ffn.core — ffn 0.3.0 documentation

```
from __future__ import print_function
from future.utils import listvalues, iteritems
import random
from . import utils
from .utils import fmtp, fmtn, fmtpn, get_freq_name
import numpy as np
import pandas as pd
from pandas.core.base import PandasObject
from tabulate import tabulate
import sklearn.manifold
import sklearn.cluster
import sklearn.covariance
from scipy.optimize import minimize
import scipy.stats
from scipy.stats import t
try:
    import prettyplotlib  # NOQA
except ImportError:
    pass

# avoid pyplot import failure in headless environment
import os
import matplotlib

if 'DISPLAY' not in os.environ:
    matplotlib.use('agg', warn=False)

from matplotlib import pyplot as plt  # noqa


[docs]class PerformanceStats(object):

    """
    PerformanceStats is a convenience class used for the performance
    evaluation of a price series. It contains various helper functions
    to help with plotting and contains a large amount of descriptive
    statistics.

    Args:
        * prices (Series): A price series.
        * rf (float): Risk-free rate used in various calculation. Should be
            expressed as a yearly (annualized) return

    Attributes:
        * name (str): Name, derived from price series name
        * return_table (DataFrame): A table of monthly returns with
```

```
            YTD figures as well.
        * lookback_returns (Series): Returns for different
            lookback periods (1m, 3m, 6m, ytd...)
        * stats (Series): A series that contains all the stats

    """

    def __init__(self, prices, rf=0.):
        super(PerformanceStats, self).__init__()
        self.prices = prices
        self.name = self.prices.name
        self._start = self.prices.index[0]
        self._end = self.prices.index[-1]

        self.rf = rf

        self._update(self.prices)

[docs]    def set_riskfree_rate(self, rf):

        """
        Set annual risk-free rate property and calculate properly annualized
        monthly and daily rates. Then performance stats are recalculated.
        Affects only this instance of the PerformanceStats.

        Args:
            * rf (float): Annual risk-free rate
        """
        self.rf = rf

        # Note, that we recalculate everything.
        self._update(self.prices)


    def _update(self, obj):
        # calc
        self._calculate(obj)

        # update derived structure
        # return table as dataframe for easier manipulation
        self.return_table = pd.DataFrame(self.return_table).T
        # name columns
        if len(self.return_table.columns) == 13:
            self.return_table.columns = ['Jan', 'Feb', 'Mar', 'Apr', 'May',
                                         'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
                                         'Nov', 'Dec', 'YTD']

        self.lookback_returns = pd.Series(
            [self.mtd, self.three_month, self.six_month, self.ytd,
             self.one_year, self.three_year, self.five_year,
```

```
            self.ten_year, self.cagr],
            ['mtd', '3m', '6m', 'ytd', '1y', '3y', '5y', '10y', 'incep'])
        self.lookback_returns.name = self.name

        st = self._stats()
        self.stats = pd.Series(
            [getattr(self, x[0]) for x in st if x[0] is not None],
            [x[0] for x in st if x[0] is not None]).drop_duplicates()

    def _calculate(self, obj):
        # default values
        self.daily_mean = np.nan
        self.daily_vol = np.nan
        self.daily_sharpe = np.nan
        self.daily_sortino = np.nan
        self.best_day = np.nan
        self.worst_day = np.nan
        self.total_return = np.nan
        self.cagr = np.nan
        self.incep = np.nan
        self.drawdown = np.nan
        self.max_drawdown = np.nan
        self.drawdown_details = np.nan
        self.daily_skew = np.nan
        self.daily_kurt = np.nan
        self.monthly_returns = np.nan
        self.avg_drawdown = np.nan
        self.avg_drawdown_days = np.nan
        self.monthly_mean = np.nan
        self.monthly_vol = np.nan
        self.monthly_sharpe = np.nan
        self.monthly_sortino = np.nan
        self.best_month = np.nan
        self.worst_month = np.nan
        self.mtd = np.nan
        self.three_month = np.nan
        self.pos_month_perc = np.nan
        self.avg_up_month = np.nan
        self.avg_down_month = np.nan
        self.monthly_skew = np.nan
        self.monthly_kurt = np.nan
        self.six_month = np.nan
        self.yearly_returns = np.nan
        self.ytd = np.nan
        self.one_year = np.nan
        self.yearly_mean = np.nan
        self.yearly_vol = np.nan
        self.yearly_sharpe = np.nan
        self.yearly_sortino = np.nan
        self.best_year = np.nan
```

```
self.worst_year = np.nan
self.three_year = np.nan
self.win_year_perc = np.nan
self.twelve_month_win_perc = np.nan
self.yearly_skew = np.nan
self.yearly_kurt = np.nan
self.five_year = np.nan
self.ten_year = np.nan
self.calmar = np.nan

self.return_table = {}
# end default values

if len(obj) is 0:
    return

self.start = obj.index[0]
self.end = obj.index[-1]

# save daily prices for future use
self.daily_prices = obj
# M = month end frequency
self.monthly_prices = obj.resample('M').last()
# A == year end frequency
self.yearly_prices = obj.resample('A').last()

# let's save some typing
p = obj
mp = self.monthly_prices
yp = self.yearly_prices

if len(p) is 1:
    return

# stats using daily data
self.returns = p.to_returns()
self.log_returns = p.to_log_returns()
r = self.returns

if len(r) < 2:
    return

self.daily_mean = r.mean() * 252
self.daily_vol = r.std() * np.sqrt(252)
self.daily_sharpe = r.calc_sharpe(rf=self.rf, nperiods=252)
self.daily_sortino = calc_sortino_ratio(r, rf=self.rf, nperiods=252)
self.best_day = r.max()
self.worst_day = r.min()

self.total_return = obj[-1] / obj[0] - 1
```

```
# save ytd as total_return for now - if we get to real ytd
# then it will get updated
self.ytd = self.total_return
self.cagr = calc_cagr(p)
self.incep = self.cagr

self.drawdown = p.to_drawdown_series()
self.max_drawdown = self.drawdown.min()
self.drawdown_details = drawdown_details(self.drawdown)
if self.drawdown_details is not None:
    self.avg_drawdown = self.drawdown_details['drawdown'].mean()
    self.avg_drawdown_days = self.drawdown_details['days'].mean()

self.calmar = self.cagr / abs(self.max_drawdown)

if len(r) < 4:
    return

self.daily_skew = r.skew()

# if all zero/nan kurt fails division by zero
if len(r[(~np.isnan(r)) & (r != 0)]) > 0:
    self.daily_kurt = r.kurt()

# stats using monthly data
self.monthly_returns = self.monthly_prices.to_returns()
mr = self.monthly_returns

if len(mr) < 2:
    return

self.monthly_mean = mr.mean() * 12
self.monthly_vol = mr.std() * np.sqrt(12)
self.monthly_sharpe = mr.calc_sharpe(rf=self.rf, nperiods=12)
self.monthly_sortino = calc_sortino_ratio(mr, rf=self.rf, nperiods=12)
self.best_month = mr.max()
self.worst_month = mr.min()

# -2 because p[-1] will be mp[-1]
self.mtd = p[-1] / mp[-2] - 1

# -1 here to account for first return that will be nan
self.pos_month_perc = len(mr[mr > 0]) / float(len(mr) - 1)
self.avg_up_month = mr[mr > 0].mean()
self.avg_down_month = mr[mr <= 0].mean()

# return_table
for idx in mr.index:
    if idx.year not in self.return_table:
        self.return_table[idx.year] = {1: 0, 2: 0, 3: 0,
```

```
                                    4: 0, 5: 0, 6: 0,
                                    7: 0, 8: 0, 9: 0,
                                    10: 0, 11: 0, 12: 0}
    if not np.isnan(mr[idx]):
        self.return_table[idx.year][idx.month] = mr[idx]
# add first month
fidx = mr.index[0]
try:
    self.return_table[fidx.year][fidx.month] = float(mp[0]) / p[0] - 1
except ZeroDivisionError:
    self.return_table[fidx.year][fidx.month] = 0
# calculate the YTD values
for idx in self.return_table:
