Cheat Sheet: The pandas DataFrame Object

Preliminaries

Always start by importing these Python modules

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

**Note**: these are the recommended import aliases  
**Note**: you can put these into a PYTHONSTARTUP file

Cheat sheet conventions

Code examples

# Code examples are found in yellow boxes

In the code examples, typically I use:

- `s` to represent a pandas Series object;
- `df` to represent a pandas DataFrame object;
- `idx` to represent a pandas Index object.
- Also: `t` – tuple, `l` – list, `b` – Boolean, `i` – integer,  
  `a` – numpy array, `st` – string, `d` – dictionary, etc.

The conceptual model

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

**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.

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;
  - DatetimeIndex or PeriodIndex – for time series

Get your data into a DataFrame

Instantiate an empty DataFrame

```python
df = DataFrame()
```

Load a DataFrame from a CSV file

```python
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

```python
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"
```

```python
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

```python
# 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

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

Data in Series then combine into a DataFrame

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

```python
# Example 2 ...
s3 = Series({'Tom':1, 'Dick':4, 'Har':9})
s4 = Series({'Tom':3, 'Dick':2, 'Mar':5})
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

Get a DataFrame from a Python dictionary

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

Version 30 April 2017 - [Draft – Mark Graph – mark dot the dot graph at gmail dot com – @Mark_Graph on twitter]
Get a DataFrame from data in a Python dictionary

```python
# --- 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/fake data (useful for testing)

```python
# --- simple - default integer indexes
df = DataFrame(np.random.rand(50,5))

# --- with a time-stamp row index:
df = DataFrame(np.random.rand(500,5))
df.index = pd.date_range('1/1/2005', periods=len(df), freq='M')

# --- with alphabetic row and col indexes
#          and a "groupable" 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

```python
df.to_csv(‘name.csv’, encoding=’utf-8’)```

Maths on the whole DataFrame (not a complete list)

```python
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
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)
```

Select/filter rows/cols based on index label values

```python
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
```

Select a series using a boolean condition:

```python
df = df.filter(lambda x: x > 0.5)
```

Note: methods returning a series default to work on cols

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

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[df.columns[:-1]]  # all but last col
s = df.pop('c')  # gets & drops from df

Selecting columns with Python attributes

s = df.a  # same as s = df['a']
# cannot create new columns by attribute
def.existing_column = df.a / df.b
df['new_column'] = 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
df['new_col2'] = np.log(df['col1'])

Trap: When adding a new column, only items from the new series that have a corresponding index in the DataFrame will be added. The receiving DataFrame is not extended to accommodate the new series.

Trap: when adding a python list or numpy array, the column will be added by integer position.

Swap column contents

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

Dropping (deleting) columns (mostly by label)

df = df.drop('col1', axis=1)
df.drop('col1', axis=1, inplace=True)
df = df.drop(['col1','col2'], axis=1)
s = df.pop('col')  # drops from frame
def = df.drop(df.columns[0], axis=1)  # first
def = df.drop(df.columns[-1:],axis=1)  # last

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 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.f.c)

Note: where other can be a Series or a scalar

Data type conversions

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

Note: useful dtypes for Series conversion: int, float, str

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 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'].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, maxes, etc.

Multiply every column in DataFrame by Series

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

Note: also add, sub, div, etc.

Selecting columns with .loc, .iloc and .ix

df = df.loc[:, 'col1':'col2']  # inclusive

Get the integer position of a column index label

df = 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
Working with rows

Get the row index and labels

```python
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

```python
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'])
df.rename(index={'old':'new'}, inplace=True)
```

Adding rows

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

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

Dropping rows (by name)

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

Boolean row selection by values in a column

```python
df = df[df["col2"] >= 0.0] # cols
df = df[df["col"]>1.0 | (df["col"]<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

```python
# hack 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

```python
idx = df[df["col"] >= 2].index
print(df.ix[idx])
```

Select a slice of rows by label/index

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

Trap: cannot work for integer labelled rows – see
previous code snippet on integer position slicing.

Append a row of column totals to a DataFrame

```python
# 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

```python
for (index, row) in df.iterrows(): # pass
Trap: row data type may be coerced.
```

Sorting DataFrame rows values

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

Sort DataFrame by its row index

```python
df.sort_index(inplace=True) # sort by row
df = df.sort_index(ascending=False)
```

Random selection of rows

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

Note: this randomly selected sample is not sorted

Drop duplicates in the row index

```python
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 have same row index

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

Get the integer position of a row or col index label

```python
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

```python
a = np.where(df["col"] == 2)
```

Test if the row index values are unique/monotonic

```python
if df.index.is_unique: pass # ...
```

Find row index duplicates

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

Note: also similar for column label duplicates.
Working with cells

Selecting 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

Setting a cell by row and column labels

df.at['row', 'col'] = value
df.loc['row', 'col'] = value
df['col'].at['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 "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] # subset of the df
s = df.iloc[5, :] # return left corner 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 a range of cells by int 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

.ix for mixed label and integer position indexing

value = df.ix[5, 'col1']
df = df.ix[1:5, 'col1':'col3']

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 label col 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
df = df.ix[x] # loc then iloc

**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.ix[r, c] # label access int fallback
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, .ix, .at,.iat) can be used to get and set values in the DataFrame.

**Note:** the .loc, iloc and .ix 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.levels # 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 indexes

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

**How:** 'left', 'right', 'outer', 'inner'
**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.

### Merge on columns

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

**Use:** 
- combines the two DataFrames based on a common column.
- The index of the combined DataFrame will be the union of the indexes from df1 and df2.

### Simple concatenation is often the best

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

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

### Combine_first

