# bt Package — bt 0.2.5 documentation

## algos Module¶

A collection of Algos used to create Strategy logic.

*class* bt.algos.CapitalFlow(*amount*)[source]¶

Bases: bt.core.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

*class* bt.algos.CloseDead[source]¶

Bases: bt.core.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

*class* bt.algos.Debug(*name=None*)[source]¶

Bases: bt.core.Algo

Utility Algo that calls pdb.set_trace when triggered.

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

*class* bt.algos.LimitDeltas(*limit=0.1*)[source]¶

Bases: bt.core.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

*class* bt.algos.LimitWeights(*limit=0.1*)[source]¶

Bases: bt.core.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

*class* bt.algos.PrintDate(*name=None*)[source]¶

Bases: bt.core.Algo

This Algo simply print's the current date.

Can be useful for debugging purposes.

*class* bt.algos.PrintInfo(*fmt_string='{full_name} {now}'*)[source]¶

Bases: bt.core.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__

*class* bt.algos.PrintTempData(*name=None*)[source]¶

Bases: bt.core.Algo

This Algo prints the temp data.

Useful for debugging.

*class* bt.algos.Rebalance[source]¶

Bases: bt.core.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.

*class* bt.algos.RebalanceOverTime(*n=10*)[source]¶

Bases: bt.core.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

*class* bt.algos.Require(*pred*, *item*, *if_none=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.RunAfterDate(*date*)[source]¶

Bases: bt.core.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

*class* bt.algos.RunAfterDays(*days*)[source]¶

Bases: bt.core.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

*class* bt.algos.RunDaily[source]¶

Bases: bt.core.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.

*class* bt.algos.RunEveryNPeriods(*n*, *offset=0*)[source]¶

Bases: bt.core.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.

*class* bt.algos.RunMonthly[source]¶

Bases: bt.core.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)

*class* bt.algos.RunOnDate(**dates*)[source]¶

Bases: bt.core.Algo

Returns True on a specific set of dates.

Args:

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

*class* bt.algos.RunOnce[source]¶

Bases: bt.core.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).

*class* bt.algos.RunQuarterly[source]¶

Bases: bt.core.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)

*class* bt.algos.RunWeekly[source]¶

Bases: bt.core.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)

*class* bt.algos.RunYearly[source]¶

Bases: bt.core.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)

*class* bt.algos.SelectAll(*include_no_data=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectHasData(*lookback=<DateOffset: kwds={'months': 3}>*, *min_count=None*, *include_no_data=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectMomentum(*n*, *lookback=<DateOffset: kwds={'months': 3}>*, *lag=<DateOffset: kwds={'days': 0}>*, *sort_descending=True*, *all_or_none=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectN(*n*, *sort_descending=True*, *all_or_none=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectRandomly(*n=None*, *include_no_data=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectThese(*tickers*, *include_no_data=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.SelectWhere(*signal*, *include_no_data=False*)[source]¶

Bases: bt.core.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

*class* bt.algos.StatTotalReturn(*lookback=<DateOffset: kwds={'months': 3}>*, *lag=<DateOffset: kwds={'days': 0}>*)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighERC(*lookback=<DateOffset: kwds={'months': 3}>*, *initial_weights=None*, *risk_weights=None*, *covar_method='ledoit-wolf'*, *risk_parity_method='ccd'*, *maximum_iterations=100*, *tolerance=1e-08*, *lag=<DateOffset:*

*kwds={'days': 0}*>)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighEqually[source]¶

Bases: bt.core.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

*class* bt.algos.WeighInvVol(*lookback=<DateOffset: kwds={'months': 3}>, lag=<DateOffset: kwds={'days': 0}>*)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighMeanVar(*lookback=<DateOffset: kwds={'months': 3}>, bounds=(0.0, 1.0), covar_method='ledoit-wolf', rf=0.0, lag=<DateOffset: kwds={'days': 0}>*)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighRandomly(*bounds=(0.0, 1.0), weight_sum=1*)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighSpecified(***weights*)[source]¶

Bases: bt.core.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

*class* bt.algos.WeighTarget(*weights*)[source]¶

Bases: bt.core.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

bt.algos.run_always(*f*)[source]¶

Run always decorator to be used with Algo to ensure stack runs the decorated Algo on each pass, regardless of failures in the stack.

