Cheat Sheet: The pandas DataFrame Object

Start by importing these Python modules

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

**Note**: these are the recommended import aliases

### Preliminaries

#### Get your data into a DataFrame

**Load a DataFrame from a CSV file**

```python
df = pd.read_csv('file.csv')  # often works
```

**Note**: refer to pandas docs for all arguments

**From inline CSV text to a DataFrame**

```python
from StringIO import StringIO
df = pd.read_csv(StringIO(data), skipinitialspace=True)
```

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

**Load DataFrames from a Microsoft Excel file**

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

e = create_engine('mysql+pymysql://USER:PASSWORD@localhost/DATABASE')
df = pd.read_sql_table('table', engine=e)
```

### The conceptual model

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

**Series object**: an ordered, one-dimensional array of data with an index. 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.

```python
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'
```

**The index object**: The pandas Index provides the axis labels for the Series and DataFrame objects. It can only contain hashable objects. A pandas Series has one Index; and a DataFrame has two Indexes.

```python
# --- get Index from Series and DataFrame
idx = s.index
idx = df.columns # the column index
idx = df.index   # the row index

# --- some Index attributes
b = idx.is_monotonic_decreasing
b = idx.is_monotonic_increasing
b = idx.has_duplicates
i = idx.nlevels   # multi-level indexes

# --- some Index methods
a = idx.values()  # get as numpy array
l = idx.tolist()  # get as a python list
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
```

### Data in Series then combine into a DataFrame

```python
# Example 1 ...
s1 = Series(range(6))
s2 = s1 + s2
s2.index = s2.index + 2
s3 = Series(["Tom", "Dick", "Har", "Mar", "Sam", "Tom"])  
s4 = Series({"Tom":1, "Dick":4, "Har":9})
s4 = Series({"Tom":3, "Dick":2, "Mar":5})
df = pd.concat([s1, s2], axis=1)
```

**Note**: 1st method has in integer column labels

**Note**: 2nd method does not guarantee col order

**Note**: index alignment on DataFrame creation

Get a DataFrame from data in a Python dictionary

```python
# default --- assume data is in columns
df = DataFrame({'col1' : [1.0, 2.0, 3.0, 4.0], 'col2' : [100, 200, 300, 400]})
```
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')
```

```python
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
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('1/1/2006', periods=len(df), freq='M')
```

```python
# --- with alphabetic row and col indexes
import random
r = 52 # note: min r is 1; max r is 52
c = 5
df = DataFrame(np.random.randn(r, c),
columns = ['col'+str(i) for i in range(c)]
index = list((string.uppercase + string.lowercase)[0:r])
df['group'] = list( 
'.join(random.choice('abcd')
for _ in range(r))
)
```

### Saving a DataFrame

#### Saving a DataFrame to a CSV file

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

#### Saving DataFrames to an Excel Workbook

```python
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

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

#### Saving a DataFrame to a Python dictionary

```python
dictionary = df.to_dict()
```

#### Saving a DataFrame to a Python string

```python
string = df.to_string()
```

### Working with the whole DataFrame

#### Peek at the DataFrame contents

```python
df.info() # index & data types
n = 4
dfh = df.head(n) # get first n rows
dft = df.tail(n) # get last n rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:5,:5]
```

#### Dataframe non-indexing attributes

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

#### Dataframe utility methods

```python
dfc = df.copy() # copy a DataFrame
dfr = df.rank() # rank each col (default)
dfs = df.sort() # sort each col (default)
dfc = df.astype(dtype) # type conversion
```

#### Dataframe iteration methods

```python
dfs = df.iteritems() # (col-index, Series) pairs
dfr = df.iterrows() # (row-index, Series) pairs
```

#### Maths on the whole DataFrame (not a complete list)

```python
df = df.abs() # absolute values
df = df.add(o) # add df, Series or value
df = df.add(o) # add df, Series or value
s = df.count() # non NA/null values
```

```python
df = df.corr() # matrix correlation product
s = df.corr() # matrix correlation product
```

#### Dataframe filter/select rows or cols on label info

```python
df = df.filter(items=['a', 'b']) # by col
df = df.filter(items=[5], axis=0) # by row
```

```python
df = df.filter(regex='x') # keep x in col
df = df.select(regex='x') # regex in col
```

**Note:** select takes a Boolean function, for cols: axis=1

**Note:** filter defaults to cols; select defaults to rows

---

**Version 2 May 2015 – [Draft – Mark Graph – mark dot the dot graph at gmail dot com – @Mark_Graph on twitter]**
Working with Columns

A DataFrame column is a pandas Series object

Get column index and labels

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

Change column labels

```python
df.rename(columns={'old':'new'}, inplace=True)
df = df.rename(columns={'a':1,'b':'x'})
```

Selecting columns