    arr = np.array(listvalues(self.return_table[idx]))
    self.return_table[idx][13] = np.prod(arr + 1) - 1

if len(mr) < 3:
    return

denom = p[:p.index[-1] - pd.DateOffset(months=3)]
if len(denom) > 0:
    self.three_month = p[-1] / denom[-1] - 1

if len(mr) < 4:
    return

self.monthly_skew = mr.skew()

# if all zero/nan kurt fails division by zero
if len(mr[(~np.isnan(mr)) & (mr != 0)]) > 0:
    self.monthly_kurt = mr.kurt()

denom = p[:p.index[-1] - pd.DateOffset(months=6)]
if len(denom) > 0:
    self.six_month = p[-1] / denom[-1] - 1

self.yearly_returns = self.yearly_prices.to_returns()
yr = self.yearly_returns

if len(yr) < 2:
    return

self.ytd = p[-1] / yp[-2] - 1

denom = p[:p.index[-1] - pd.DateOffset(years=1)]
if len(denom) > 0:
    self.one_year = p[-1] / denom[-1] - 1

self.yearly_mean = yr.mean()
self.yearly_vol = yr.std()
```

```
    self.yearly_sortino = calc_sortino_ratio(yr, rf=self.rf, nperiods=1)
    if self.yearly_vol > 0:
        self.yearly_sharpe = yr.calc_sharpe(rf=self.rf, nperiods=1)

    self.best_year = yr.max()
    self.worst_year = yr.min()

    # annualize stat for over 1 year
    self.three_year = calc_cagr(p[p.index[-1] - pd.DateOffset(years=3):])

    # -1 here to account for first return that will be nan
    self.win_year_perc = len(yr[yr > 0]) / float(len(yr) - 1)

    # need at least 1 year of monthly returns
    if mr.size > 11:
        tot = 0
        win = 0
        for i in range(11, len(mr)):
            tot += 1
            if mp[i] / mp[i - 11] > 1:
                win += 1
        self.twelve_month_win_perc = float(win) / tot

    if len(yr) < 4:
        return

    self.yearly_skew = yr.skew()

    # if all zero/nan kurt fails division by zero
    if len(yr[(~np.isnan(yr)) & (yr != 0)]) > 0:
        self.yearly_kurt = yr.kurt()

    self.five_year = calc_cagr(p[p.index[-1] - pd.DateOffset(years=5):])
    self.ten_year = calc_cagr(p[p.index[-1] - pd.DateOffset(years=10):])

    return

def _stats(self):
    stats = [('start', 'Start', 'dt'),
             ('end', 'End', 'dt'),
             ('rf', 'Risk-free rate', 'p'),
             (None, None, None),
             ('total_return', 'Total Return', 'p'),
             ('daily_sharpe', 'Daily Sharpe', 'n'),
             ('daily_sortino', 'Daily Sortino', 'n'),
             ('cagr', 'CAGR', 'p'),
             ('max_drawdown', 'Max Drawdown', 'p'),
             ('calmar', 'Calmar Ratio', 'n'),
             (None, None, None),
             ('mtd', 'MTD', 'p'),
```

```
                ('three_month', '3m', 'p'),
                ('six_month', '6m', 'p'),
                ('ytd', 'YTD', 'p'),
                ('one_year', '1Y', 'p'),
                ('three_year', '3Y (ann.)', 'p'),
                ('five_year', '5Y (ann.)', 'p'),
                ('ten_year', '10Y (ann.)', 'p'),
                ('incep', 'Since Incep. (ann.)', 'p'),
                (None, None, None),
                ('daily_sharpe', 'Daily Sharpe', 'n'),
                ('daily_sortino', 'Daily Sortino', 'n'),
                ('daily_mean', 'Daily Mean (ann.)', 'p'),
                ('daily_vol', 'Daily Vol (ann.)', 'p'),
                ('daily_skew', 'Daily Skew', 'n'),
                ('daily_kurt', 'Daily Kurt', 'n'),
                ('best_day', 'Best Day', 'p'),
                ('worst_day', 'Worst Day', 'p'),
                (None, None, None),
                ('monthly_sharpe', 'Monthly Sharpe', 'n'),
                ('monthly_sortino', 'Monthly Sortino', 'n'),
                ('monthly_mean', 'Monthly Mean (ann.)', 'p'),
                ('monthly_vol', 'Monthly Vol (ann.)', 'p'),
                ('monthly_skew', 'Monthly Skew', 'n'),
                ('monthly_kurt', 'Monthly Kurt', 'n'),
                ('best_month', 'Best Month', 'p'),
                ('worst_month', 'Worst Month', 'p'),
                (None, None, None),
                ('yearly_sharpe', 'Yearly Sharpe', 'n'),
                ('yearly_sharpe', 'Yearly Sortino', 'n'),
                ('yearly_mean', 'Yearly Mean', 'p'),
                ('yearly_vol', 'Yearly Vol', 'p'),
                ('yearly_skew', 'Yearly Skew', 'n'),
                ('yearly_kurt', 'Yearly Kurt', 'n'),
                ('best_year', 'Best Year', 'p'),
                ('worst_year', 'Worst Year', 'p'),
                (None, None, None),
                ('avg_drawdown', 'Avg. Drawdown', 'p'),
                ('avg_drawdown_days', 'Avg. Drawdown Days', 'n'),
                ('avg_up_month', 'Avg. Up Month', 'p'),
                ('avg_down_month', 'Avg. Down Month', 'p'),
                ('win_year_perc', 'Win Year %', 'p'),
                ('twelve_month_win_perc', 'Win 12m %', 'p')]

        return stats

[docs]    def set_date_range(self, start=None, end=None):
        """
        Update date range of stats, charts, etc. If None then
        the original date is used. So to reset to the original
        range, just call with no args.
```

```
        Args:
            * start (date): start date
            * end (end): end date

        """
        if start is None:
            start = self._start
        else:
            start = pd.to_datetime(start)

        if end is None:
            end = self._end
        else:
            end = pd.to_datetime(end)

        self._update(self.prices.ix[start:end])


[docs]   def display(self):
        """
        Displays an overview containing descriptive stats for the Series
        provided.
        """
        print('Stats for %s from %s - %s' % (self.name, self.start, self.end))
        print('Annual risk-free rate considered: %s' % (fmtp(self.rf)))
        print('Summary:')
        data = [[fmtp(self.total_return), fmtn(self.daily_sharpe),
                 fmtp(self.cagr), fmtp(self.max_drawdown)]]
        print(tabulate(data, headers=['Total Return', 'Sharpe',
                                      'CAGR', 'Max Drawdown']))

        print('\nAnnualized Returns:')
        data = [[fmtp(self.mtd), fmtp(self.three_month), fmtp(self.six_month),
                 fmtp(self.ytd), fmtp(self.one_year), fmtp(self.three_year),
                 fmtp(self.five_year), fmtp(self.ten_year),
                 fmtp(self.incep)]]
        print(tabulate(data,
                       headers=['mtd', '3m', '6m', 'ytd', '1y',
                                '3y', '5y', '10y', 'incep.']))

        print('\nPeriodic:')
        data = [
            ['sharpe', fmtn(self.daily_sharpe), fmtn(self.monthly_sharpe),
             fmtn(self.yearly_sharpe)],
            ['mean', fmtp(self.daily_mean), fmtp(self.monthly_mean),
             fmtp(self.yearly_mean)],
            ['vol', fmtp(self.daily_vol), fmtp(self.monthly_vol),
             fmtp(self.yearly_vol)],
            ['skew', fmtn(self.daily_skew), fmtn(self.monthly_skew),
```

```
             fmtn(self.yearly_skew)],
            ['kurt', fmtn(self.daily_kurt), fmtn(self.monthly_kurt),
             fmtn(self.yearly_kurt)],
            ['best', fmtp(self.best_day), fmtp(self.best_month),
             fmtp(self.best_year)],
            ['worst', fmtp(self.worst_day), fmtp(self.worst_month),
             fmtp(self.worst_year)]]
        print(tabulate(data, headers=['daily', 'monthly', 'yearly']))

        print('\nDrawdowns:')
        data = [
            [fmtp(self.max_drawdown), fmtp(self.avg_drawdown),
             fmtn(self.avg_drawdown_days)]]
        print(tabulate(data, headers=['max', 'avg', '# days']))

        print('\nMisc:')
        data = [['avg. up month', fmtp(self.avg_up_month)],
                ['avg. down month', fmtp(self.avg_down_month)],
                ['up year %', fmtp(self.win_year_perc)],
                ['12m up %', fmtp(self.twelve_month_win_perc)]]
        print(tabulate(data))


[docs]    def display_monthly_returns(self):
        """
        Display a table containing monthly returns and ytd returns
        for every year in range.