```
df = df1.combine_first(other=df2)
```

**Use:** 
- combines the two DataFrames based on a common column.

### Groupby: Split-Apply-Combine

**Grouping**

```
# by one columns
gb = df.groupby('cat')
# by 2 cols
gb = df.groupby(['c1', 'c2'])
```

**Note:**
- groupby() returns a pandas groupby object
- gb.m1 returns a dictionary mapping of the groups.

**Trapping NaN values**

- NaN values in the group key are automatically dropped – there will never be a NaN group.

### Iterating groups – usually not needed

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

### Selecting a group

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

**Use:** 
- apply a function to each group independently and then combine the results.

**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])
```

**Use:** 
- apply multiple functions to multiple columns

**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)
```

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

### Applying filtering functions

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

**Use:** filtering functions allow you to make selections based on whether each group meets specified criteria
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

```python
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"
```

```python
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
long3 = pd.melt(wide1, value_vars=['Est_blue', 'Est_red'],
var_name='Party', id_vars=['Date', 'Pollster', 'State'])
```

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

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).

```python
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 should be in range 1678 and 2261 years. (Check Timestamp.max and Timestamp.min).

A Series of Timestamps or Periods

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

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

# Series of Periods (hard to make)
s = pd.Series(
    [pd.Period(x, freq='M') for x in ts])

Note: While Periods make a very useful index; they may be less useful in a Series.

From non-standard strings to Timestamps

```python
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.

```python
date_strs = ['2014-01-01', '2014-04-01',
            '2014-07-01', '2014-10-01']
```

```
dti = pd.DatetimeIndex(date_strs)
```

```python
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']
```

```
print (pd.to_datetime(time_strs, freq='U'))
```

Hint: unless you are working in less than seconds, prefer PeriodIndex over DateTimeImdex.
### Period frequency constants (not a complete list)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Microsecond</td>
</tr>
<tr>
<td>L</td>
<td>Millisecond</td>
</tr>
<tr>
<td>S</td>
<td>Second</td>
</tr>
<tr>
<td>T</td>
<td>Minute</td>
</tr>
<tr>
<td>H</td>
<td>Hour</td>
</tr>
<tr>
<td>D</td>
<td>Calendar day</td>
</tr>
<tr>
<td>B</td>
<td>Business day</td>
</tr>
<tr>
<td>W-(MON, TUE,...)</td>
<td>Week ending on...</td>
</tr>
<tr>
<td>MS</td>
<td>Calendar start of month</td>
</tr>
<tr>
<td>M</td>
<td>Calendar end of month</td>
</tr>
<tr>
<td>QS-(JAN, FEB,...)</td>
<td>Quarter start with year starting (QS – December)</td>
</tr>
<tr>
<td>Q-(JAN, FEB,...)</td>
<td>Quarter end with year ending (Q – December)</td>
</tr>
<tr>
<td>AS-(JAN, FEB,...)</td>
<td>Year start (AS - December)</td>
</tr>
<tr>
<td>A-(JAN, FEB,...)</td>
<td>Year end (A - December)</td>
</tr>
</tbody>
</table>

### From DatetimeIndex to Python datetime objects

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

### From Timestamps to Python dates or times

```python
df['date'] = [x.date() for x in df['TS']]
df['time'] = [x.time() for x in df['TS']]  # Note: converts to datetime.date or datetime.time. But does not convert to datetime.datetime.
```

### From DatetimeIndex to PeriodIndex and back

```python
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

```python
pi = pd.period_range('1960-01', '2015-12', freq='M')
na = pi.values # numpy array of integers
lp = pi.tolist() # python list of Periods
sp = Series(pi) # pandas Series of Periods
ss = Series(pi).astype(str) # S of strs
ls = Series(pi).astype(str).tolist()
```

### Get a range of Timestamps

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

### Error handling with dates

```python
# 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

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

### Upsampling and downsampling

```python
# 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

```python
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 pytz
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

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

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

Version 30 April 2017 - [Draft – Mark Graph – mark dot the dot graph at gmail dot com – @Mark_Graph on twitter]
Plotting from the DataFrame

Import matplotlib, choose a matplotlib style

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

Fake up some data (which we reuse repeatedly)

```python
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

```python
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

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

Bar plots

```python
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

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

Density plot

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

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

Pie chart

```python
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

```python
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

```python
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

```python
# 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
```

Recoding missing data
```
s.fillna(0, inplace=True) # np.nan -> 0
```

Non-finite numbers
With floating point numbers, pandas provides for positive and negative infinity.
```
s = Series([float('inf'), float('-inf'), np.inf, -np.inf])
```

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_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

# 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)

cross = 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.

Version: This cheat sheet was last updated with Python 3.6 and pandas 0.19.2 in mind.

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).