## backtest Module¶

Contains backtesting logic and objects.

*class* bt.backtest.Backtest(*strategy*, *data*, *name=None*, *initial_capital=1000000.0*, *commissions=None*, *integer_positions=True*, *progress_bar=True*)[source]¶

Bases: object

A Backtest combines a Strategy with data to produce a Result.

A backtest is basically testing a strategy over a data set.

Note:

The Strategy will be deepcopied so it is re-usable in other backtests. To access the backtested strategy, simply access the strategy attribute.

Args:

- strategy (Strategy, Node, StrategyBase): The Strategy to be tested.
- data (DataFrame): DataFrame containing data used in backtest. This will be the Strategy's "universe".
- name (str): Backtest name - defaults to strategy name
- initial_capital (float): Initial amount of capital passed to Strategy.
- commissions (fn(quantity, price)): The commission function

  to be used. Ex: commissions=lambda q, p: max(1, abs(q) * 0.01) * progress_bar (Bool): Display progress bar while running backtest

Attributes:

- strategy (Strategy): The Backtest's Strategy. This will be a deepcopy of the Strategy that was passed in.
- data (DataFrame): Data passed in
- dates (DateTimeIndex): Data's index

- initial_capital (float): Initial capital
- name (str): Backtest name
- stats (ffn.PerformanceStats): Performance statistics
- has_run (bool): Run flag
- weights (DataFrame): Weights of each component over time
- security_weights (DataFrame): Weights of each security as a percentage of the whole portfolio over time

herfindahl_index¶

Calculate Herfindahl-Hirschman Index (HHI) for the portfolio. For each given day, HHI is defined as a sum of squared weights of securities in a portfolio; and varies from 1/N to 1. Value of 1/N would correspond to an equally weighted portfolio and value of 1 corresponds to an extreme case when all amount is invested in a single asset.

1 / HHI is often considered as "an effective number of assets" in a given portfolio

positions¶

DataFrame of each component's position over time

run()[source]¶

Runs the Backtest.

security_weights¶

DataFrame containing weights of each security as a percentage of the whole portfolio over time

turnover¶

Calculate the turnover for the backtest.

This function will calculate the turnover for the strategy. Turnover is defined as the lesser of positive or negative outlays divided by NAV

weights¶

DataFrame of each component's weight over time

*class* bt.backtest.RandomBenchmarkResult(**backtests*)[source]¶

Bases: bt.backtest.Result

RandomBenchmarkResult expands on Result to add methods specific to random strategy benchmarking.

Args:

- backtests (list): List of backtests

Attributes:

- base_name (str): Name of backtest being benchmarked
- r_stats (Result): Stats for random strategies
- b_stats (Result): Stats for benchmarked strategy

plot_histogram(*statistic='monthly_sharpe', figsize=(15, 5), title=None, bins=20, **kwargs*)[source]¶

Plots the distribution of a given statistic. The histogram represents the distribution of the random strategies' statistic and the vertical line is the value of the benchmarked strategy's statistic.

This helps you determine if your strategy is statistically 'better' than the random versions.

Args:

- statistic (str): Statistic - any numeric statistic in Result is valid.
- figsize ((x, y)): Figure size
- title (str): Chart title

- bins (int): Number of bins
- kwargs (dict): Passed to pandas hist function.

*class* bt.backtest.Result(**backtests*)[source]¶

Bases: ffn.core.GroupStats

Based on ffn's GroupStats with a few extra helper methods.

Args:

- backtests (list): List of backtests

Attributes:

- backtest_list (list): List of bactests in the same order as provided
- backtests (dict): Dict of backtests by name

display_monthly_returns(*backtest=0*)[source]¶

Display monthly returns for a specific backtest.

Args:

- backtest (str, int): Backtest. Can be either a index (int) or the name (str)

get_transactions(*strategy_name=None*)[source]¶

Helper function that returns the transactions in the following format:

dt, security | quantity, price

The result is a MultiIndex DataFrame.

Args:

- strategy_name (str): If none, it will take the first backtest's strategy (self.backtest_list[0].name)

plot_histogram(*backtest=0*, ***kwds*)[source]¶

Plots the return histogram of a given backtest over time.

