # Series of strs
l = Series(pi).astype(str).tolist()
```

Get a range of Timestamps

```
dr = pd.date_range('2013-01-01',
    '2013-12-31', freq='D')
```

Error handling with dates

```
# 1st example returns string not Timestamp
t = pd.to_datetime('2014-02-30')
# 2nd example returns NaT (not a time)
t = pd.to_datetime('2014-02-30', coerce=True)
# NaT like NaN tests True for isnull()
b = pd.isnull(t) # --> True
```

The tail of a time-series DataFrame

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

Upsampling and downsampling

```
# upsample from quarterly to monthly
pi = pd.period_range('1960Q1',
        periods=220, freq='Q')
df = DataFrame(np.random.rand(len(pi),5),
        index=pi)
dfm = df.resample('M', convention='end')
# use ffill or bfill to fill with values

# downsample from monthly to quarterly
dfq = dfm.resample('Q', how='sum')
```

Time zones

```
t = ['2015-06-30 00:00:00',
      '2015-12-31 00:00:00']
dti = pd.to_datetime(t
      ).tz_localize('Australia/Canberra')
dti = dti.tz_convert('UTC')
ts = pd.Timestamp('now',
      tz='Europe/London')

# get a list of all time zones
import pyzt
for tz in pytz.all_timezones:
    print tz
```

Note: by default, Timestamps are created without time zone information.

Row selection with a time-series index

```
# start with the play data above
idx = pd.period_range('2015-01',
        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)]

mayornov_data = df[(df.index.month == 5) |
   (df.index.month == 11)]

totals = 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], …

The Series.dt accessor attribute

DataFrame columns that contain datetime-like objects can be manipulated with the .dt accessor attribute

```
t = ['2012-04-14 04:06:56.307000',
     '2011-05-14 06:14:24.457000',
     '2010-06-14 08:23:07.520000']

# a Series of time stamps
s = pd.Series(pd.to_datetime(t))
print(s.dtype)      # datetime64[ns]
print(s.dt.second)  # 56, 24,  7
print(s.dt.month)   #  4,  5,  6
# a Series of time periods
s = pd.Series(pd.PeriodIndex(t,freq='Q'))
print(s.dtype)      # datetime64[ns]
print(s.dt.quarter) #  2,  2,  2
print(s.dt.year)    #  2012, 2011, 2010
```

Plotting from the DataFrame

Import matplotlib, choose a matplotlib style

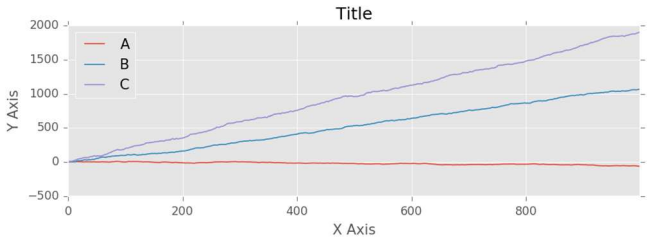
```
import matplotlib.pyplot as plt
print(plt.style.available)
plt.style.use('ggplot')
```

Fake up some data (which we reuse repeatedly)

```
a = np.random.normal(0,1,999)
b = np.random.normal(1,2,999)
c = np.random.normal(2,3,999)
df = pd.DataFrame([a,b,c]).T
df.columns =['A', 'B', 'C']
```

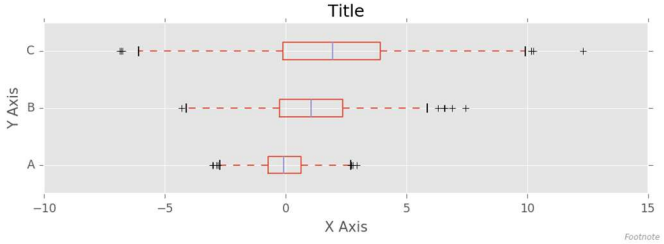
Line plot

```
df1 = df.cumsum()
ax = df1.plot()

# from here down - standard plot output
ax.set_title('Title')
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')

fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```



Box plot

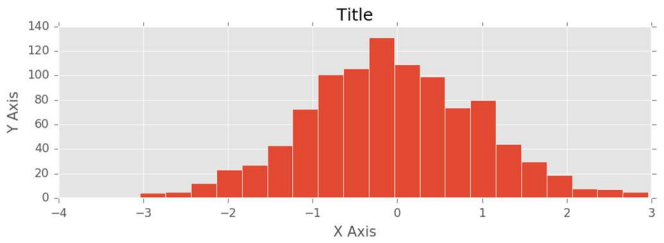
```
ax = df.plot.box(vert=False)
# followed by the standard plot code as above
```




