

Module `backtesting.backtesting`

Core framework data structures. Objects from this module can also be imported from the top-level module directly, e.g.

```
from backtesting import Backtest, Strategy
```

Classes

```
class Backtest (data,
               strategy,
               *,
               cash=10000,
               spread=0.0,
               commission=0.0,
               margin=1.0,
               trade_on_close=False,
               hedging=False,
               exclusive_orders=False,
               finalize_trades=False)
```

Backtest a particular (parameterized) strategy on particular data.

Initialize a backtest. Requires data and a strategy to test. After initialization, you can call method [Backtest.run\(\)](#) to run a backtest instance, or [Backtest.optimize\(\)](#) to optimize it.

`data` is a `pd.DataFrame` with columns: Open, High, Low, Close, and (optionally) Volume. If any columns are missing, set them to what you have available, e.g.

```
df['Open'] = df['High'] = df['Low'] = df['Close']
```

The passed data frame can contain additional columns that can be used by the strategy (e.g. sentiment info). `DataFrame` index can be either a datetime index (timestamps) or a monotonic range index (i.e. a sequence of periods).

`strategy` is a [Strategy](#) subclass (not an instance).

`cash` is the initial cash to start with.

`spread` is the the constant bid-ask spread rate (relative to the price). E.g. set it to `0.0002` for commission-less forex trading where the average spread is roughly 0.2‰ of the asking price.

`commission` is the commission rate. E.g. if your broker's commission is 1% of order value, set commission to `0.01`. The commission is applied twice: at trade entry and at trade exit. Besides one single floating value, `commission` can also be a tuple of floating values (`fixed, relative`). E.g. set it to `(100, .01)` if your broker charges minimum \$100 + 1%. Additionally, `commission` can be a callable `func(order_size: int, price: float) -> float` (note, order size is negative for short orders), which can be used to model more complex commission structures. Negative commission values are interpreted as market-maker's rebates.

Note

Before v0.4.0, the commission was only applied once, like `spread` is now. If you want to keep the old behavior, simply set `spread` instead.

Note

With nonzero `commission`, long and short orders will be placed at an adjusted price that is slightly higher or lower (respectively) than the current price. See e.g. [#153](#), [#538](#), [#633](#).

`margin` is the required margin (ratio) of a leveraged account. No difference is made between initial and maintenance margins. To run the backtest using e.g. 50:1 leverage that your broker allows, set margin to `0.02` ($1 / \text{leverage}$).

If `trade_on_close` is `True`, market orders will be filled with respect to the current bar's closing price instead of the next bar's open.

If `hedging` is `True`, allow trades in both directions simultaneously. If `False`, the opposite-facing orders first close existing trades in a [FIFO](#) manner.

If `exclusive_orders` is `True`, each new order auto-closes the previous trade/position, making at most a single trade (long or short) in effect at each time.

If `finalize_trades` is `True`, the trades that are still [active and ongoing](#) at the end of the backtest will be closed on the last bar and will contribute to the computed backtest statistics.

Tip: Fractional trading

See also [FractionalBacktest](#) if you want to trade fractional units (of e.g. bitcoin).

Subclasses

- [FractionalBacktest](#)

Methods

```
def optimize(self,
*,
maximize='SQN',
method='grid',
max_tries=None,
constraint=None,
return_heatmap=False,
return_optimization=False,
random_state=None,
**kwargs)
```

Optimize strategy parameters to an optimal combination. Returns result `pd.Series` of the best run.

`maximize` is a string key from the [Backtest.run\(\)](#)-returned results series, or a function that accepts this series object and returns a number; the higher the better. By default, the method maximizes Van Tharp's [System Quality Number](#).

`method` is the optimization method. Currently two methods are supported:

- `"grid"` which does an exhaustive (or randomized) search over the cartesian product of parameter combinations, and
- `"sambo"` which finds close-to-optimal strategy parameters using [model-based optimization](#), making at most `max_tries` evaluations.

`max_tries` is the maximal number of strategy runs to perform. If `method="grid"`, this results in randomized grid search. If `max_tries` is a floating value between (0, 1], this sets the number of runs to approximately that fraction of full grid space. Alternatively, if integer, it denotes the absolute maximum number of evaluations. If unspecified (default), grid search is exhaustive, whereas for `method="sambo"`, `max_tries` is set to 200.

`constraint` is a function that accepts a dict-like object of parameters (with values) and returns `True` when the combination is admissible to test with. By default, any parameters combination is considered admissible.

If `return_heatmap` is `True`, besides returning the result series, an additional `pd.Series` is returned with a multiindex of all admissible parameter combinations, which can be further inspected or projected onto 2D to plot a heatmap (see [plot_heatmaps\(\)](#)).