        """
        data = [['Year', 'Jan', 'Feb', 'Mar', 'Apr', 'May',
                 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'YTD']]
        for k in self.return_table.index:
            r = self.return_table.ix[k].values
            data.append([k] + [fmtpn(x) for x in r])
        print(tabulate(data, headers='firstrow'))


[docs]    def display_lookback_returns(self):
        """
        Displays the current lookback returns.
        """
        return self.lookback_returns.map('{:,.2%}'.format)


    def _get_default_plot_title(self, name, freq, kind):
        if freq is None:
            return '%s %s' % (name, kind)
        else:
            return '%s %s %s' % (name, get_freq_name(freq), kind)

[docs]    def plot(self, freq=None, figsize=(15, 5), title=None,
```

```
             logy=False, **kwargs):
        """
        Helper function for plotting the series.

        Args:
            * freq (str): Data frequency used for display purposes.
                Refer to pandas docs for valid freq strings.
            * figsize ((x,y)): figure size
            * title (str): Title if default not appropriate
            * logy (bool): log-scale for y axis
            * kwargs: passed to pandas' plot method
        """
        if title is None:
            title = self._get_default_plot_title(
                self.name, freq, 'Price Series')

        ser = self._get_series(freq)
        ser.plot(figsize=figsize, title=title, logy=logy, **kwargs)


[docs]   def plot_histogram(self, freq=None, figsize=(15, 5), title=None,
                    bins=20, **kwargs):
        """
        Plots a histogram of returns given a return frequency.

        Args:
            * freq (str): Data frequency used for display purposes.
                This will dictate the type of returns
                (daily returns, monthly, ...)
                Refer to pandas docs for valid period strings.
            * figsize ((x,y)): figure size
            * title (str): Title if default not appropriate
            * bins (int): number of bins for the histogram
            * kwargs: passed to pandas' hist method
        """
        if title is None:
            title = self._get_default_plot_title(
                self.name, freq, 'Return Histogram')

        ser = self._get_series(freq).to_returns().dropna()

        plt.figure(figsize=figsize)
        ax = ser.hist(bins=bins, figsize=figsize, normed=True, **kwargs)
        ax.set_title(title)
        plt.axvline(0, linewidth=4)
        ser.plot(kind='kde')


    def _get_series(self, freq):
        if freq is None:
```

```
            return self.daily_prices

        if freq == 'y':
            freq = 'a'
        return self.daily_prices.asfreq(freq, 'ffill')

[docs]    def to_csv(self, sep=',', path=None):
        """
        Returns a CSV string with appropriate formatting.
        If path is not None, the string will be saved to file
        at path.

        Args:
            * sep (char): Separator
            * path (str): If None, CSV string returned. Else file written
                to specified path.
        """
        stats = self._stats()

        data = []
        first_row = ['Stat', self.name]
        data.append(sep.join(first_row))

        for stat in stats:
            k, n, f = stat

            # blank row
            if k is None:
                row = [''] * len(data[0])
                data.append(sep.join(row))
                continue

            row = [n]
            raw = getattr(self, k)
            if f is None:
                row.append(raw)
            elif f == 'p':
                row.append(fmtp(raw))
            elif f == 'n':
                row.append(fmtn(raw))
            elif f == 'dt':
                row.append(raw.strftime('%Y-%m-%d'))
            else:
                raise NotImplementedError('unsupported format %s' % f)

            data.append(sep.join(row))

        res = '\n'.join(data)

        if path is not None:
```

```
            with open(path, 'w') as fl:
                fl.write(res)
        else:
            return res



[docs]class GroupStats(dict):

    """
    GroupStats enables one to compare multiple series side by side.
    It is a wrapper around a dict of {price.name: PerformanceStats} and
    provides many convenience methods.

    The order of the series passed in will be preserved.
    Individual PerformanceStats objects can be accessed via index
    position or name via the [] accessor.

    Args:
        * prices (Series): Multiple price series to be compared.

    Attributes:
        * stats (DataFrame): Dataframe containing stats for each
            series provided.  Stats in rows, series in columns.
        * lookback_returns (DataFrame): Returns for diffrent
            lookback periods (1m, 3m, 6m, ytd...)
            Period in rows, series in columns.
        * prices (DataFrame): The merged and rebased prices.

    """

    def __init__(self, *prices):
        names = []
        for p in prices:
            if isinstance(p, pd.DataFrame):
                names.extend(p.columns)
            elif isinstance(p, pd.Series):
                names.append(p.name)
            else:
                print('else')
                names.append(getattr(p, 'name', 'n/a'))
        self._names = names

        # store original prices
        self._prices = merge(*prices).dropna()

        # proper ordering
        self._prices = self._prices[self._names]

        # check for duplicate columns
```

```
    if len(self._prices.columns) != len(set(self._prices.columns)):
        raise ValueError('One or more data series provided',
                         'have same name! Please provide unique names')

    self._start = self._prices.index[0]
    self._end = self._prices.index[-1]
    # calculate stats for entire series
    self._update(self._prices)

def __getitem__(self, key):
    if type(key) == int:
        return self[self._names[key]]
    else:
        return self.get(key)

def _update(self, data):
    self._calculate(data)
    # lookback returns dataframe
    self.lookback_returns = pd.DataFrame(
        {x.lookback_returns.name: x.lookback_returns for x in
         self.values()})

    self.stats = pd.DataFrame(
        {x.name: x.stats for x in self.values()})

def _calculate(self, data):
    self.prices = data
    for c in data.columns:
        prc = data[c]
        self[c] = PerformanceStats(prc)

def _stats(self):
    stats = [('start', 'Start', 'dt'),
             ('end', 'End', 'dt'),
             ('rf', 'Risk-free rate', 'p'),
             (None, None, None),
             ('total_return', 'Total Return', 'p'),
             ('daily_sharpe', 'Daily Sharpe', 'n'),
             ('daily_sortino', 'Daily Sortino', 'n'),
             ('cagr', 'CAGR', 'p'),
             ('max_drawdown', 'Max Drawdown', 'p'),
             ('calmar', 'Calmar Ratio', 'n'),
             (None, None, None),
             ('mtd', 'MTD', 'p'),
             ('three_month', '3m', 'p'),
             ('six_month', '6m', 'p'),
             ('ytd', 'YTD', 'p'),
             ('one_year', '1Y', 'p'),
             ('three_year', '3Y (ann.)', 'p'),
             ('five_year', '5Y (ann.)', 'p'),
```

```
                ('ten_year', '10Y (ann.)', 'p'),
                ('incep', 'Since Incep. (ann.)', 'p'),
                (None, None, None),
                ('daily_sharpe', 'Daily Sharpe', 'n'),
                ('daily_sortino', 'Daily Sortino', 'n'),
                ('daily_mean', 'Daily Mean (ann.)', 'p'),
                ('daily_vol', 'Daily Vol (ann.)', 'p'),
                ('daily_skew', 'Daily Skew', 'n'),
                ('daily_kurt', 'Daily Kurt', 'n'),
                ('best_day', 'Best Day', 'p'),
                ('worst_day', 'Worst Day', 'p'),
                (None, None, None),
                ('monthly_sharpe', 'Monthly Sharpe', 'n'),
                ('monthly_sortino', 'Monthly Sortino', 'n'),
                ('monthly_mean', 'Monthly Mean (ann.)', 'p'),
                ('monthly_vol', 'Monthly Vol (ann.)', 'p'),
                ('monthly_skew', 'Monthly Skew', 'n'),
                ('monthly_kurt', 'Monthly Kurt', 'n'),
                ('best_month', 'Best Month', 'p'),
                ('worst_month', 'Worst Month', 'p'),
                (None, None, None),
                ('yearly_sharpe', 'Yearly Sharpe', 'n'),
                ('yearly_sortino', 'Yearly Sortino', 'n'),
                ('yearly_mean', 'Yearly Mean', 'p'),
                ('yearly_vol', 'Yearly Vol', 'p'),
                ('yearly_skew', 'Yearly Skew', 'n'),
                ('yearly_kurt', 'Yearly Kurt', 'n'),
                ('best_year', 'Best Year', 'p'),
                ('worst_year', 'Worst Year', 'p'),
                (None, None, None),
                ('avg_drawdown', 'Avg. Drawdown', 'p'),
                ('avg_drawdown_days', 'Avg. Drawdown Days', 'n'),
                ('avg_up_month', 'Avg. Up Month', 'p'),
                ('avg_down_month', 'Avg. Down Month', 'p'),
                ('win_year_perc', 'Win Year %', 'p'),
                ('twelve_month_win_perc', 'Win 12m %', 'p')]

        return stats

    def _get_default_plot_title(self, freq, kind):
        if freq is None:
            return '%s' % kind
        else:
            return '%s %s' % (get_freq_name(freq), kind)

[docs]    def set_riskfree_rate(self, rf):

        """
        Set annual risk-free rate property and calculate properly annualized
        monthly and daily rates. Then performance stats are recalculated.