```
ax = df.plot.box(column='c1', by='c2')
```

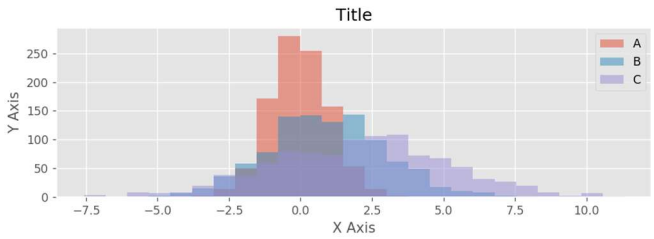
Histogram

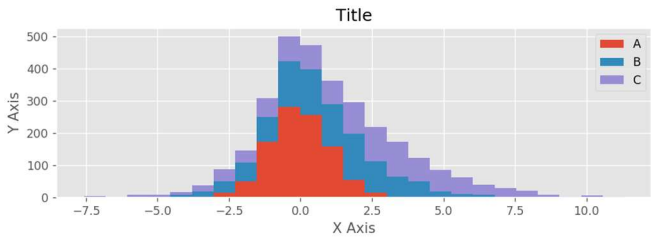
```
ax = df['A'].plot.hist(bins=20)
# followed by the standard plot code as above
```



Multiple histograms (overlapping or stacked)

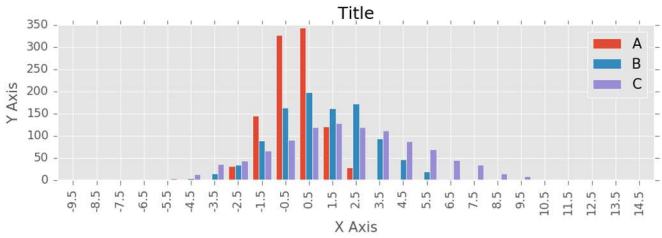
```
ax = df.plot.hist(bins=25, alpha=0.5) # or...
ax = df.plot.hist(bins=25, stacked=True)
# followed by the standard plot code as above
```





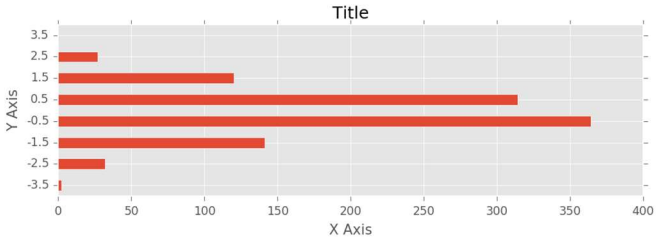
Bar plots

```
bins = np.linspace(-10,15,26)
binned = pd.DataFrame()
for x in df.columns:
    y=pd.cut(df[x],bins,labels=bins[:-1])
    y=y.value_counts().sort_index()
    binned = pd.concat([binned,y],axis=1)
binned.index = binned.index.astype(float)
binned.index += (np.diff(bins) / 2.0)
ax = binned.plot.bar(stacked=False,
    width=0.8) # for bar width
# followed by the standard plot code as above
```



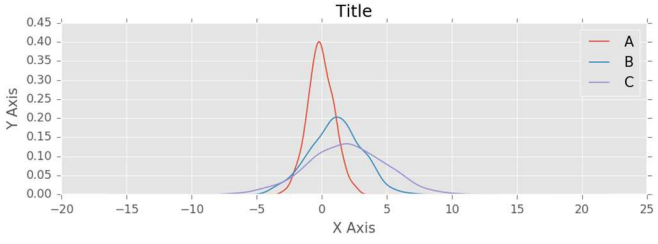
Horizontal bars