If `return_optimization` is `True` and `method = 'sambo'`, in addition to result series (and maybe heatmap), return raw [scipy.optimize.OptimizeResult](#) for further inspection, e.g. with [SAMBO's plotting tools](#).

If you want reproducible optimization results, set `random_state` to a fixed integer random seed.

Additional keyword arguments represent strategy arguments with list-like collections of possible values. For example, the following code finds and returns the "best" of the 7 admissible (of the 9 possible) parameter combinations:

```
best_stats = backtest.optimize(sma1=[5, 10, 15], sma2=[10, 20, 40], constraint=lambda p: p.sma1 <
p.sma2)
```

```
def plot(self,
*,
results=None,
filename=None,
plot_width=None,
plot_equity=True,
plot_return=False,
plot_pl=True,
plot_volume=True,
plot_drawdown=False,
plot_trades=True,
smooth_equity=False,
relative_equity=True,
superimpose=True,
resample=True,
reverse_indicators=False,
show_legend=True,
open_browser=True)
```

Plot the progression of the last backtest run.

If `results` is provided, it should be a particular result `pd.Series` such as returned by [Backtest.run\(\)](#) or [Backtest.optimize\(\)](#), otherwise the last run's results are used.

`filename` is the path to save the interactive HTML plot to. By default, a strategy/parameter-dependent file is created in the current working directory.

`plot_width` is the width of the plot in pixels. If `None` (default), the plot is made to span 100% of browser width. The height is currently non-adjustable.

If `plot_equity` is `True`, the resulting plot will contain an equity (initial cash plus assets) graph section. This is the same as `plot_return` plus initial 100%.

If `plot_return` is `True`, the resulting plot will contain a cumulative return graph section. This is the same as `plot_equity` minus initial 100%.

If `plot_pl` is `True`, the resulting plot will contain a profit/loss (P/L) indicator section.

If `plot_volume` is `True`, the resulting plot will contain a trade volume section.

If `plot_drawdown` is `True`, the resulting plot will contain a separate drawdown graph section.

If `plot_trades` is `True`, the stretches between trade entries and trade exits are marked by hash-marked tractor beams.

If `smooth_equity` is `True`, the equity graph will be interpolated between fixed points at trade closing times, unaffected by any interim asset volatility.

If `relative_equity` is `True`, scale and label equity graph axis with return percent, not absolute cash-equivalent values.

If `superimpose` is `True`, superimpose larger-timeframe candlesticks over the original candlestick chart. Default downsampling rule is: monthly for daily data, daily for hourly data, hourly for minute data, and minute for (sub-)second data. `superimpose` can also be a valid [Pandas offset string](#), such as `'5T'` or `'5min'`, in which case this frequency will be used to superimpose. Note, this only works for data with a datetime index.

If `resample` is `True`, the OHLC data is resampled in a way that makes the upper number of candles for Bokeh to plot limited to 10_000. This may, in situations of overabundant data, improve plot's interactive performance and avoid browser's `Javascript Error: Maximum call stack size exceeded` or similar. Equity & dropdown curves and individual trades data is, likewise, [reasonably aggregated](#). `resample` can also be a [Pandas offset string](#), such as `'5T'` or `'5min'`, in which case this frequency will be used to resample, overriding above numeric limitation. Note, all this only works for data with a datetime index.

If `reverse_indicators` is `True`, the indicators below the OHLC chart are plotted in reverse order of declaration.

If `show_legend` is `True`, the resulting plot graphs will contain labeled legends.

If `open_browser` is `True`, the resulting `filename` will be opened in the default web browser.

```
def run(self, **kwargs)
```

Run the backtest. Returns `pd.Series` with results and statistics.

Keyword arguments are interpreted as strategy parameters.

```
>>> Backtest(GOOG, SmaCross).run() Start 2004-08-19 00:00:00 End 2013-03-01 00:00:00 Duration 3116 days
00:00:00 Exposure Time [%] 96.74115 Equity Final [$] 51422.99 Equity Peak [$] 75787.44 Return [%]
414.2299 Buy & Hold Return [%] 703.45824 Return (Ann.) [%] 21.18026 Volatility (Ann.) [%] 36.49391 CAGR
[%] 14.15984 Sharpe Ratio 0.58038 Sortino Ratio 1.08479 Calmar Ratio 0.44144 Alpha [%] 394.37391 Beta
0.03803 Max. Drawdown [%] -47.98013 Avg. Drawdown [%] -5.92585 Max. Drawdown Duration 584 days 00:00:00
Avg. Drawdown Duration 41 days 00:00:00 # Trades 66 Win Rate [%] 46.9697 Best Trade [%] 53.59595 Worst
Trade [%] -18.39887 Avg. Trade [%] 2.53172 Max. Trade Duration 183 days 00:00:00 Avg. Trade Duration 46
days 00:00:00 Profit Factor 2.16795 Expectancy [%] 3.27481 SQN 1.07662 Kelly Criterion 0.15187 _strategy
SmaCross _equity_curve Eq... _trades Size EntryB... dtype: object
```

