bt.algos — bt 0.2.5 documentation

```
"""
A collection of Algos used to create Strategy logic.
"""
from __future__ import division
from future.utils import iteritems
import bt
from bt.core import Algo, AlgoStack
import pandas as pd
import numpy as np
import random


[docs]def run_always(f):
    """
    Run always decorator to be used with Algo
    to ensure stack runs the decorated Algo
    on each pass, regardless of failures in the stack.
    """
    f.run_always = True
    return f



[docs]class PrintDate(Algo):

    """
    This Algo simply print's the current date.

    Can be useful for debugging purposes.
    """

    def __call__(self, target):
        print(target.now)
        return True



[docs]class PrintTempData(Algo):

    """
    This Algo prints the temp data.

    Useful for debugging.
    """

    def __call__(self, target):
```

```
        print(target.temp)
        return True



[docs]class PrintInfo(Algo):

    """
    Prints out info associated with the target strategy. Useful for debugging
    purposes.

    Args:
        * fmt_string (str): A string that will later be formatted with the
            target object's __dict__ attribute. Therefore, you should provide
            what you want to examine within curly braces ( { } )

    Ex:
        PrintInfo('Strategy {name} : {now}')



    This will print out the name and the date (now) on each call.
    Basically, you provide a string that will be formatted with target.__dict__

    """

    def __init__(self, fmt_string='{full_name} {now}'):
        self.fmt_string = fmt_string

    def __call__(self, target):
        print(self.fmt_string.format(target.__dict__))
        return True



[docs]class Debug(Algo):

    """
    Utility Algo that calls pdb.set_trace when triggered.

    In the debug session, target is available and can be examined.
    """

    def __call__(self, target):
        import pdb
        pdb.set_trace()
        return True



[docs]class RunOnce(Algo):
```

```
    """
    Returns True on first run then returns False.

    As the name says, the algo only runs once. Useful in situations
    where we want to run the logic once (buy and hold for example).
    """

    def __init__(self):
        super(RunOnce, self).__init__()
        self.has_run = False

    def __call__(self, target):
        # if it hasn't run then we will
        # run it and set flag
        if not self.has_run:
            self.has_run = True
            return True

        # return false to stop future execution
        return False
```

[docs]

```
class RunDaily(Algo):

    """
    Returns True on day change.

    Returns True if the target.now's day has changed
    since the last run, if not returns False. Useful for
    daily rebalancing strategies.

    """

    def __init__(self):
        super(RunDaily, self).__init__()
        self.last_date = None

    def __call__(self, target):
        # get last date
        now = target.now

        # if none nothing to do - return false
        if now is None:
            return False

        # create pandas.Timestamp for useful .week property
        now = pd.Timestamp(now)
```

```
        if self.last_date is None:
            self.last_date = now
            return False

        result = False
        if now.date() != self.last_date.date():
            result = True

        self.last_date = now
        return result
```

```
[docs]class RunWeekly(Algo):

    """
    Returns True on week change.

    Returns True if the target.now's week has changed
    since the last run, if not returns False. Useful for
    weekly rebalancing strategies.

    Note:
        This algo will typically run on the first day of the
        week (assuming we have daily data)

    """

    def __init__(self):
        super(RunWeekly, self).__init__()
        self.last_date = None

    def __call__(self, target):
        # get last date
        now = target.now

        # if none nothing to do - return false
        if now is None:
            return False

        # create pandas.Timestamp for useful .week property
        now = pd.Timestamp(now)

        if self.last_date is None:
            self.last_date = now
            return False

        result = False
        if now.week != self.last_date.week:
            result = True
```

```
        self.last_date = now
        return result



[docs]class RunMonthly(Algo):

    """
    Returns True on month change.

    Returns True if the target.now's month has changed
    since the last run, if not returns False. Useful for
    monthly rebalancing strategies.

    Note:
        This algo will typically run on the first day of the
        month (assuming we have daily data)

    """

    def __init__(self):
        super(RunMonthly, self).__init__()
        self.last_date = None

    def __call__(self, target):
        # get last date
        now = target.now

        # if none nothing to do - return false
        if now is None:
            return False

        if self.last_date is None:
            self.last_date = now
            return False

        result = False
        if now.month != self.last_date.month:
            result = True

        self.last_date = now
        return result



[docs]class RunQuarterly(Algo):

    """
    Returns True on quarter change.