```

```
        Affects only those instances of PerformanceStats that are children of
        this GroupStats object.

        Args:
            * rf (float): Annual risk-free rate
        """
        for key in self._names:
            self[key].set_riskfree_rate(rf)


[docs]   def set_date_range(self, start=None, end=None):
        """
        Update date range of stats, charts, etc. If None then
        the original date range is used. So to reset to the original
        range, just call with no args.

        Args:
            * start (date): start date
            * end (end): end date
        """
        if start is None:
            start = self._start
        else:
            start = pd.to_datetime(start)

        if end is None:
            end = self._end
        else:
            end = pd.to_datetime(end)

        self._update(self._prices.ix[start:end])


[docs]   def display(self):
        """
        Display summary stats table.
        """
        data = []
        first_row = ['Stat']
        first_row.extend(self._names)
        data.append(first_row)

        stats = self._stats()

        for stat in stats:
            k, n, f = stat
            # blank row
            if k is None:
                row = [''] * len(data[0])
                data.append(row)
```

```
                continue

            row = [n]
            for key in self._names:
                raw = getattr(self[key], k)
                if f is None:
                    row.append(raw)
                elif f == 'p':
                    row.append(fmtp(raw))
                elif f == 'n':
                    row.append(fmtn(raw))
                elif f == 'dt':
                    row.append(raw.strftime('%Y-%m-%d'))
                else:
                    raise NotImplementedError('unsupported format %s' % f)
            data.append(row)

        print(tabulate(data, headers='firstrow'))


[docs]    def display_lookback_returns(self):
        """
        Displays the current lookback returns for each series.
        """
        return self.lookback_returns.apply(
            lambda x: x.map('{:,.2%}'.format), axis=1)


[docs]    def plot(self, freq=None, figsize=(15, 5), title=None,
             logy=False, **kwargs):
        """
        Helper function for plotting the series.

        Args:
            * freq (str): Data frequency used for display purposes.
                Refer to pandas docs for valid freq strings.
            * figsize ((x,y)): figure size
            * title (str): Title if default not appropriate
            * logy (bool): log-scale for y axis
            * kwargs: passed to pandas' plot method

        """

        if title is None:
            title = self._get_default_plot_title(
                freq, 'Equity Progression')

        ser = self._get_series(freq).rebase()
        ser.plot(figsize=figsize, logy=logy,
                 title=title, **kwargs)
```

```
[docs]    def plot_scatter_matrix(self, freq=None, title=None,
                          figsize=(10, 10), **kwargs):
        """
        Wrapper around pandas' scatter_matrix.

        Args:
            * freq (str): Data frequency used for display purposes.
                Refer to pandas docs for valid freq strings.
            * figsize ((x,y)): figure size
            * title (str): Title if default not appropriate
            * kwargs: passed to pandas' scatter_matrix method

        """
        if title is None:
            title = self._get_default_plot_title(
                freq, 'Return Scatter Matrix')

        plt.figure()
        ser = self._get_series(freq).to_returns().dropna()
        pd.scatter_matrix(ser, figsize=figsize, **kwargs)
        plt.suptitle(title)


[docs]    def plot_histograms(self, freq=None, title=None,
                       figsize=(10, 10), **kwargs):
        """
        Wrapper around pandas' hist.

        Args:
            * freq (str): Data frequency used for display purposes.
                Refer to pandas docs for valid freq strings.
            * figsize ((x,y)): figure size
            * title (str): Title if default not appropriate
            * kwargs: passed to pandas' hist method

        """
        if title is None:
            title = self._get_default_plot_title(
                freq, 'Return Histogram Matrix')

        plt.figure()
        ser = self._get_series(freq).to_returns().dropna()
        ser.hist(figsize=figsize, **kwargs)
        plt.suptitle(title)


[docs]    def plot_correlation(self, freq=None, title=None,
                        figsize=(12, 6), **kwargs):
```

```
        """
        Utility function to plot correlations.

        Args:
            * freq (str): Pandas data frequency alias string
            * title (str): Plot title
            * figsize (tuple (x,y)): figure size
            * kwargs: passed to Pandas' plot_corr_heatmap function

        """
        if title is None:
            title = self._get_default_plot_title(
                freq, 'Return Correlation Matrix')

        rets = self._get_series(freq).to_returns().dropna()
        return rets.plot_corr_heatmap(title=title, figsize=figsize, **kwargs)


    def _get_series(self, freq):
        if freq is None:
            return self.prices

        if freq == 'y':
            freq = 'a'
        return self.prices.asfreq(freq, 'ffill')

[docs]    def to_csv(self, sep=',', path=None):
        """
        Returns a CSV string with appropriate formatting.
        If path is not None, the string will be saved to file
        at path.

        Args:
            * sep (char): Separator
            * path (str): If None, CSV string returned. Else file
                written to specified path.

        """
        data = []
        first_row = ['Stat']
        first_row.extend(self._names)
        data.append(sep.join(first_row))

        stats = self._stats()

        for stat in stats:
            k, n, f = stat
            # blank row
            if k is None:
                row = [''] * len(data[0])
```

```
                data.append(sep.join(row))
                continue

            row = [n]
            for key in self._names:
                raw = getattr(self[key], k)
                if f is None:
                    row.append(raw)
                elif f == 'p':
                    row.append(fmtp(raw))
                elif f == 'n':
                    row.append(fmtn(raw))
                elif f == 'dt':
                    row.append(raw.strftime('%Y-%m-%d'))
                else:
                    raise NotImplementedError('unsupported format %s' % f)
            data.append(sep.join(row))

        res = '\n'.join(data)

        if path is not None:
            with open(path, 'w') as fl:
                fl.write(res)
        else:
            return res



[docs]def to_returns(prices):
    """
    Calculates the simple arithmetic returns of a price series.

    Formula is: (t1 / t0) - 1

    Args:
        * prices: Expects a price series

    """
    return prices / prices.shift(1) - 1



[docs]def to_log_returns(prices):
    """
    Calculates the log returns of a price series.

    Formula is: ln(p1/p0)

    Args:
        * prices: Expects a price series
```

```
    """
    return np.log(prices / prices.shift(1))



[docs]def to_price_index(returns, start=100):
    """
    Returns a price index given a series of returns.

    Args:
        * returns: Expects a return series
        * start (number): Starting level

    Assumes arithmetic returns.