Warning

You may obtain different results for different strategy parameters. E.g. if you use 50- and 200-bar SMA, the trading simulation will begin on bar 201. The actual length of delay is equal to the lookback period of the [Strategy.I\(\)](#) indicator which lags the most. Obviously, this can affect results.

```
class Order
```

Place new orders through [Strategy.buy\(\)](#) and [Strategy.sell\(\)](#). Query existing orders through [Strategy.orders](#).

When an order is executed or [filled](#), it results in a [Trade](#).

If you wish to modify aspects of a placed but not yet filled order, cancel it and place a new one instead.

All placed orders are [Good 'Til Canceled](#).

Instance variables

```
prop is_contingent
```

True for [contingent](#) orders, i.e. [OCO](#) stop-loss and take-profit bracket orders placed upon an active trade. Remaining contingent orders are canceled when their parent [Trade](#) is closed.

You can modify contingent orders through [Trade.sl](#) and [Trade.tp](#).

```
prop is_long
```

True if the order is long (order size is positive).

```
prop is_short
```

True if the order is short (order size is negative).

```
prop limit
```

Order limit price for [limit orders](#), or None for [market orders](#), which are filled at next available price.

```
prop size
```

Order size (negative for short orders).

If size is a value between 0 and 1, it is interpreted as a fraction of current available liquidity (cash plus [Position.pl](#) minus used margin). A value greater than or equal to 1 indicates an absolute number of units.

```
prop sl
```

A stop-loss price at which, if set, a new contingent stop-market order will be placed upon the [Trade](#) following this order's execution. See also [Trade.sl](#).

```
prop stop
```

Order stop price for [stop-limit/stop-market](#) order, otherwise None if no stop was set, or the stop price has already been hit.

```
prop tag
```

Arbitrary value (such as a string) which, if set, enables tracking of this order and the associated [Trade](#) (see [Trade.tag](#)).

`prop tp`

A take-profit price at which, if set, a new contingent limit order will be placed upon the [Trade](#) following this order's execution. See also [Trade.tp](#).

Methods

```
def cancel(self)
```

Cancel the order.

```
class Position
```

Currently held asset position, available as [Strategy.position](#) within [Strategy.next\(\)](#). Can be used in boolean contexts, e.g.

```
if self.position: ... # we have a position, either long or short
```

Instance variables

```
prop is_long
```

True if the position is long (position size is positive).

```
prop is_short
```

True if the position is short (position size is negative).

```
prop pl
```

Profit (positive) or loss (negative) of the current position in cash units.

```
prop pl_pct
```

Profit (positive) or loss (negative) of the current position in percent.

```
prop size
```

Position size in units of asset. Negative if position is short.

Methods

```
def close(self, portion=1.0)
```

Close portion of position by closing `portion` of each active trade. See [Trade.close\(\)](#).

```
class Strategy
```

A trading strategy base class. Extend this class and override methods [Strategy.init\(\)](#) and [Strategy.next\(\)](#) to define your own strategy.

Subclasses

- [SignalStrategy](#)
- [TrailingStrategy](#)

Instance variables

`prop closed_trades`

List of settled trades (see [Trade](#)).

`prop data`

Price data, roughly as passed into [Backtest](#), but with two significant exceptions:

- `data` is *not* a `DataFrame`, but a custom structure that serves customized numpy arrays for reasons of performance and convenience. Besides OHLCV columns, `.index` and `length`, it offers `.pip` property, the smallest price unit of change.
- Within [Strategy.init\(\)](#), `data` arrays are available in full length, as passed into [Backtest](#) (for precomputing indicators and such). However, within [Strategy.next\(\)](#), `data` arrays are only as long as the current iteration, simulating gradual price point revelation. In each call of [Strategy.next\(\)](#) (iteratively called by [Backtest](#) internally), the last array value (e.g. `data.Close[-1]`) is always the *most recent* value.
- If you need data arrays (e.g. `data.Close`) to be indexed **Pandas series**, you can call their `.s` accessor (e.g. `data.Close.s`). If you need the whole of data as a **DataFrame**, use `.df` accessor (i.e. `data.df`).