```

```
    Returns True if the target.now's month has changed
    since the last run and the month is the first month
    of the quarter, if not returns False. Useful for
    quarterly rebalancing strategies.

    Note:
        This algo will typically run on the first day of the
        quarter (assuming we have daily data)

    """

    def __init__(self):
        super(RunQuarterly, self).__init__()
        self.last_date = None

    def __call__(self, target):
        # get last date
        now = target.now

        # if none nothing to do - return false
        if now is None:
            return False

        if self.last_date is None:
            self.last_date = now
            return False

        result = False
        if now.quarter != self.last_date.quarter:
            result = True

        self.last_date = now
        return result



[docs]class RunYearly(Algo):

    """
    Returns True on year change.

    Returns True if the target.now's year has changed
    since the last run, if not returns False. Useful for
    yearly rebalancing strategies.

    Note:
        This algo will typically run on the first day of the
        year (assuming we have daily data)
```

```
    """

    def __init__(self):
        super(RunYearly, self).__init__()
        self.last_date = None

    def __call__(self, target):
        # get last date
        now = target.now

        # if none nothing to do - return false
        if now is None:
            return False

        if self.last_date is None:
            self.last_date = now
            return False

        result = False
        if now.year != self.last_date.year:
            result = True

        self.last_date = now
        return result



[docs]class RunOnDate(Algo):

    """
    Returns True on a specific set of dates.

    Args:
        * dates (list): List of dates to run Algo on.

    """

    def __init__(self, *dates):
        """
        Args:
            * dates (*args): A list of dates. Dates will be parsed
                by pandas.to_datetime so pass anything that it can
                parse. Typically, you will pass a string 'yyyy-mm-dd'.
        """
        super(RunOnDate, self).__init__()
        # parse dates and save
        self.dates = [pd.to_datetime(d) for d in dates]

    def __call__(self, target):
        return target.now in self.dates
```

```
[docs]class RunAfterDate(Algo):

    """
    Returns True after a date has passed

    Args:
        * date: Date after which to start trading

    Note:
        This is useful for algos that rely on trailing averages where you
        don't want to start trading until some amount of data has been built up

    """

    def __init__(self, date):
        """
        Args:
            * date: Date after which to start trading
        """
        super(RunAfterDate, self).__init__()
        # parse dates and save
        self.date = pd.to_datetime(date)

    def __call__(self, target):
        return target.now > self.date



[docs]class RunAfterDays(Algo):

    """
    Returns True after a specific number of 'warmup' trading days have passed

    Args:
        * days (int): Number of trading days to wait before starting

    Note:
        This is useful for algos that rely on trailing averages where you
        don't want to start trading until some amount of data has been built up

    """

    def __init__(self, days):
        """
        Args:
            * days (int): Number of trading days to wait before starting
        """
```

```
        super(RunAfterDays, self).__init__()
        self.days = days

    def __call__(self, target):
        if self.days > 0:
            self.days -= 1
            return False
        return True



[docs]class RunEveryNPeriods(Algo):

    """
    This algo runs every n periods.

    Args:
        * n (int): Run each n periods
        * offset (int): Applies to the first run. If 0, this algo will run the
            first time it is called.

    This Algo can be useful for the following type of strategy:
        Each month, select the top 5 performers. Hold them for 3 months.

    You could then create 3 strategies with different offsets and create a
    master strategy that would allocate equal amounts of capital to each.

    """

    def __init__(self, n, offset=0):
        self.n = n
        self.offset = offset
        self.idx = n - offset - 1
        self.lcall = 0

    def __call__(self, target):
        # ignore multiple calls on same period
        if self.lcall == target.now:
            return False
        else:
            self.lcall = target.now
            # run when idx == (n-1)
            if self.idx == (self.n - 1):
                self.idx = 0
                return True
            else:
                self.idx += 1
                return False
```

```
[docs]class SelectAll(Algo):

    """
    Sets temp['selected'] with all securities (based on universe).

    Selects all the securities and saves them in temp['selected'].
    By default, SelectAll does not include securities that have no
    data (nan) on current date or those whose price is zero.

    Args:
        * include_no_data (bool): Include securities that do not have data?