```python
s = df['colName']  # select col to Series
df = df[['col1', 'col2']]  # select 2 or more
s = df[df.columns[0]]  # select by number
s = df.pop('c')  # get col & drop from df
```

Selecting columns with Python attributes

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

Adding new columns to a DataFrame

```python
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan, len(df))
df = df['col1'].to_datetime()
df['new_col'] = df['col1'].round(decimals=0)
df['new_col'] = df['col1'].notnull()  # not isnull()
df['new_col'] = df['col1'].isnull()
df['new_col'] = df['col1'].sum()
df['new_col'] = df['col1'].mean()
df['new_col'] = df['col1'].max()
df['new_col'] = df['col1'].min()
df['new_col'] = df['col1'].median()
df['new_col'] = df['col1'].count()
df['new_col'] = df['col1'].cov(df['col2'])
df['new_col'] = df['col1'].describe()
df['new_col'] = df['col1'].value_counts()
```

Swap column contents

```python
df[['B', 'A']] = df[['A', 'B']]  # change column order
```

Dropping columns (mostly by label)

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

Vectorised arithmetic on columns

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

Apply numpy mathematical functions to columns

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

Test if column index values are unique/monotonic

```python
if df.columns.is_unique: pass
```

Data type conversions

```python
s = df['col'].astype(str)  # Series dtype
na = df['col'].values  # numpy array
pl = df['col'].tolist()  # python list
```

Common column-wide methods/attributes

```python
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'].mean()
value = df['col'].median()
value = df['col'].min()
value = df['col'].max()
value = df['col'].describe()
```

Common column element-wise methods

```python
s = df['col'].isnan()
s = df['col'].isnull()  # not isnan()
s = df['col'].notnull()
s = df['col'].astype(float)
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_sum(periods=4, window=4)
```

Find index label for min/max values in column

```python
label = df['col'].idxmin()
lable = df['col'].idxmax()
```

Append a column of row sums to a DataFrame

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

Multiply every column in a DataFrame by Series

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

Selecting columns with .loc, .iloc and .ix

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

Get the integer position of a column index label

```python
j = df.columns.get_loc('col_name')
```

Columns value set based on criteria

```python
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

Mark Graph on twitter

Mark_Graph on twitter
Working with rows

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
```

Adding rows

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

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

Dropping rows (by name)

```
df = df.drop('row_label')
```

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

Selecting rows using isin over multiple columns

```
data = {1:[1,2,3], 2:[1,4,9], 3:[1,8,27]}
df = pd.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.ix[idx])
```

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: 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
```

Sorting DataFrame rows values

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

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, :]
```

Note: this sample is not sorted

Sort DataFrame by its row index

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

Drop duplicates in the row index

```
df['index'] = df.index
df = df.drop_duplicates(cols='index', take_last=True)
del df['index']
df.sort_index(inplace=True)  # 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 or 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
### 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.loc['row', 'col'] = value`
- `df['col'].at['row'] = value`  # tricky

#### Selecting and slicing on labels

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

**Note:** The "to" on this slice is inclusive.

#### Selecting a cross-section by labels

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

```python
df = df.iloc[2:4, 2:4]  # subset of the df
```