`prop equity`

Current account equity (cash plus assets).

`prop orders`

List of orders (see [Order](#)) waiting for execution.

`prop position`

Instance of [Position](#).

`prop trades`

List of active trades (see [Trade](#)).

Methods

```
def I(self, func, *args, name=None, plot=True, overlay=None, color=None, scatter=False, **kwargs)
```

Declare an indicator. An indicator is just an array of values (or a tuple of such arrays in case of, e.g., MACD indicator), but one that is revealed gradually in [Strategy.next\(\)](#) much like [Strategy.data](#) is. Returns `np.ndarray` of indicator values.

`func` is a function that returns the indicator array(s) of same length as [Strategy.data](#).

In the plot legend, the indicator is labeled with function name, unless `name` overrides it. If `func` returns a tuple of arrays, `name` can be a sequence of strings, and its size must agree with the number of arrays returned.

If `plot` is `True`, the indicator is plotted on the resulting [Backtest.plot\(\)](#).

If `overlay` is `True`, the indicator is plotted overlaying the price candlestick chart (suitable e.g. for moving averages). If `False`, the indicator is plotted standalone below the candlestick chart. By default, a heuristic is used which decides correctly most of the time.

`color` can be string hex RGB triplet or X11 color name. By default, the next available color is assigned.

If `scatter` is `True`, the plotted indicator marker will be a circle instead of a connected line segment (default).

Additional `*args` and `**kwargs` are passed to `func` and can be used for parameters.

For example, using simple moving average function from TA-Lib:

```
def init(): self.sma = self.I(ta.SMA, self.data.Close, self.n_sma)
```

Warning

Rolling indicators may front-pad warm-up values with NaNs. In this case, the **backtest will only begin on the first bar when all declared indicators have non-NaN values** (e.g. bar 201 for a strategy that uses a 200-bar MA). This can affect results.

```
def buy(self, *, size=.9999, limit=None, stop=None, sl=None, tp=None, tag=None)
```

Place a new long order and return it. For explanation of parameters, see [Order](#) and its properties. Unless you're running `Backtest(..., trade_on_close=True)`, market orders are filled on next bar's open, whereas other order types (limit, stop-limit, stop-market) are filled when the respective conditions are met.

See [Position.close\(\)](#) and [Trade.close\(\)](#) for closing existing positions.

See also [Strategy.sell\(\)](#).

```
def init(self)
```

Initialize the strategy. Override this method. Declare indicators (with [Strategy.I\(\)](#)). Precompute what needs to be precomputed or can be precomputed in a vectorized fashion before the strategy starts.

If you extend composable strategies from [backtesting.lib](#), make sure to call:

```
super().init()
```

```
def next(self)
```

Main strategy runtime method, called as each new [Strategy.data](#) instance (row; full candlestick bar) becomes available. This is the main method where strategy decisions upon data precomputed in [Strategy.init\(\)](#) take place.

If you extend composable strategies from [backtesting.lib](#), make sure to call:

```
super().next()
```

```
def sell(self, *, size=.9999, limit=None, stop=None, sl=None, tp=None, tag=None)
```

Place a new short order and return it. For explanation of parameters, see [Order](#) and its properties.

Caution

Keep in mind that `self.sell(size=.1)` doesn't close existing `self.buy(size=.1)` trade unless:

- the backtest was run with `exclusive_orders=True`,
- the underlying asset price is equal in both cases and the backtest was run with `spread = commission = 0`.

Use [Trade.close\(\)](#) or [Position.close\(\)](#) to explicitly exit trades.

See also [Strategy.buy\(\)](#).

Note

If you merely want to close an existing long position, use [Position.close\(\)](#) or [Trade.close\(\)](#).

```
class Trade
```

When an [Order](#) is filled, it results in an active [Trade](#). Find active trades in [Strategy.trades](#) and closed, settled trades in [Strategy.closed_trades](#).

Instance variables

```
prop entry_bar
```

Candlestick bar index of when the trade was entered.

```
prop entry_price
```

Trade entry price.

```
prop entry_time
```

Datetime of when the trade was entered.

```
prop exit_bar
```

Candlestick bar index of when the trade was exited (or None if the trade is still active).

```
prop exit_price
```

Trade exit price (or None if the trade is still active).

```
prop exit_time
```

Datetime of when the trade was exited.

```
prop is_long
```

True if the trade is long (trade size is positive).

```
prop is_short
```

True if the trade is short (trade size is negative).

```
prop pl
```

Trade profit (positive) or loss (negative) in cash units. Commissions are reflected only after the Trade is closed.

```
prop pl_pct
```

Trade profit (positive) or loss (negative) in percent.

```
prop size
```

Trade size (volume; negative for short trades).

```
prop sl
```

Stop-loss price at which to close the trade.