    Sets:
        * selected

    """

    def __init__(self, include_no_data=False):
        super(SelectAll, self).__init__()
        self.include_no_data = include_no_data

    def __call__(self, target):
        if self.include_no_data:
            target.temp['selected'] = target.universe.columns
        else:
            universe = target.universe.ix[target.now].dropna()
            target.temp['selected'] = list(universe[universe > 0].index)
        return True



[docs]class SelectThese(Algo):

    """
    Sets temp['selected'] with a set list of tickers.

    Sets the temp['selected'] to a set list of tickers.

    Args:
        * ticker (list): List of tickers to select.

    Sets:
        * selected

    """

    def __init__(self, tickers, include_no_data=False):
        super(SelectThese, self).__init__()
        self.tickers = tickers
```

```
        self.include_no_data = include_no_data

    def __call__(self, target):
        if self.include_no_data:
            target.temp['selected'] = self.tickers
        else:
            universe = target.universe[self.tickers].ix[target.now].dropna()
            target.temp['selected'] = list(universe[universe > 0].index)
        return True
```

```
[docs]class SelectHasData(Algo):

    """
    Sets temp['selected'] based on all items in universe that meet
    data requirements.

    This is a more advanced version of SelectAll. Useful for selecting
    tickers that need a certain amount of data for future algos to run
    properly.

    For example, if we need the items with 3 months of data or more,
    we could use this Algo with a lookback period of 3 months.

    When providing a lookback period, it is also wise to provide a min_count.
    This is basically the number of data points needed within the lookback
    period for a series to be considered valid. For example, in our 3 month
    lookback above, we might want to specify the min_count as being
    57 -> a typical trading month has give or take 20 trading days. If we
    factor in some holidays, we can use 57 or 58. It's really up to you.

    If you don't specify min_count, min_count will default to ffn's
    get_num_days_required.

    Args:
        * lookback (DateOffset): A DateOffset that determines the lookback
            period.
        * min_count (int): Minimum number of days required for a series to be
            considered valid. If not provided, ffn's get_num_days_required is
            used to estimate the number of points required.

    Sets:
        * selected

    """

    def __init__(self, lookback=pd.DateOffset(months=3),
                 min_count=None, include_no_data=False):
        super(SelectHasData, self).__init__()
```

```
        self.lookback = lookback
        if min_count is None:
            min_count = bt.ffn.get_num_days_required(lookback)
        self.min_count = min_count
        self.include_no_data = include_no_data

    def __call__(self, target):
        if 'selected' in target.temp:
            selected = target.temp['selected']
        else:
            selected = target.universe.columns

        filt = target.universe[selected].ix[target.now - self.lookback:]
        cnt = filt.count()
        cnt = cnt[cnt >= self.min_count]
        if not self.include_no_data:
            cnt = cnt[target.universe[selected].ix[target.now] > 0]
        target.temp['selected'] = list(cnt.index)
        return True



[docs]class SelectN(Algo):

    """
    Sets temp['selected'] based on ranking temp['stat'].

    Selects the top or botton N items based on temp['stat'].
    This is usually some kind of metric that will be computed in a
    previous Algo and will be used for ranking purposes. Can select
    top or bottom N based on sort_descending parameter.

    Args:
        * n (int): select top n items.
        * sort_descending (bool): Should the stat be sorted in descending order
            before selecting the first n items?
        * all_or_none (bool): If true, only populates temp['selected'] if we
            have n items. If we have less than n, then temp['selected'] = [].

    Sets:
        * selected

    Requires:
        * stat

    """

    def __init__(self, n, sort_descending=True,
                 all_or_none=False):
        super(SelectN, self).__init__()
```

```
        if n < 0:
            raise ValueError('n cannot be negative')
        self.n = n
        self.ascending = not sort_descending
        self.all_or_none = all_or_none

    def __call__(self, target):
        stat = target.temp['stat'].dropna()
        stat.sort_values(ascending=self.ascending,
                         inplace=True)

        # handle percent n
        keep_n = self.n
        if self.n < 1:
            keep_n = int(self.n * len(stat))

        sel = list(stat[:keep_n].index)

        if self.all_or_none and len(sel) < keep_n:
            sel = []

        target.temp['selected'] = sel

        return True
```

[docs]

```
class SelectMomentum(AlgoStack):

    """
    Sets temp['selected'] based on a simple momentum filter.