```
ax = binned['A'][(binned.index >= -4) &
    (binned.index <= 4)].plot.barh()
# followed by the standard plot code as above
```



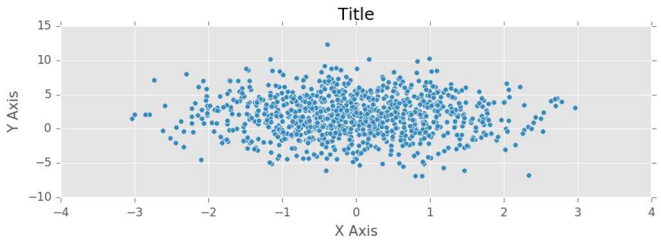
Density plot

```
ax = df.plot.kde()
# followed by the standard plot code as above
```



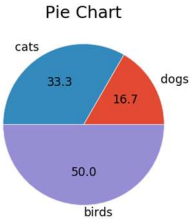
Scatter plot

```
ax = df.plot.scatter(x='A', y='C')
# followed by the standard plot code as above
```



Pie chart

```
s = pd.Series(data=[10, 20, 30],
    index = ['dogs', 'cats', 'birds'])
ax = s.plot.pie(autopct='%.1f')

# followed by the standard plot output ...
ax.set_title('Pie Chart')
ax.set_aspect(1) # make it round
ax.set_ylabel('') # remove default

fig = ax.figure
fig.set_size_inches(8, 3)
fig.savefig('filename.png', dpi=125)

plt.close(fig)
```

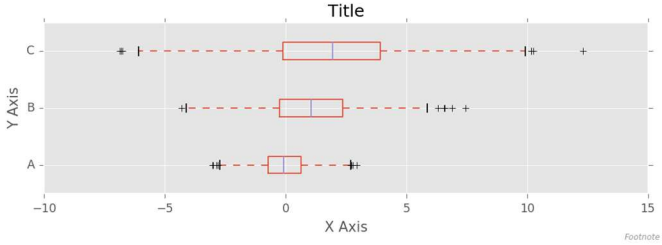


Change the range plotted

```
ax.set_xlim([-5, 5])

# for some white space on the chart ...
lower, upper = ax.get_ylim()
ax.set_ylim([lower-1, upper+1])
```

Add a footnote to the chart

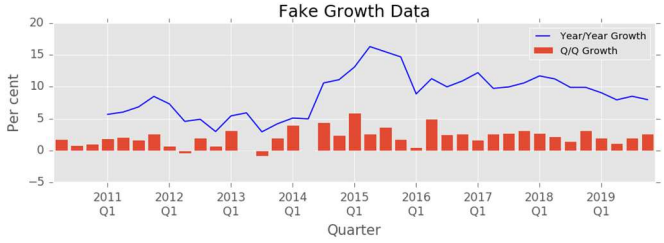
```
# after the fig.tight_layout(pad=1) above
fig.text(0.99, 0.01, 'Footnote',
    ha='right', va='bottom',
    fontsize='x-small',
    fontstyle='italic', color='#999999')
```



A line and bar on the same chart

In matplotlib, bar charts visualise categorical or discrete data. Line charts visualise continuous data. This makes it hard to get bars and lines on the same chart. Typically combined charts either have too many labels, and/or the lines and bars are misaligned or missing. You need to trick matplotlib a bit … pandas makes this tricking easier

```
# start with fake percentage growth data
s = pd.Series(np.random.normal(
    1.02, 0.015, 40))
s = s.cumprod()
dfg = (pd.concat([s / s.shift(1),
    s / s.shift(4)], axis=1) * 100) - 100
dfg.columns = ['Quarter', 'Annual']
dfg.index = pd.period_range('2010-Q1',
    periods=len(dfg), freq='Q')

# reindex with integers from 0; keep old
old = dfg.index
dfg.index = range(len(dfg))

# plot the line from pandas
ax = dfg['Annual'].plot(color='blue',
    label='Year/Year Growth')

# plot the bars from pandas
dfg['Quarter'].plot.bar(ax=ax,
    label='Q/Q Growth', width=0.8)

# relabel the x-axis more appropriately
ticks = dfg.index[((dfg.index+0)%4)==0]
labs = pd.Series(old[ticks]).astype(str)
ax.set_xticks(ticks)
ax.set_xticklabels(labs.str.replace('Q',
    '\nQ'), rotation=0)

# fix the range of the x-axis … skip 1st
ax.set_xlim([0.5,len(dfg)-0.5])

# add the legend
l=ax.legend(loc='best',fontsize='small')

# finish off and plot in the usual manner
ax.set_title('Fake Growth Data')
ax.set_xlabel('Quarter')
ax.set_ylabel('Per cent')

fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```



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 construct (pandas.NaT).