This variable is writable. By assigning it a new price value, you create or modify the existing SL order. By assigning it `None`, you cancel it.

```
prop tag
```

A tag value inherited from the [Order](#) that opened this trade.

This can be used to track trades and apply conditional logic / subgroup analysis.

See also [Order.tag](#).

```
prop tp
```

Take-profit price at which to close the trade.

This property is writable. By assigning it a new price value, you create or modify the existing TP order. By assigning it `None`, you cancel it.

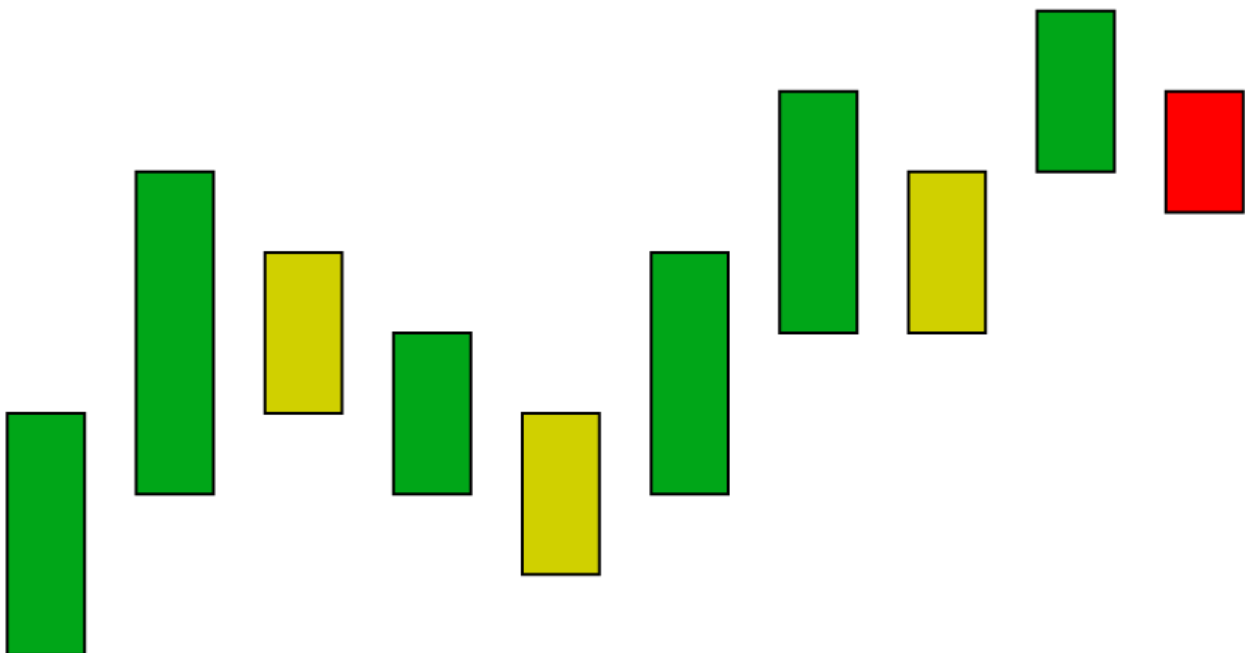
```
prop value
```

Trade total value in cash (volume × price).

Methods

```
def close(self, portion=1.0)
```

Place new [Order](#) to close `portion` of the trade at next market price.



Super-module

- [backtesting](#)

Classes

Backtest

- [optimize](#)
- [plot](#)
- [run](#)

Order

- [cancel](#)
- [is_contingent](#)
- [is_long](#)
- [is_short](#)
- [limit](#)
- [size](#)
- [sl](#)
- [stop](#)
- [tag](#)
- [tp](#)

Position

- [close](#)
- [is_long](#)
- [is_short](#)
- [pl](#)
- [pl_pct](#)
- [size](#)

Strategy

- [I](#)
- [buy](#)
- [closed_trades](#)
- [data](#)
- [equity](#)
- [init](#)
- [next](#)
- [orders](#)
- [position](#)
- [sell](#)
- [trades](#)

Trade

- [close](#)
- [entry_bar](#)
- [entry_price](#)
- [entry_time](#)
- [exit_bar](#)
- [exit_price](#)
- [exit_time](#)
- [is_long](#)
- [is_short](#)
- [pl](#)
- [pl_pct](#)
- [size](#)
- [sl](#)
- [tag](#)
- [tp](#)
- [value](#)

[backtesting 0.6.5](#) ■

Generated by [pdoc 0.11.6](#).