    Formula is: cumprod (1+r)
    """
    return (returns.replace(to_replace=np.nan, value=0) + 1).cumprod() * start



[docs]def rebase(prices, value=100):
    """
    Rebase all series to a given intial value.

    This makes comparing/plotting different series
    together easier.

    Args:
        * prices: Expects a price series
        * value (number): starting value for all series.

    """
    return prices / prices.ix[0] * value



[docs]def calc_perf_stats(prices):
    """
    Calculates the performance statistics given an object.
    The object should be a Series of prices.

    A PerformanceStats object will be returned containing all the stats.

    Args:
        * prices (Series): Series of prices

    """
    return PerformanceStats(prices)
```

```
[docs]def calc_stats(prices):
    """
    Calculates performance stats of a given object.

    If object is Series, a PerformanceStats object is
    returned. If object is DataFrame, a GroupStats object
    is returned.

    Args:
        * prices (Series, DataFrame): Set of prices
    """
    if isinstance(prices, pd.Series):
        return PerformanceStats(prices)
    elif isinstance(prices, pd.DataFrame):
        return GroupStats(*[prices[x] for x in prices.columns])
    else:
        raise NotImplementedError('Unsupported type')



[docs]def to_drawdown_series(prices):
    """
    Calculates the drawdown series.

    This returns a series representing a drawdown.
    When the price is at all time highs, the drawdown
    is 0. However, when prices are below high water marks,
    the drawdown series = current / hwm - 1

    The max drawdown can be obtained by simply calling .min()
    on the result (since the drawdown series is negative)

    Method ignores all gaps of NaN's in the price series.

    Args:
        * prices (Series or DataFrame): Series of prices.

    """
    # make a copy so that we don't modify original data
    drawdown = prices.copy()

    # Fill NaN's with previous values
    drawdown = drawdown.fillna(method='ffill')

    # Ignore problems with NaN's in the beginning
    drawdown[np.isnan(drawdown)] = -np.Inf
```

```
    # Rolling maximum
    roll_max = np.maximum.accumulate(drawdown)
    drawdown = drawdown / roll_max - 1.
    return drawdown



[docs]def calc_max_drawdown(prices):
    """
    Calculates the max drawdown of a price series. If you want the
    actual drawdown series, please use to_drawdown_series.
    """
    return (prices / prices.expanding(min_periods=1).max()).min() - 1



[docs]def drawdown_details(drawdown):
    """
    Returns a data frame with start, end, days (duration) and
    drawdown for each drawdown in a drawdown series.

    .. note::

        days are actual calendar days, not trading days

    Args:
        * drawdown (pandas.Series): A drawdown Series
            (can be obtained w/ drawdown(prices).
    Returns:
        * pandas.DataFrame -- A data frame with the following
            columns: start, end, days, drawdown.

    """
    is_zero = drawdown == 0
    # find start dates (first day where dd is non-zero after a zero)
    start = ~is_zero & is_zero.shift(1)
    start = list(start[start == True].index)  # NOQA

    # find end dates (first day where dd is 0 after non-zero)
    end = is_zero & (~is_zero).shift(1)
    end = list(end[end == True].index)  # NOQA

    if len(start) is 0:
        return None

    # drawdown has no end (end period in dd)
    if len(end) is 0:
        end.append(drawdown.index[-1])

    # if the first drawdown start is larger than the first drawdown end it
```

```
    # means the drawdown series begins in a drawdown and therefore we must add
    # the first index to the start series
    if start[0] > end[0]:
        start.insert(0, drawdown.index[0])

    # if the last start is greater than the end then we must add the last index
    # to the end series since the drawdown series must finish with a drawdown
    if start[-1] > end[-1]:
        end.append(drawdown.index[-1])

    result = pd.DataFrame(columns=('start', 'end', 'days', 'drawdown'),
                          index=range(0, len(start)))

    for i in range(0, len(start)):
        dd = drawdown[start[i]:end[i]].min()
        result.ix[i] = (start[i], end[i], (end[i] - start[i]).days, dd)

    return result



[docs]def calc_cagr(prices):
    """
    Calculates the CAGR (compound annual growth rate) for a given price series.

    Args:
        * prices (pandas.Series): A Series of prices.
    Returns:
        * float -- cagr.

    """
    start = prices.index[0]
    end = prices.index[-1]
    return (prices.ix[-1] / prices.ix[0]) ** (1 / year_frac(start, end)) - 1



[docs]def calc_risk_return_ratio(returns):
    """
    Calculates the return / risk ratio. Basically the
    Sharpe ratio without factoring in the risk-free rate.
    """
    return calc_sharpe(returns)



[docs]def calc_sharpe(returns, rf=0., nperiods=None, annualize=True):
    """
    Calculates the Sharpe ratio.
```

```
    If rf is non-zero, you must specify nperiods. In this case, rf is assumed
    to be expressed in yearly (annualized) terms.

    Args:
        * returns (Series, DataFrame): Input return series
        * rf (float): Risk-free rate expressed as a yearly (annualized) return
        * nperiods (int): Frequency of returns (252 for daily, 12 for monthly,
            etc.)

    """
    if rf != 0 and nperiods is None:
        raise Exception('Must provide nperiods if rf != 0')

    er = returns.to_excess_returns(rf, nperiods=nperiods)
    res = er.mean() / er.std()

    if annualize:
        if nperiods is None:
            nperiods = 1
        return res * np.sqrt(nperiods)
    else:
        return res



[docs]def calc_information_ratio(returns, benchmark_returns):
    """
    http://en.wikipedia.org/wiki/Information_ratio
    """
    diff_rets = returns - benchmark_returns
    diff_std = diff_rets.std()

    if np.isnan(diff_std) or diff_std == 0:
        return 0.0

    return diff_rets.mean() / diff_std



[docs]def calc_prob_mom(returns, other_returns):
    """
    Probabilistic momentum

    Basically the "probability or confidence that one asset
    is going to outperform the other".

    Source:
        http://cssanalytics.wordpress.com/2014/01/28/are-simple-momentum-strategies-too-dumb-introducing-
probabilistic-momentum/ # NOQA
    """
```

```
    return t.cdf(returns.calc_information_ratio(other_returns),
                 len(returns) - 1)



[docs]def calc_total_return(prices):
    """
    Calculates the total return of a series.

    last / first - 1
    """
    return (prices.ix[-1] / prices.ix[0]) - 1



[docs]def year_frac(start, end):
    """
    Similar to excel's yearfrac function. Returns
    a year fraction between two dates (i.e. 1.53 years).

    Approximation using the average number of seconds
    in a year.

    Args:
        * start (datetime): start date
        * end (datetime): end date

    """
    if start > end:
        raise ValueError('start cannot be larger than end')

    # obviously not perfect but good enough
    return (end - start).total_seconds() / (31557600)



[docs]def merge(*series):
    """
    Merge Series and/or DataFrames together.

    Returns a DataFrame.
    """
    dfs = []
    for s in series:
        if isinstance(s, pd.DataFrame):
            dfs.append(s)
        elif isinstance(s, pd.Series):
            tmpdf = pd.DataFrame({s.name: s})
            dfs.append(tmpdf)
        else:
```

```
            raise NotImplementedError('Unsupported merge type')

    return pd.concat(dfs, axis=1)



[docs]def drop_duplicate_cols(df):
    """
    Removes duplicate columns from a dataframe
    and keeps column w/ longest history
    """
    names = set(df.columns)
    for n in names:
        if len(df[n].shape) > 1:
            # get subset of df w/ colname n
            sub = df[n]
            # make unique colnames
            sub.columns = ['%s-%s' % (n, x) for x in range(sub.shape[1])]
            # get colname w/ max # of data
            keep = sub.count().idxmax()
            # drop all columns of name n from original df
            del df[n]
            # update original df w/ longest col with name n
            df[n] = sub[keep]

    return df



[docs]def to_monthly(series, method='ffill', how='end'):
    """
    Convenience method that wraps asfreq_actual
    with 'M' param (method='ffill', how='end').