    Selects the top n securities based on the total return over
    a given lookback period. This is just a wrapper around an
    AlgoStack with two algos: StatTotalReturn and SelectN.

    Note, that SelectAll() or similar should be called before
    SelectMomentum(), as StatTotalReturn uses values of temp['selected']

    Args:
        * n (int): select first N elements
        * lookback (DateOffset): lookback period for total return
            calculation
        * lag (DateOffset): Lag interval for total return calculation
        * sort_descending (bool): Sort descending (highest return is best)
        * all_or_none (bool): If true, only populates temp['selected'] if we
            have n items. If we have less than n, then temp['selected'] = [].

    Sets:
        * selected
```

```
    Requires:
        * selected

    """

    def __init__(self, n, lookback=pd.DateOffset(months=3),
                 lag=pd.DateOffset(days=0), sort_descending=True,
                 all_or_none=False):
        super(SelectMomentum, self).__init__(
            StatTotalReturn(lookback=lookback, lag=lag),
            SelectN(n=n, sort_descending=sort_descending,
                    all_or_none=all_or_none))
```

```
[docs]class SelectWhere(Algo):

    """
    Selects securities based on an indicator DataFrame.

    Selects securities where the value is True on the current date
    (target.now) only if current date is present in signal DataFrame.

    For example, this could be the result of a pandas boolean comparison such
    as data > 100.

    Args:
        * signal (DataFrame): Boolean DataFrame containing selection logic.

    Sets:
        * selected

    """

    def __init__(self, signal, include_no_data=False):
        self.signal = signal
        self.include_no_data = include_no_data

    def __call__(self, target):
        # get signal Series at target.now
        if target.now in self.signal.index:
            sig = self.signal.ix[target.now]
            # get tickers where True
            selected = sig.index[sig]
            # save as list
            if not self.include_no_data:
                universe = target.universe[
                    list(selected)].ix[target.now].dropna()
                selected = list(universe[universe > 0].index)
```

```
            target.temp['selected'] = list(selected)

        return True



[docs]class SelectRandomly(AlgoStack):

    """
    Sets temp['selected'] based on a random subset of
    the items currently in temp['selected'].

    Selects n random elements from the list stored in temp['selected'].
    This is useful for benchmarking against a strategy where we believe
    the selection algorithm is adding value.

    For example, if we are testing a momentum strategy and we want to see if
    selecting securities based on momentum is better than just selecting
    securities randomly, we could use this Algo to create a random Strategy
    used for random benchmarking.

    Note:
        Another selection algorithm should be use prior to this Algo to
        populate temp['selected']. This will typically be SelectAll.

    Args:
        * n (int): Select N elements randomly.

    Sets:
        * selected

    Requires:
        * selected

    """

    def __init__(self, n=None, include_no_data=False):
        super(SelectRandomly, self).__init__()
        self.n = n
        self.include_no_data = include_no_data

    def __call__(self, target):
        if 'selected' in target.temp:
            sel = target.temp['selected']
        else:
            sel = target.universe.columns

        if not self.include_no_data:
            universe = target.universe[list(sel)].ix[target.now].dropna()
            sel = list(universe[universe > 0].index)
```

```
        if self.n is not None:
            n = self.n if self.n < len(sel) else len(sel)
            sel = random.sample(sel, int(n))

        target.temp['selected'] = sel
        return True
```

```
[docs]class StatTotalReturn(Algo):

    """
    Sets temp['stat'] with total returns over a given period.

    Sets the 'stat' based on the total return of each element in
    temp['selected'] over a given lookback period. The total return
    is determined by ffn's calc_total_return.

    Args:
        * lookback (DateOffset): lookback period.
        * lag (DateOffset): Lag interval. Total return is calculated in
            the inteval [now - lookback - lag, now - lag]

    Sets:
        * stat

    Requires:
        * selected

    """

    def __init__(self, lookback=pd.DateOffset(months=3),
                 lag=pd.DateOffset(days=0)):
        super(StatTotalReturn, self).__init__()
        self.lookback = lookback
        self.lag = lag

    def __call__(self, target):
        selected = target.temp['selected']
        t0 = target.now - self.lag
        prc = target.universe[selected].ix[t0 - self.lookback:t0]
        target.temp['stat'] = prc.calc_total_return()
        return True
```

```
[docs]class WeighEqually(Algo):

    """
```

```
    Sets temp['weights'] by calculating equal weights for all items in
    selected.