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

#### Setting cell by integer position

```python
df.iloc[0, 0] = value
```

#### Setting cell range by integer position

```python
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.

### In summary: indexes and addresses

**In the main, these notes focus on the simple, single level Indexes. Pandas also has a hierarchical or multi-level Indexes (aka the MultiIndex).**

#### A DataFrame has two Indexes

- Typically, the column index (df.columns) is a list of strings (observed variable names) or (less commonly) integers (the default is numbered from 0 to length-1).
- Typically, the row index (df.index) might be:
  - Integers - for case or row numbers (default is numbered from 0 to length-1);
  - Strings – for case names; or
  - DatetimeIndex or PeriodIndex – for time series data (more below)

#### Indexing

- **# --- selecting columns**
  - `s = df['col_label']`  # scalar
  - `df = df[['col_label']]`  # one item list
  - `df = df[['L1', 'L2']]`  # many item list
  - `df = df.index`  # pandas Index
  - `df = df[s]`  # pandas Series

- **# --- selecting rows**
  - `df = df['from':'inc_to']`  # label slice
  - `df = df[3:7]`  # integer slice
  - `df = df[df['col'] > 0.5]`  # Boolean Series

- **# --- select DataFrame 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.index[r,c] # label access int fallback df[c].iloc[r] # chained also for .loc`

- **# --- select 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`

- **# --- 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.
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
\[
df\_new = \text{pd.merge(left=df1, right=df2, how='outer', left_index=True, right_index=True)}
\]

Merge on columns
\[
df\_new = \text{pd.merge(left=df1, right=df2, how='left', left_on='col1', right_on='col2')}
\]
Trap: When joining on columns, the indexes on the passed DataFrames are ignored.

Join on indexes (another way of merging)
\[
df\_new = \text{df1.join(other=df2, on='col1', how='outer')}
\]
\[
df\_new = \text{df1.join(other=df2, on=["a", "b"], how='outer')}
\]
Note: DataFrame.join() joins on indexes by default. DataFrame.merge() joins on common columns by default.

Simple concatenation is often the best
\[
df = \text{pd.concat([df1, df2], axis=0)}
\]
\[
df = \text{df1.append([df2, df3])}
\]
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()
\[
df = \text{reduce(lambda x, y: x.combine_first(y), [df1, df2, df3, df4, df5])}
\]
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

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

Grouping
\[
\begin{align*}
gb &= \text{df.groupby('cat') \# by one columns} \\
gb &= \text{df.groupby(["c1", 'c2']) \# by 2 cols} \\
gb &= \text{df.groupby(level=0) \# multi-index} \\
\end{align*}
\]
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.

Iterating groups – usually not needed
for name, group in gb:
print(name)
print(group)

Selecting a group
\[
dfa = \text{df.groupby('cat').get_group('a')}
\]
\[
dfb = \text{df.groupby('cat').get_group('b')}
\]

Applying an aggregating function
\[
\begin{align*}
s &= \text{df.groupby('cat')['col1'].sum()} \\
s &= \text{df.groupby('cat')['col1'].agg(np.sum)} \\
\end{align*}
\]
# apply to the every column in DataFrame
\[
s = \text{df.groupby('cat').agg(np.sum)} \\
df\_summary = \text{df.groupby('cat').describe()} \\
df\_row\_ls = \text{df.groupby('cat').head(1)}
\]
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
\[
\begin{align*}
gb &= \text{df.groupby('cat')} \\
gbx = gb["col2"].agg([np.sum, np.mean]) \\
gby = gb.agg({
'\text{cat}': np.count_nonzero, 
'\text{col1}': [np.sum, np.mean, np.std], 
'\text{col2}': [np.min, np.max]
})
\end{align*}
\]
Note: gb["col2"] above is shorthand for df.groupby('cat')['col2'], without the need for regrouping.

Transforming functions
\[
\begin{align*}
\text{# transform to group z-scores, which have a group mean of 0, and a std dev of 1.} \\
z\text{score} &= \lambda x: (x-x.\text{mean()})/x.\text{std()} \\
dfz &= gb.groupby('cat').\text{transform(zscore)}
\end{align*}
\]
# replace missing data with group mean
\[
\begin{align*}
\text{mean}\_r &= \lambda x: x.\text{fillna(x.\text{mean()})} \\
dfm &= gb.groupby('cat').\text{transform(mean}\_r)
\end{align*}
\]
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
Pivot
Pivot tables move from long format to wide format data

df = DataFrame(np.random.rand(100,1))
df.columns = ['data'] # rename col
df.index = pd.period_range('3/3/2014', periods=len(df), freq='M')
df['year'] = df.index.year
df['month'] = df.index.month
# pivot to wide format
df = df.pivot(index='year', columns='month', values='data')
# melt to long format
dfm = pd.melt(df, id_vars=['year'], var_name='month', value_name='data')
# unstack to long format
# reset index to remove multi-level index
dfu=df.unstack().reset_index(name='data')

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

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

A Series of Timestamps or Periods

ts = ['2015-04-01 13:17:27',
'2014-04-02 13:17:29']

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

# Series of Periods (often not so good)
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

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.

dti = pd.DatetimeIndex(date_strs)
pid = pd.PeriodIndex(dati_strs, freq='D')

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

Hint: unless you are working in less than seconds, prefer PeriodIndex over DatetimeIndex.
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</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</td>
<td>Quarter start with year starting (QS – December)</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter end with year ending (Q – December)</td>
</tr>
<tr>
<td>AS</td>
<td>Year start (AS – December)</td>
</tr>
<tr>
<td>A</td>
<td>Year end (A – December)</td>
</tr>
</tbody>
</table>

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)
na = dti.to_pydatetime()  # numpy array
na = s.to_pydatetime()    # numpy array

From Timestap to Python dates or times

def['date'] = [x.date() for x in df['TS']]
def['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

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')
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

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') |
|   | 2014-02-30                  |
|   | NaT (not a time)            |

# NaT like NaN tests True for isnnull()

| b | pd.isnull(t) | --> True |

The tail of a time-series DataFrame

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

Version 2 May 2015 - [Draft – Mark Graph – mark dot the dot graph at gmail dot com – @Mark_Graph on twitter]
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

# 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['col'].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['col1'],
                          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())

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)

Cautionary note

This cheat sheet was cobbled together by bots roaming the dark recesses of the Internet seeking ursine and pythonic myths. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. You have been warned.