    """
    return series.asfreq_actual('M', method=method, how=how)



[docs]def asfreq_actual(series, freq, method='ffill', how='end', normalize=False):
    """
    Similar to pandas' asfreq but keeps the actual dates.
    For example, if last data point in Jan is on the 29th,
    that date will be used instead of the 31st.
    """
    orig = series
    is_series = False
    if isinstance(series, pd.Series):
        is_series = True
        name = series.name if series.name else 'data'
        orig = pd.DataFrame({name: series})
```

```
    # add date column
    t = pd.concat([orig, pd.DataFrame({'dt': orig.index.values},
                                      index=orig.index.values)], axis=1)
    # fetch dates
    dts = t.asfreq(freq=freq, method=method, how=how,
                   normalize=normalize)['dt']

    res = orig.ix[dts.values]

    if is_series:
        return res[name]
    else:
        return res



[docs]def calc_inv_vol_weights(returns):
    """
    Calculates weights proportional to inverse volatility of each column.

    Returns weights that are inversely proportional to the column's
    volatility resulting in a set of portfolio weights where each position
    has the same level of volatility.

    Note, that assets with returns all equal to NaN or 0 are excluded from
    the portfolio (their weight is set to NaN).

    Returns:
        Series {col_name: weight}
    """
    # calc vols
    vol = 1.0 / returns.std()
    vol[np.isinf(vol)] = np.NaN
    vols = vol.sum()
    return vol / vols



[docs]def calc_mean_var_weights(returns, weight_bounds=(0., 1.),
                           rf=0.,
                           covar_method='ledoit-wolf'):
    """
    Calculates the mean-variance weights given a DataFrame of returns.

    Args:
        * returns (DataFrame): Returns for multiple securities.
        * weight_bounds ((low, high)): Weigh limits for optimization.
        * rf (float): Risk-free rate used in utility calculation
        * covar_method (str): Covariance matrix estimation method.
```

```
            Currently supported:
                - ledoit-wolf
                - standard

    Returns:
        Series {col_name: weight}

    """
    def fitness(weights, exp_rets, covar, rf):
        # portfolio mean
        mean = sum(exp_rets * weights)
        # portfolio var
        var = np.dot(np.dot(weights, covar), weights)
        # utility - i.e. sharpe ratio
        util = (mean - rf) / np.sqrt(var)
        # negative because we want to maximize and optimizer
        # minimizes metric
        return -util

    n = len(returns.columns)

    # expected return defaults to mean return by default
    exp_rets = returns.mean()

    # calc covariance matrix
    if covar_method == 'ledoit-wolf':
        covar = sklearn.covariance.ledoit_wolf(returns)[0]
    elif covar_method == 'standard':
        covar = returns.cov()
    else:
        raise NotImplementedError('covar_method not implemented')

    weights = np.ones([n]) / n
    bounds = [weight_bounds for i in range(n)]
    # sum of weights must be equal to 1
    constraints = ({'type': 'eq', 'fun': lambda W: sum(W) - 1.})
    optimized = minimize(fitness, weights, (exp_rets, covar, rf),
                         method='SLSQP', constraints=constraints,
                         bounds=bounds)
    # check if success
    if not optimized.success:
        raise Exception(optimized.message)

    # return weight vector
    return pd.Series({returns.columns[i]: optimized.x[i] for i in range(n)})



def _erc_weights_ccd(x0,
                     cov,
```

```
                b,
                maximum_iterations,
                tolerance):
"""
Calculates the equal risk contribution / risk parity weights given
a DataFrame of returns.

Args:
    * x0 (np.array): Starting asset weights.
    * cov (np.array): covariance matrix.
    * b (np.array): Risk target weights.
    * maximum_iterations (int): Maximum iterations in iterative solutions.
    * tolerance (float): Tolerance level in iterative solutions.

Returns:
    np.array {weight}

Reference:
    Griveau-Billion, Theophile and Richard, Jean-Charles and Roncalli,
    Thierry, A Fast Algorithm for Computing High-Dimensional Risk Parity
    Portfolios (2013).
    Available at SSRN: https://ssrn.com/abstract=2325255

"""
n = len(x0)
x = x0.copy()
var = np.diagonal(cov)
ctr = cov.dot(x)
sigma_x = np.sqrt(x.T.dot(ctr))

for iteration in range(maximum_iterations):

    for i in range(n):
        alpha = var[i]
        beta = ctr[i] - x[i] * alpha
        gamma = -b[i] * sigma_x

        x_tilde = (-beta + np.sqrt(
            beta * beta - 4 * alpha * gamma)) / (2 * alpha)
        x_i = x[i]

        ctr = ctr - cov[i] * x_i + cov[i] * x_tilde
        sigma_x = sigma_x * sigma_x - 2 * x_i * cov[i].dot(
            x) + x_i * x_i * var[i]
        x[i] = x_tilde
        sigma_x = np.sqrt(sigma_x + 2 * x_tilde * cov[i].dot(
            x) - x_tilde * x_tilde * var[i])

    # check convergence
    if np.power((x - x0) / x.sum(), 2).sum() < tolerance:
```

```
            return x / x.sum()

        x0 = x.copy()

    # no solution found
    raise ValueError('No solution found after {0} iterations.'.format(
        maximum_iterations))


[docs]def calc_erc_weights(returns,
                      initial_weights=None,
                      risk_weights=None,
                      covar_method='ledoit-wolf',
                      risk_parity_method='ccd',
                      maximum_iterations=100,
                      tolerance=1E-8):
    """
    Calculates the equal risk contribution / risk parity weights given a
    DataFrame of returns.

    Args:
        * returns (DataFrame): Returns for multiple securities.
        * initial_weights (list): Starting asset weights [default inverse vol].
        * risk_weights (list): Risk target weights [default equal weight].
        * covar_method (str): Covariance matrix estimation method.
            Currently supported:
                - ledoit-wolf [default]
                - standard
        * risk_parity_method (str): Risk parity estimation method.
            Currently supported:
                - ccd (cyclical coordinate descent)[default]
        * maximum_iterations (int): Maximum iterations in iterative solutions.
        * tolerance (float): Tolerance level in iterative solutions.

    Returns:
        Series {col_name: weight}

    """
    n = len(returns.columns)

    # calc covariance matrix
    if covar_method == 'ledoit-wolf':
        covar = sklearn.covariance.ledoit_wolf(returns)[0]
    elif covar_method == 'standard':
        covar = returns.cov().values
    else:
        raise NotImplementedError('covar_method not implemented')

    # initial weights (default to inverse vol)
    if initial_weights is None:
```

```
        inv_vol = 1. / np.sqrt(np.diagonal(covar))
        initial_weights = inv_vol / inv_vol.sum()

    # default to equal risk weight
    if risk_weights is None:
        risk_weights = np.ones(n) / n

    # calc risk parity weights matrix
    if risk_parity_method == 'ccd':
        # cyclical coordinate descent implementation
        erc_weights = _erc_weights_ccd(initial_weights,
                                       covar,
                                       risk_weights,
                                       maximum_iterations,
                                       tolerance)
    else:
        raise NotImplementedError('risk_parity_method not implemented')

    # return erc weights vector
    return pd.Series(erc_weights, index=returns.columns, name='erc')



[docs]def get_num_days_required(offset, period='d', perc_required=0.90):
    """
    Estimates the number of days required to assume that data is OK.

    Helper function used to determine if there are enough "good" data
    days over a given period.

    Args:
        * offset (DateOffset): Offset (lookback) period.
        * period (str): Period string.
        * perc_required (float): percentage of number of days
            expected required.

    """
    x = pd.to_datetime('2010-01-01')
    delta = x - (x - offset)
    # convert to 'trading days' - rough guestimate
    days = delta.days * 0.69

    if period == 'd':
        req = days * perc_required
    elif period == 'm':
        req = (days / 20) * perc_required
    elif period == 'y':
        req = (days / 252) * perc_required
    else:
        raise NotImplementedError(
```

```
            'period not supported. Supported periods are d, m, y')

    return req



[docs]def calc_clusters(returns, n=None, plot=False):
    """
    Calculates the clusters based on k-means
    clustering.