    Equal weight Algo. Sets the 'weights' to 1/n for each item in 'selected'.

    Sets:
        * weights

    Requires:
        * selected

    """

    def __init__(self):
        super(WeighEqually, self).__init__()

    def __call__(self, target):
        selected = target.temp['selected']
        n = len(selected)

        if n == 0:
            target.temp['weights'] = {}
        else:
            w = 1.0 / n
            target.temp['weights'] = {x: w for x in selected}

        return True



[docs]class WeighSpecified(Algo):

    """
    Sets temp['weights'] based on a provided dict of ticker:weights.

    Sets the weights based on pre-specified targets.

    Args:
        * weights (dict): target weights -> ticker: weight

    Sets:
        * weights

    """

    def __init__(self, **weights):
        super(WeighSpecified, self).__init__()
        self.weights = weights

    def __call__(self, target):
```

```
        # added copy to make sure these are not overwritten
        target.temp['weights'] = self.weights.copy()
        return True
```

[docs]class WeighTarget(Algo):

```
    """
    Sets target weights based on a target weight DataFrame.

    If the target weight dataFrame is  of same dimension
    as the target.universe, the portfolio will effectively be rebalanced on
    each period. For example, if we have daily data and the target DataFrame
    is of the same shape, we will have daily rebalancing.

    However, if we provide a target weight dataframe that has only month end
    dates, then rebalancing only occurs monthly.

    Basically, if a weight is provided on a given date, the target weights are
    set and the algo moves on (presumably to a Rebalance algo). If not, not
    target weights are set.

    Args:
        * weights (DataFrame): DataFrame containing the target weights

    Sets:
        * weights

    """

    def __init__(self, weights):
        self.weights = weights

    def __call__(self, target):
        # get current target weights
        if target.now in self.weights.index:
            w = self.weights.ix[target.now]

            # dropna and save
            target.temp['weights'] = w.dropna()

            return True
        else:
            return False
```

[docs]class WeighInvVol(Algo):

```
    """
    Sets temp['weights'] based on the inverse volatility Algo.

    Sets the target weights based on ffn's calc_inv_vol_weights. This
    is a commonly used technique for risk parity portfolios. The least
    volatile elements receive the highest weight under this scheme. Weights
    are proportional to the inverse of their volatility.

    Args:
        * lookback (DateOffset): lookback period for estimating volatility

    Sets:
        * weights

    Requires:
        * selected

    """

    def __init__(self, lookback=pd.DateOffset(months=3),
                 lag=pd.DateOffset(days=0)):
        super(WeighInvVol, self).__init__()
        self.lookback = lookback
        self.lag = lag

    def __call__(self, target):
        selected = target.temp['selected']

        if len(selected) == 0:
            target.temp['weights'] = {}
            return True

        if len(selected) == 1:
            target.temp['weights'] = {selected[0]: 1.}
            return True

        t0 = target.now - self.lag
        prc = target.universe[selected].ix[t0 - self.lookback:t0]
        tw = bt.ffn.calc_inv_vol_weights(
            prc.to_returns().dropna())
        target.temp['weights'] = tw.dropna()
        return True



[docs]class WeighERC(Algo):

    """
    Sets temp['weights'] based on equal risk contribution algorithm.
```

```
Sets the target weights based on ffn's calc_erc_weights. This
is an extension of the inverse volatility risk parity portfolio in
which the correlation of asset returns is incorporated into the
calculation of risk contribution of each asset.

The resulting portfolio is similar to a minimum variance portfolio
subject to a diversification constraint on the weights of its components
and its volatility is located between those of the minimum variance and
equally-weighted portfolios (Maillard 2008).

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

Args:
    * lookback (DateOffset): lookback period for estimating covariance
    * initial_weights (list): Starting asset weights [default inverse vol].
    * risk_weights (list): Risk target weights [default equal weight].
    * covar_method (str): method used to estimate the covariance. See ffn's
        calc_erc_weights for more details. (default ledoit-wolf).
    * risk_parity_method (str): Risk parity estimation method. see ffn's
        calc_erc_weights for more details. (default ccd).
    * maximum_iterations (int): Maximum iterations in iterative solutions
        (default 100).
    * tolerance (float): Tolerance level in iterative solutions (default 1E-8).