Args:

- backtest (str, int): Backtest. Can be either a index (int) or the name (str)
- kwds (dict): Keywords passed to plot_histogram

plot_security_weights(*backtest=0*, *filter=None*, *figsize=(15, 5)*, ***kwds*)[source]¶

Plots the security weights of a given backtest over time.

Args:

- backtest (str, int): Backtest. Can be either a index (int) or the name (str)
- filter (list, str): filter columns for specific columns. Filter is simply passed as is to DataFrame[filter], so use something that makes sense with a DataFrame.
- figsize ((width, height)): figure size
- kwds (dict): Keywords passed to plot

plot_weights(*backtest=0*, *filter=None*, *figsize=(15, 5)*, ***kwds*)[source]¶

Plots the weights of a given backtest over time.

Args:

- backtest (str, int): Backtest. Can be either a index (int) or the name (str)
- filter (list, str): filter columns for specific columns. Filter

is simply passed as is to DataFrame[filter], so use something that makes sense with a DataFrame.

- figsize ((width, height)): figure size
- kwds (dict): Keywords passed to plot

`bt.backtest.benchmark_random`(*backtest*, *random_strategy*, *nsim=100*)[source]¶

Given a backtest and a random strategy, compare backtest to a number of random portfolios.

The idea here is to benchmark your strategy vs a bunch of random strategies that have a similar structure but execute some part of the logic randomly - basically you are trying to determine if your strategy has any merit - does it beat randomly picking weight? Or randomly picking the selected securities?

Args:

- backtest (Backtest): A backtest you want to benchmark
- random_strategy (Strategy): A strategy you want to benchmark
  against. The strategy should have a random component to emulate skilless behavior.
- nsim (int): number of random strategies to create.

Returns:

RandomBenchmarkResult

`bt.backtest.run`(**backtests*)[source]¶

Runs a series of backtests and returns a Result object containing the results of the backtests.

Args:

- backtest (*list): List of backtests.

Returns:

Result

## core Module¶

Contains the core building blocks of the framework.

*class* `bt.core.Algo`(*name=None*)[source]¶

Bases: `object`

Algos are used to modularize strategy logic so that strategy logic becomes modular, composable, more testable and less error prone. Basically, the Algo should follow the unix philosophy - do one thing well.

In practice, algos are simply a function that receives one argument, the Strategy (refered to as target) and are expected to return a bool.

When some state preservation is necessary between calls, the Algo object can be used (this object). The __call___ method should be implemented and logic defined therein to mimic a function call. A simple function may also be used if no state preservation is neceesary.

Args:

- name (str): Algo name

`name`¶

Algo name.

*class* `bt.core.AlgoStack`(**algos*)[source]¶

Bases: `bt.core.Algo`

An AlgoStack derives from Algo runs multiple Algos until a failure is encountered.

The purpose of an AlgoStack is to group a logic set of Algos together. Each Algo in the stack is run. Execution stops if one Algo returns False.

Args:

- algos (list): List of algos.

*class* bt.core.Node(*name*, *parent=None*, *children=None*)[source]¶

Bases: object

The Node is the main building block in bt's tree structure design. Both StrategyBase and SecurityBase inherit Node. It contains the core functionality of a tree node.

Args:

- name (str): The Node name
- parent (Node): The parent Node
- children (dict, list): A collection of children. If dict, the format is {name: child}, if list then list of children.

Attributes:

- name (str): Node name
- parent (Node): Node parent
- root (Node): Root node of the tree (topmost node)
- children (dict): Node's children
- now (datetime): Used when backtesting to store current date
- stale (bool): Flag used to determine if Node is stale and need updating
- prices (TimeSeries): Prices of the Node. Prices for a security will be the security's price, for a strategy it will be an index that reflects the value of the strategy over time.
- price (float): last price
- value (float): last value
- weight (float): weight in parent
- full_name (str): Name including parents' names
- members (list): Current Node + node's children

adjust(*amount*, *update=True*, *isflow=True*)[source]¶

Adjust Node value by amount.

allocate(*amount*, *update=True*)[source]¶

Allocate capital to Node.

full_name¶

members¶

Node members. Members include current node as well as Node's children.

price¶

Current price of the Node

prices¶

A TimeSeries of the Node's price.

setup(*dates*)[source]¶

Setup method used to initialize a Node with a set of dates.

update(*date*, *data=None*, *inow=None*)[source]¶

Update Node with latest date, and optionally some data.

use_integer_positions(*integer_positions*)[source]¶

Set indicator to use (or not) integer positions for a given strategy or security.