    Args:
        * returns (pd.DataFrame): DataFrame of returns
        * n (int): Specify # of clusters. If None, this
            will be automatically determined
        * plot (bool): Show plot?

    Returns:
        * dict with structure: {cluster# : [col names]}
    """
    # calculate correlation
    corr = returns.corr()

    # calculate dissimilarity matrix
    diss = 1 - corr

    # scale down to 2 dimensions using MDS
    # (multi-dimensional scaling) using the
    # dissimilarity matrix
    mds = sklearn.manifold.MDS(dissimilarity='precomputed')
    xy = mds.fit_transform(diss)

    def routine(k):
        # fit KMeans
        km = sklearn.cluster.KMeans(n_clusters=k)
        km_fit = km.fit(xy)
        labels = km_fit.labels_
        centers = km_fit.cluster_centers_

        # get {ticker: label} mappings
        mappings = dict(zip(returns.columns, labels))

        # print % of var explained
        totss = 0
        withinss = 0
        # column average fot totss
        avg = np.array([np.mean(xy[:, 0]), np.mean(xy[:, 1])])
        for idx, lbl in enumerate(labels):
            withinss += sum((xy[idx] - centers[lbl]) ** 2)
            totss += sum((xy[idx] - avg) ** 2)
```

```
            pvar_expl = 1.0 - withinss / totss

            return mappings, pvar_expl, labels

        if n:
            result = routine(n)
        else:
            n = len(returns.columns)
            n1 = int(np.ceil(n * 0.6666666666))
            for i in range(2, n1 + 1):
                result = routine(i)
                if result[1] > 0.9:
                    break

        if plot:
            fig, ax = plt.subplots()
            ax.scatter(xy[:, 0], xy[:, 1], c=result[2], s=90)
            for i, txt in enumerate(returns.columns):
                ax.annotate(txt, (xy[i, 0], xy[i, 1]), size=14)

        # sanitize return value
        tmp = result[0]
        # map as such {cluster: [list of tickers], cluster2: [...]}
        inv_map = {}
        for k, v in iteritems(tmp):
            inv_map[v] = inv_map.get(v, [])
            inv_map[v].append(k)

        return inv_map



[docs]def calc_ftca(returns, threshold=0.5):
    """
    Implementation of David Varadi's Fast Threshold
    Clustering Algorithm (FTCA).

    http://cssanalytics.wordpress.com/2013/11/26/fast-threshold-clustering-algorithm-ftca/  # NOQA

    More stable than k-means for clustering purposes.
    If you want more clusters, use a higher threshold.

    Args:
        * returns - expects a pandas dataframe of returns where
            each column is the name of a given security.
        * threshold (float): Threshold parameter - use higher value
            for more clusters. Basically controls how similar
            (correlated) series have to be.
    Returns:
        dict of cluster name (a number) and list of securities in cluster
```

```
"""
# cluster index (name)
i = 0
# correlation matrix
corr = returns.corr()
# remaining securities to cluster
remain = list(corr.index.copy())
n = len(remain)
res = {}

while n > 0:
    # if only one left then create cluster and finish
    if n == 1:
        i += 1
        res[i] = remain
        n = 0
    # if not then we have some work to do
    else:
        # filter down correlation matrix to current remain
        cur_corr = corr[remain].ix[remain]
        # get mean correlations, ordered
        mc = cur_corr.mean().order()
        # get lowest and highest mean correlation
        low = mc.index[0]
        high = mc.index[-1]

        # case if corr(high,low) > threshold
        if corr[high][low] > threshold:
            i += 1

            # new cluster for high and low
            res[i] = [low, high]
            remain.remove(low)
            remain.remove(high)

            rmv = []
            for x in remain:
                avg_corr = (corr[x][high] + corr[x][low]) / 2.0
                if avg_corr > threshold:
                    res[i].append(x)
                    rmv.append(x)
            [remain.remove(x) for x in rmv]

            n = len(remain)

        # otherwise we are creating two clusters - one for high
        # and one for low
        else:
            # add cluster with HC
```

```
                i += 1
                res[i] = [high]
                remain.remove(high)
                remain.remove(low)

                rmv = []
                for x in remain:
                    if corr[x][high] > threshold:
                        res[i].append(x)
                        rmv.append(x)
                [remain.remove(x) for x in rmv]

                i += 1
                res[i] = [low]

                rmv = []
                for x in remain:
                    if corr[x][low] > threshold:
                        res[i].append(x)
                        rmv.append(x)
                [remain.remove(x) for x in rmv]

                n = len(remain)

    return res



[docs]def limit_weights(weights, limit=0.1):
    """
    Limits weights and redistributes excedent amount
    proportionally.

    ex:
        - weights are {a: 0.7, b: 0.2, c: 0.1}
        - call with limit=0.5
        - excess 0.2 in a is ditributed to b and c
            proportionally.
            - result is {a: 0.5, b: 0.33, c: 0.167}

    Args:
        * weights (Series): A series describing the weights
        * limit (float): Maximum weight allowed
    """
    if 1.0 / limit > len(weights):
        raise ValueError('invalid limit -> 1 / limit must be <= len(weights)')

    if isinstance(weights, dict):
        weights = pd.Series(weights)
```

```
    if np.round(weights.sum(), 1) != 1.0:
        raise ValueError('Expecting weights (that sum to 1) - sum is %s'
                         % weights.sum())

    res = np.round(weights.copy(), 4)
    to_rebalance = (res[res > limit] - limit).sum()

    ok = res[res < limit]
    ok += (ok / ok.sum()) * to_rebalance

    res[res > limit] = limit
    res[res < limit] = ok

    if not np.all([x <= limit for x in res]):
        return limit_weights(res, limit=limit)

    return res



[docs]def random_weights(n, bounds=(0., 1.), total=1.0):
    """
    Generate pseudo-random weights.

    Returns a list of random weights that is of length
    n, where each weight is in the range bounds, and
    where the weights sum up to total.

    Useful for creating random portfolios when benchmarking.

    Args:
        * n (int): number of random weights
        * bounds ((low, high)): bounds for each weight
        * total (float): total sum of the weights

    """
    low = bounds[0]
    high = bounds[1]

    if high < low:
        raise ValueError('Higher bound must be greater or '
                         'equal to lower bound')

    if n * high < total or n * low > total:
        raise ValueError('solution not possible with given n and bounds')

    w = [0] * n
    tgt = -float(total)

    for i in range(n):
```

```
            rn = n - i - 1
            rhigh = rn * high
            rlow = rn * low

            lowb = max(-rhigh - tgt, low)
            highb = min(-rlow - tgt, high)

            rw = random.uniform(lowb, highb)
            w[i] = rw

            tgt += rw

        random.shuffle(w)
        return w



[docs]def plot_heatmap(data, title='Heatmap', show_legend=True,
                    show_labels=True, label_fmt='.2f',
                    vmin=None, vmax=None,
                    figsize=None, label_color='w',
                    cmap='RdBu', **kwargs):
    """
    Plot a heatmap using matplotlib's pcolor.

    Args:
        * data (DataFrame): DataFrame to plot. Usually small matrix (ex.
            correlation matrix).