Sets:
    * weights

Requires:
    * selected

"""

def __init__(self,
             lookback=pd.DateOffset(months=3),
             initial_weights=None,
             risk_weights=None,
             covar_method='ledoit-wolf',
             risk_parity_method='ccd',
             maximum_iterations=100,
             tolerance=1E-8,
             lag=pd.DateOffset(days=0)):

    super(WeighERC, self).__init__()
    self.lookback = lookback
    self.initial_weights = initial_weights
    self.risk_weights = risk_weights
    self.covar_method = covar_method
    self.risk_parity_method = risk_parity_method
```

```
        self.maximum_iterations = maximum_iterations
        self.tolerance = tolerance
        self.lag = lag

    def __call__(self, target):
        selected = target.temp['selected']

        if len(selected) == 0:
            target.temp['weights'] = {}
            return True

        if len(selected) == 1:
            target.temp['weights'] = {selected[0]: 1.}
            return True

        t0 = target.now - self.lag
        prc = target.universe[selected].ix[t0 - self.lookback:t0]
        tw = bt.ffn.calc_erc_weights(
            prc.to_returns().dropna(),
            initial_weights=self.initial_weights,
            risk_weights=self.risk_weights,
            covar_method=self.covar_method,
            risk_parity_method=self.risk_parity_method,
            maximum_iterations=self.maximum_iterations,
            tolerance=self.tolerance)

        target.temp['weights'] = tw.dropna()
        return True



[docs]class WeighMeanVar(Algo):

    """
    Sets temp['weights'] based on mean-variance optimization.

    Sets the target weights based on ffn's calc_mean_var_weights. This is a
    Python implementation of Markowitz's mean-variance optimization.

    See:
        http://en.wikipedia.org/wiki/Modern_portfolio_theory#The_efficient_frontier_with_no_risk-free_asset

    Args:
        * lookback (DateOffset): lookback period for estimating volatility
        * bounds ((min, max)): tuple specifying the min and max weights for
            each asset in the optimization.
        * covar_method (str): method used to estimate the covariance. See ffn's
            calc_mean_var_weights for more details.
        * rf (float): risk-free rate used in optimization.
```

```
    Sets:
        * weights

    Requires:
        * selected

    """

    def __init__(self, lookback=pd.DateOffset(months=3),
                 bounds=(0., 1.), covar_method='ledoit-wolf',
                 rf=0., lag=pd.DateOffset(days=0)):
        super(WeighMeanVar, self).__init__()
        self.lookback = lookback
        self.lag = lag
        self.bounds = bounds
        self.covar_method = covar_method
        self.rf = rf

    def __call__(self, target):
        selected = target.temp['selected']

        if len(selected) == 0:
            target.temp['weights'] = {}
            return True

        if len(selected) == 1:
            target.temp['weights'] = {selected[0]: 1.}
            return True

        t0 = target.now - self.lag
        prc = target.universe[selected].ix[t0 - self.lookback:t0]
        tw = bt.ffn.calc_mean_var_weights(
            prc.to_returns().dropna(), weight_bounds=self.bounds,
            covar_method=self.covar_method, rf=self.rf)

        target.temp['weights'] = tw.dropna()
        return True



[docs]class WeighRandomly(Algo):

    """
    Sets temp['weights'] based on a random weight vector.

    Sets random target weights for each security in 'selected'.
    This is useful for benchmarking against a strategy where we believe
    the weighing algorithm is adding value.

    For example, if we are testing a low-vol strategy and we want to see if
```

```
    our weighing strategy is better than just weighing
    securities randomly, we could use this Algo to create a random Strategy
    used for random benchmarking.

    This is an Algo wrapper around ffn's random_weights function.

    Args:
        * bounds ((low, high)): Tuple including low and high bounds for each
            security
        * weight_sum (float): What should the weights sum up to?

    Sets:
        * weights

    Requires:
        * selected

    """

    def __init__(self, bounds=(0., 1.), weight_sum=1):
        super(WeighRandomly, self).__init__()
        self.bounds = bounds
        self.weight_sum = weight_sum

    def __call__(self, target):
        sel = target.temp['selected']
        n = len(sel)

        w = {}
        try:
            rw = bt.ffn.random_weights(
                n, self.bounds, self.weight_sum)
            w = dict(list(zip(sel, rw)))
        except ValueError:
            pass

        target.temp['weights'] = w
        return True
```

[docs]

```
class LimitDeltas(Algo):

    """
    Modifies temp['weights'] based on weight delta limits.