By default all positions in number of stocks should be integer. However this may lead to unexpected results when working with adjusted prices of stocks. Because of series of reverse splits of stocks, the adjusted prices back in time might be high. Thus rounding of desired amount of stocks to buy may lead to having 0, and thus ignoring this stock from backtesting.

`value`¶

Current value of the Node

`weight`¶

Current weight of the Node (with respect to the parent).

*class* `bt.core.SecurityBase`(*name*, *multiplier=1*)[source]¶

Bases: `bt.core.Node`

Security Node. Used to define a security within a tree. A Security's has no children. It simply models an asset that can be bought or sold.

Args:

- name (str): Security name
- multiplier (float): security multiplier - typically used for derivatives.

Attributes:

- name (str): Security name
- parent (Security): Security parent
- root (Security): Root node of the tree (topmost node)
- now (datetime): Used when backtesting to store current date
- stale (bool): Flag used to determine if Security is stale and need updating
- prices (TimeSeries): Security prices.
- price (float): last price
- outlays (TimeSeries): Series of outlays. Positive outlays mean capital was allocated to security and security consumed that amount. Negative outlays are the opposite. This can be useful for calculating turnover at the strategy level.
- value (float): last value - basically position * price * multiplier
- weight (float): weight in parent
- full_name (str): Name including parents' names
- members (list): Current Security + strategy's children
- position (float): Current position (quantity).

`allocate`(*amount*, *update=True*)[source]¶

This allocates capital to the Security. This is the method used to buy/sell the security.

A given amount of shares will be determined on the current price, a commisison will be calculated based on the parent's commission fn, and any remaining capital will be passed back up to parent as an adjustment.

Args:

- amount (float): Amount of adjustment.
- update (bool): Force update?

`commission`(*q*, *p*)[source]¶

Calculates the commission (transaction fee) based on quantity and price. Uses the parent's commission_fn.

Args:

- q (float): quantity
- p (float): price

`multiplier` = *0.0*¶

`outlay`(*q*)[source]¶

Determines the complete cash outlay (including commission) necessary given a quantity q. Second returning parameter is a commission itself.

Args:

- q (float): quantity

`outlays`¶

TimeSeries of outlays. Positive outlays (buys) mean this security received and consumed capital (capital was allocated to it). Negative outlays are the opposite (the security close/sold, and returned capital to parent).

`position`¶

Current position

`positions`¶

TimeSeries of positions.

`price`¶

Current price.

`prices`¶

TimeSeries of prices.

`run`()[source]¶

Does nothing - securities have nothing to do on run.

`setup`(*universe*)[source]¶

Setup Security with universe. Speeds up future runs.

Args:

- universe (DataFrame): DataFrame of prices with security's name as one of the columns.

`update`(*date*, *data=None*, *inow=None*)[source]¶

Update security with a given date and optionally, some data. This will update price, value, weight, etc.

`values`¶

TimeSeries of values.

*class* `bt.core.Strategy`(*name*, *algos=[]*, *children=None*)[source]¶

Bases: `bt.core.StrategyBase`

Strategy expands on the StrategyBase and incorporates Algos.

Basically, a Strategy is built by passing in a set of algos. These algos will be placed in an Algo stack and the run function will call the stack.

Furthermore, two class attributes are created to pass data between algos. perm for permanent data, temp for temporary data.

Args:

- name (str): Strategy name
- algos (list): List of Algos to be passed into an AlgoStack
- children (dict, list): Children - useful when you want to create strategies of strategies

Attributes:

- stack (AlgoStack): The stack

- temp (dict): A dict containing temporary data - cleared on each call to run. This can be used to pass info to other algos.
- perm (dict): Permanent data used to pass info from one algo to another. Not cleared on each pass.

`run`()[source]¶

*class* `bt.core.StrategyBase`(*name*, *children=None*, *parent=None*)[source]¶

Bases: `bt.core.Node`

Strategy Node. Used to define strategy logic within a tree. A Strategy's role is to allocate capital to it's children based on a function.