Missing data in a Series

```
s = Series( [8,None,float('nan'),np.nan])
          #[8,     NaN,   NaN,   NaN]
s.isnull() #[False, True,  True,  True]
s.notnull()#[True,  False, False, False]
s.fillna(0)#[8,     0,     0,     0]
```

Missing data in a DataFrame

```
df = df.dropna() # drop all rows with NaN
df = df.dropna(axis=1) # same for cols
df=df.dropna(how='all') #drop all NaN row
df=df.dropna(thresh=2) # drop 2+ NaN in r
# only drop row if NaN in a specified col
df = df.dropna(df['col'].notnull())
```

Recoding missing data

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

Non-finite numbers

With floating point numbers, pandas provides for positive and negative infinity.

```
s = Series([float('inf'), float('-inf'),
     np.inf, -np.inf])
```

Pandas treats integer comparisons with plus or minus infinity as expected.

Testing for finite numbers

(using the data from the previous example)

```
b = np.isfinite(s)
```

Working with Categorical Data

Categorical data

The pandas Series has an R factors-like data type for encoding categorical data.

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
df['B'] = df['A'].astype('category')
```

Note: the key here is to specify the "category" data type.
Note: categories will be ordered on creation if they are sortable. This can be turned off. See ordering below.

Convert back to the original data type

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
s = s.astype('string')
```

Ordering, reordering and sorting

```
s = Series(list('abc'), dtype='category')
print (s.cat.ordered)
s=s.cat.reorder_categories(['b','c','a'])
s = s.sort()
s.cat.ordered = False
```

Trap: category must be ordered for it to be sorted

Renaming categories

```
s = Series(list('abc'), dtype='category')
s.cat.categories = [1, 2, 3] # in place
s = s.cat.rename_categories([4,5,6])
# using a comprehension ...
s.cat.categories = ['Group ' + str(i)
    for i in s.cat.categories]
```

Trap: categories must be uniquely named

Adding new categories

```
s = s.cat.add_categories([4])
```

Removing categories

```
s = s.cat.remove_categories([4])
s.cat.remove_unused_categories() #inplace
```

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()

# the next set work like Python
df['col'] += 'suffix'       # append
df['col'] *= 2              # duplicate
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')
df['b'] = df.a.str.extract('(pattern)')
```

Note: pandas has many more regex methods.

Basic Statistics

Summary statistics

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

DataFrame – key stats methods

```
df.corr()     # pairwise correlation cols
df.cov()      # pairwise covariance cols
df.kurt()     # kurtosis over cols (def)
df.mad()      # mean absolute deviation
df.sem()      # standard error of mean
df.var()      # variance over cols (def)
```

Value counts

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

Cross-tabulation (frequency count)

```
ct = pd.crosstab(index=df['a'],
        cols=df['b'])
```

Quantiles and ranking

```
quants = [0.05, 0.25, 0.5, 0.75, 0.95]
q = df.quantile(quants)
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])
```

Regression

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

Simple smoothing example using a rolling apply

```
k3x5 = np.array([1,2,3,3,3,2,1]) / 15.0
s = df['A'].rolling(window=len(k3x5),
    min_periods=len(k3x5),
    center=True).apply(
         func=lambda x: (x * k3x5).sum())
# fix the missing end data ... unsmoothed
s = df['A'].where(s.isnull(), other=s)
```

Cautionary note

This cheat sheet was cobbled together by tireless bots roaming the dark recesses of the Internet seeking ursine and anguine myths from a fabled land of milk and honey where it is rumoured pandas and pythons gambol together. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. You have been warned. I will not be held responsible for whatever happens to you and those you love once your eyes begin to see what is written here.

Errors: If you find any errors, please email me at markthegraph@gmail.com; (but please do not correct my use of Australian-English spelling conventions).