    Basically, this can be used if we want to restrict how much a security's
    target weight can change from day to day. Useful when we want to be more
    conservative about how much we could actually trade on a given day without
    affecting the market.
```

```
    For example, if we have a strategy that is currently long 100% one
    security, and the weighing Algo sets the new weight to 0%, but we
    use this Algo with a limit of 0.1, the new target weight will
    be 90% instead of 0%.

    Args:
        * limit (float, dict): Weight delta limit. If float, this will be a
            global limit for all securities. If dict, you may specify by-ticker
            limit.

    Sets:
        * weights

    Requires:
        * weights

    """

    def __init__(self, limit=0.1):
        super(LimitDeltas, self).__init__()
        self.limit = limit
        # determine if global or specific
        self.global_limit = True
        if isinstance(limit, dict):
            self.global_limit = False

    def __call__(self, target):
        tw = target.temp['weights']
        all_keys = set(list(target.children.keys()) + list(tw.keys()))

        for k in all_keys:
            tgt = tw[k] if k in tw else 0.
            cur = target.children[k].weight if k in target.children else 0.
            delta = tgt - cur

            # check if we need to limit
            if self.global_limit:
                if abs(delta) > self.limit:
                    tw[k] = cur + (self.limit * np.sign(delta))
            else:
                # make sure we have a limit defined in case of limit dict
                if k in self.limit:
                    lmt = self.limit[k]
                    if abs(delta) > lmt:
                        tw[k] = cur + (lmt * np.sign(delta))

        return True
```

```
[docs]class LimitWeights(Algo):

    """
    Modifies temp['weights'] based on weight limits.

    This is an Algo wrapper around ffn's limit_weights. The purpose of this
    Algo is to limit the weight of any one specifc asset. For example, some
    Algos will set some rather extreme weights that may not be acceptable.
    Therefore, we can use this Algo to limit the extreme weights. The excess
    weight is then redistributed to the other assets, proportionally to
    their current weights.

    See ffn's limit_weights for more information.

    Args:
        * limit (float): Weight limit.

    Sets:
        * weights

    Requires:
        * weights

    """

    def __init__(self, limit=0.1):
        super(LimitWeights, self).__init__()
        self.limit = limit

    def __call__(self, target):
        if 'weights' not in target.temp:
            return True

        tw = target.temp['weights']
        if len(tw) == 0:
            return True

        tw = bt.ffn.limit_weights(tw, self.limit)
        target.temp['weights'] = tw

        return True



[docs]class CapitalFlow(Algo):

    """
    Used to model capital flows. Flows can either be inflows or outflows.
```

```
    This Algo can be used to model capital flows. For example, a pension
    fund might have inflows every month or year due to contributions. This
    Algo will affect the capital of the target node without affecting returns
    for the node.

    Since this is modeled as an adjustment, the capital will remain in the
    strategy until a re-allocation/rebalancement is made.

    Args:
        * amount (float): Amount of adjustment

    """

    def __init__(self, amount):
        """
        CapitalFlow constructor.

        Args:
            * amount (float): Amount to adjust by
        """
        super(CapitalFlow, self).__init__()
        self.amount = float(amount)

    def __call__(self, target):
        target.adjust(self.amount)
        return True
```

[docs]

```
class CloseDead(Algo):

    """
    Closes all positions for which prices are equal to zero (we assume
    that these stocks are dead) and removes them from temp['weights'] if
    they enter it by any chance.
    To be called before Rebalance().

    In a normal workflow it is not needed, as those securities will not
    be selected by SelectAll(include_no_data=False) or similar method, and
    Rebalance() closes positions that are not in temp['weights'] anyway.
    However in case when for some reasons include_no_data=False could not
    be used or some modified weighting method is used, CloseDead() will
    allow to avoid errors.

    Requires:
        * weights

    """

    def __init__(self):
```

```
        super(CloseDead, self).__init__()

    def __call__(self, target):
        if 'weights' not in target.temp:
            return True

        targets = target.temp['weights']
        for c in target.children:
            if target.universe[c].ix[target.now] <= 0:
                target.close(c)
                if c in targets:
                    del targets[c]

        return True
```

[docs]

```
class Rebalance(Algo):

    """
    Rebalances capital based on temp['weights']

    Rebalances capital based on temp['weights']. Also closes
    positions if open but not in target_weights. This is typically
    the last Algo called once the target weights have been set.