Args:

- name (str): Strategy name
- children (dict, list): A collection of children. If dict, the format is {name: child}, if list then list of children. Children can be any type of Node.
- parent (Node): The parent Node

Attributes:

- name (str): Strategy name
- parent (Strategy): Strategy parent
- root (Strategy): Root node of the tree (topmost node)
- children (dict): Strategy's children
- now (datetime): Used when backtesting to store current date
- stale (bool): Flag used to determine if Strategy is stale and need updating
- prices (TimeSeries): Prices of the Strategy - basically an index that reflects the value of the strategy over time.
- outlays (DataFrame): Outlays for each SecurityBase child
- price (float): last price
- value (float): last value
- weight (float): weight in parent
- full_name (str): Name including parents' names
- members (list): Current Strategy + strategy's children
- securities (list): List of strategy children that are of type SecurityBase
- commission_fn (fn(quantity, price)): A function used to determine the commission (transaction fee) amount. Could be used to model slippage (implementation shortfall). Note that often fees are symmetric for buy and sell and absolute value of quantity should be used for calculation.
- capital (float): Capital amount in Strategy - cash
- universe (DataFrame): Data universe available at the current time. Universe contains the data passed in when creating a Backtest. Use this data to determine strategy logic.

`adjust`(*amount*, *update=True*, *flow=True*, *fee=0.0*)[source]¶

Adjust capital - used to inject capital to a Strategy. This injection of capital will have no effect on the children.

Args:

- amount (float): Amount to adjust by.

- update (bool): Force update?
- flow (bool): Is this adjustment a flow? A flow will not have an impact on the performance (price index). Example of flows are simply capital injections (say a monthly contribution to a portfolio). This should not be reflected in the returns. A non-flow (flow=False) does impact performance. A good example of this is a commission, or a dividend.

`allocate`(*amount*, *child=None*, *update=True*)[source]¶

Allocate capital to Strategy. By default, capital is allocated recursively down the children, proportionally to the children's weights. If a child is specified, capital will be allocated to that specific child.

Allocation also have a side-effect. They will deduct the same amount from the parent's "account" to offset the allocation. If there is remaining capital after allocation, it will remain in Strategy.

Args:

- amount (float): Amount to allocate.
- child (str): If specified, allocation will be directed to child only. Specified by name.
- update (bool): Force update.

`bankrupt` = *False*¶

`capital`¶

Current capital - amount of unallocated capital left in strategy.

`cash`¶

TimeSeries of unallocated capital.

`close`(*child*)[source]¶

Close a child position - alias for rebalance(0, child). This will also flatten (close out all) the child's children.

Args:

- child (str): Child, specified by name.

`fees`¶

TimeSeries of fees.

`flatten`()[source]¶

Close all child positions.

`outlays`¶

Returns a DataFrame of outlays for each child SecurityBase

`positions`¶

TimeSeries of positions.

`price`¶

Current price.

`prices`¶

TimeSeries of prices.

`rebalance`(*weight*, *child*, *base=nan*, *update=True*)[source]¶

Rebalance a child to a given weight.

This is a helper method to simplify code logic. This method is used when we want to se the weight of a particular child to a set amount. It is similar to allocate, but it calculates the appropriate allocation based on the current weight.

Args:

- weight (float): The target weight. Usually between -1.0 and 1.0.
- child (str): child to allocate to - specified by name.
- base (float): If specified, this is the base amount all weight

delta calculations will be based off of. This is useful when we determine a set of weights and want to rebalance each child given these new weights. However, as we iterate through each child and call this method, the base (which is by default the current value) will change. Therefore, we can set this base to the original value before the iteration to ensure the proper allocations are made.

- update (bool): Force update?

`run`()[source]¶

This is the main logic method. Override this method to provide some algorithm to execute on each date change. This method is called by backtester.

`securities`¶

Returns a list of children that are of type SecurityBase

`set_commissions`(*fn*)[source]¶

Set commission (transaction fee) function.

Args:

fn (fn(quantity, price)): Function used to determine commission amount.

`setup`(*universe*)[source]¶

Setup strategy with universe. This will speed up future calculations and updates.

`universe`¶

Data universe available at the current time. Universe contains the data passed in when creating a Backtest. Use this data to determine strategy logic.

`update`(*date*, *data=None*, *inow=None*)[source]¶

Update strategy. Updates prices, values, weight, etc.

`values`¶

TimeSeries of values.