        * title (string): Plot title
        * show_legend (bool): Show color legend
        * show_labels (bool): Show value labels
        * label_fmt (str): Label format string
        * vmin (float): Min value for scale
        * vmax (float): Max value for scale
        * cmap (string): Color map
        * kwargs: Passed to matplotlib's pcolor

    """
    fig, ax = plt.subplots(figsize=figsize)

    heatmap = ax.pcolor(data, vmin=vmin, vmax=vmax, cmap=cmap)
    # for some reason heatmap has the y values backwards....
    ax.invert_yaxis()

    if title is not None:
        plt.title(title)

    if show_legend:
        fig.colorbar(heatmap)
```

```
    if show_labels:
        vals = data.values
        for x in range(data.shape[0]):
            for y in range(data.shape[1]):
                plt.text(x + 0.5, y + 0.5, format(vals[y, x], label_fmt),
                         horizontalalignment='center',
                         verticalalignment='center',
                         color=label_color)

    plt.yticks(np.arange(0.5, len(data.index), 1), data.index)
    plt.xticks(np.arange(0.5, len(data.columns), 1), data.columns)

    return plt



[docs]def plot_corr_heatmap(data, **kwargs):
    """
    Plots the correlation heatmap for a given DataFrame.
    """
    return plot_heatmap(data.corr(), vmin=-1, vmax=1, **kwargs)



[docs]def rollapply(data, window, fn):
    """
    Apply a function fn over a rolling window of size window.

    Args:
        * data (Series or DataFrame): Series or DataFrame
        * window (int): Window size
        * fn (function): Function to apply over the rolling window.
            For a series, the return value is expected to be a single
            number. For a DataFrame, it shuold return a new row.

    Returns:
        * Object of same dimensions as data
    """
    res = data.copy()
    res[:] = np.nan
    n = len(data)

    if window > n:
        return res

    for i in range(window - 1, n):
        res.iloc[i] = fn(data.iloc[i - window + 1:i + 1])

    return res
```

```
def _winsorize_wrapper(x, limits):
    """
    Wraps scipy winsorize function to drop na's
    """
    if hasattr(x, 'dropna'):
        if len(x.dropna()) == 0:
            return x

        x[~np.isnan(x)] = scipy.stats.mstats.winsorize(x[~np.isnan(x)],
                                                       limits=limits)
        return x
    else:
        return scipy.stats.mstats.winsorize(x, limits=limits)


[docs]def winsorize(x, axis=0, limits=0.01):
    """
    Winsorize values based on limits
    """
    # operate on copy
    x = x.copy()

    if isinstance(x, pd.DataFrame):
        return x.apply(_winsorize_wrapper, axis=axis, args=(limits, ))
    else:
        return pd.Series(_winsorize_wrapper(x, limits).values,
                         index=x.index)



[docs]def rescale(x, min=0., max=1., axis=0):
    """
    Rescale values to fit a certain range [min, max]
    """
    def innerfn(x, min, max):
        return np.interp(x, [np.min(x), np.max(x)], [min, max])

    if isinstance(x, pd.DataFrame):
        return x.apply(innerfn, axis=axis, args=(min, max,))
    else:
        return pd.Series(innerfn(x, min, max), index=x.index)



[docs]def annualize(returns, durations, one_year=365.):
    """
    Annualize returns using their respective durations.
```

```
    Formula used is:
        (1 + returns) ** (1 / (durations / one_year)) - 1

    """
    return (1. + returns) ** (1. / (durations / one_year)) - 1.



[docs]def deannualize(returns, nperiods):
    """
    Convert return expressed in annual terms on a different basis.

    Args:
        * returns (float, Series, DataFrame): Return(s)
        * nperiods (int): Target basis, typically 252 for daily, 12 for
            monthly, etc.

    """
    return np.power(1 + returns, 1. / nperiods) - 1.



[docs]def calc_sortino_ratio(returns, rf=0, nperiods=None, annualize=True):
    """
    Calculates the sortino ratio given a series of returns

    Args:
        * returns (Series or DataFrame): Returns
        * rf (float): Risk-free rate expressed in yearly (annualized) terms.
        * nperiods (int): Number of periods used for annualization. Must be
            provided if rf is non-zero

    """
    if rf != 0 and nperiods is None:
        raise Exception('nperiods must be set if rf != 0')

    er = returns.to_excess_returns(rf, nperiods=nperiods)
    res = er.mean() / er[er < 0].std()

    if annualize:
        if nperiods is None:
            nperiods = 1
        return res * np.sqrt(nperiods)

    return res



[docs]def to_excess_returns(returns, rf, nperiods=None):
    """
```

```
    Given a series of returns, it will return the excess returns over rf.

    Args:
        * returns (Series, DataFrame): Returns
        * rf (float, Series, DataFrame): Risk-Free rate(s)
        * nperiods (int): Optional. If provided, will convert rf to different
            frequency using deannualize
    Returns:
        * excess_returns (Series, DataFrame): Returns - rf

    """
    if nperiods is not None:
        _rf = deannualize(rf, nperiods)
    else:
        _rf = rf

    return returns - _rf



[docs]def calc_calmar_ratio(prices):
    """
    Calculates the Calmar Ratio given a series of prices

    Args:
        * prices (Series, DataFrame): Price series

    """
    return prices.calc_cagr() / abs(prices.calc_max_drawdown())



[docs]def to_ulcer_index(prices):
    """
    Converts from prices -> Ulcer index

    See https://en.wikipedia.org/wiki/Ulcer_index

    Args:
        * prices (Series, DataFrame): Prices

    """
    dd = prices.to_drawdown_series()
    return np.sqrt(np.sum(dd ** 2)) / dd.count()



[docs]def to_ulcer_performance_index(prices, rf=0., nperiods=None):
    """
    Converts from prices -> ulcer performance index.
```

```
    See https://en.wikipedia.org/wiki/Ulcer_index

    Args:
        * prices (Series, DataFrame): Prices
        * rf (float): Risk-free rate of return. Assumed to be expressed in
            yearly (annualized) terms
        * nperiods (int): Used to deannualize rf if rf is provided (non-zero)

    """
    if rf != 0 and nperiods is None:
        raise Exception('nperiods must be set if rf != 0')

    er = prices.to_returns().to_excess_returns(rf, nperiods=nperiods)

    return er.mean() / prices.to_ulcer_index()



[docs]def extend_pandas():
    """
    Extends pandas' PandasObject (Series, Series,
    DataFrame) with some functions defined in this file.

    This facilitates common functional composition used in quant
    finance.

    Ex:
        prices.to_returns().dropna().calc_clusters()
        (where prices would be a DataFrame)
    """
    PandasObject.to_returns = to_returns
    PandasObject.to_log_returns = to_log_returns
    PandasObject.to_price_index = to_price_index
    PandasObject.rebase = rebase
    PandasObject.calc_perf_stats = calc_perf_stats
    PandasObject.to_drawdown_series = to_drawdown_series
    PandasObject.calc_max_drawdown = calc_max_drawdown
    PandasObject.calc_cagr = calc_cagr
    PandasObject.calc_total_return = calc_total_return
    PandasObject.as_percent = utils.as_percent
    PandasObject.as_format = utils.as_format
    PandasObject.to_monthly = to_monthly
    PandasObject.asfreq_actual = asfreq_actual
    PandasObject.drop_duplicate_cols = drop_duplicate_cols
    PandasObject.calc_information_ratio = calc_information_ratio
    PandasObject.calc_prob_mom = calc_prob_mom
    PandasObject.calc_risk_return_ratio = calc_risk_return_ratio
    PandasObject.calc_erc_weights = calc_erc_weights
    PandasObject.calc_inv_vol_weights = calc_inv_vol_weights
```

```
PandasObject.calc_mean_var_weights = calc_mean_var_weights
PandasObject.calc_clusters = calc_clusters
PandasObject.calc_ftca = calc_ftca
PandasObject.calc_stats = calc_stats
PandasObject.plot_heatmap = plot_heatmap
PandasObject.plot_corr_heatmap = plot_corr_heatmap
PandasObject.rollapply = rollapply
PandasObject.winsorize = winsorize
PandasObject.rescale = rescale
PandasObject.calc_sortino_ratio = calc_sortino_ratio
PandasObject.calc_calmar_ratio = calc_calmar_ratio
PandasObject.calc_sharpe = calc_sharpe
PandasObject.to_excess_returns = to_excess_returns
PandasObject.to_ulcer_index = to_ulcer_index
PandasObject.to_ulcer_performance_index = to_ulcer_performance_index
```