    Requires:
        * weights
        * cash (optional): You can set a 'cash' value on temp. This should be a
            number between 0-1 and determines the amount of cash to set aside.
            For example, if cash=0.3, the strategy will allocate 70% of its
            value to the provided weights, and the remaining 30% will be kept
            in cash. If this value is not provided (default), the full value
            of the strategy is allocated to securities.

    """

    def __init__(self):
        super(Rebalance, self).__init__()

    def __call__(self, target):
        if 'weights' not in target.temp:
            return True

        targets = target.temp['weights']

        # de-allocate children that are not in targets and have non-zero value
        # (open positions)
        for cname in target.children:
            # if this child is in our targets, we don't want to close it out
```

```
            if cname in targets:
                continue

            # get child and value
            c = target.children[cname]
            v = c.value
            # if non-zero and non-null, we need to close it out
            if v != 0. and not np.isnan(v):
                target.close(cname)

        # save value because it will change after each call to allocate
        # use it as base in rebalance calls
        base = target.value

        # If cash is set (it should be a value between 0-1 representing the
        # proportion of cash to keep), calculate the new 'base'
        if 'cash' in target.temp:
            base = base * (1 - target.temp['cash'])

        for item in iteritems(targets):
            target.rebalance(item[1], child=item[0], base=base)

        return True
```

[docs]

```
class RebalanceOverTime(Algo):

    """
    Similar to Rebalance but rebalances to target
    weight over n periods.

    Rebalances towards a target weight over a n periods. Splits up the weight
    delta over n periods.

    This can be useful if we want to make more conservative rebalacing
    assumptions. Some strategies can produce large swings in allocations. It
    might not be reasonable to assume that this rebalancing can occur at the
    end of one specific period. Therefore, this algo can be used to simulate
    rebalancing over n periods.

    This has typically been used in monthly strategies where we want to spread
    out the rebalancing over 5 or 10 days.

    Note:
        This Algo will require the run_always wrapper in the above case. For
        example, the RunMonthly will return True on the first day, and
        RebalanceOverTime will be 'armed'. However, RunMonthly will return
        False the rest days of the month. Therefore, we must specify that we
        want to always run this algo.
```

```
    Args:
        * n (int): number of periods over which rebalancing takes place.

    Requires:
        * weights

    """

    def __init__(self, n=10):
        super(RebalanceOverTime, self).__init__()
        self.n = float(n)
        self._rb = Rebalance()
        self._weights = None
        self._days_left = None

    def __call__(self, target):
        # new weights specified - update rebalance data
        if 'weights' in target.temp:
            self._weights = target.temp['weights']
            self._days_left = self.n

        # if _weights are not None, we have some work to do
        if self._weights:
            tgt = {}
            # scale delta relative to # of periods left and set that as the new
            # target
            for t in self._weights:
                curr = target.children[t].weight if t in \
                    target.children else 0.
                dlt = (self._weights[t] - curr) / self._days_left
                tgt[t] = curr + dlt

            # mock weights and call real Rebalance
            target.temp['weights'] = tgt
            self._rb(target)

            # dec _days_left. If 0, set to None & set _weights to None
            self._days_left -= 1

            if self._days_left == 0:
                self._days_left = None
                self._weights = None

        return True



[docs]class Require(Algo):
```

```
"""
Flow control Algo.

This algo returns the value of a predicate
on an temp entry. Useful for controlling
flow.

For example, we might want to make sure we have some items selected.
We could pass a lambda function that checks the len of 'selected':

    pred=lambda x: len(x) == 0
    item='selected'

Args:
    * pred (Algo): Function that returns a Bool given the strategy. This
        is the definition of an Algo. However, this is typically used
        with a simple lambda function.
    * item (str): An item within temp.
    * if_none (bool): Result if the item required is not in temp or if it's
        value if None

"""

def __init__(self, pred, item, if_none=False):
    super(Require, self).__init__()
    self.item = item
    self.pred = pred
    self.if_none = if_none

def __call__(self, target):
    if self.item not in target.temp:
        return self.if_none

    item = target.temp[self.item]

    if item is None:
        return self.if_none

    return self.pred(item)
```