Module `backtesting.lib`

Collection of common building blocks, helper auxiliary functions and composable strategy classes for reuse.

Intended for simple missing-link procedures, not reinventing of better-suited, state-of-the-art, fast libraries, such as TA-Lib, Tulipy, PyAlgoTrade, NumPy, SciPy …

Please raise ideas for additions to this collection on the issue tracker.

Global variables

`var OHLCV_AGG`

Dictionary of rules for aggregating resampled OHLCV data frames, e.g.

```
df.resample('4H', label='right').agg(OHLCV_AGG).dropna()
```

`var TRADES_AGG`

Dictionary of rules for aggregating resampled trades data, e.g.

```
stats['_trades'].resample('1D', on='ExitTime', label='right').agg(TRADES_AGG)
```

Functions

`def barssince(condition, default=inf)`

Return the number of bars since `condition` sequence was last `True`, or if never, return `default`.

```
>>> barssince(self.data.Close > self.data.Open) 3
```

`def compute_stats(*, stats, data, trades=None, risk_free_rate=0.0)`

(Re-)compute strategy performance metrics.

`stats` is the statistics series as returned by `Backtest.run()`. `data` is OHLC data as passed to the `Backtest` the `stats` were obtained in. `trades` can be a dataframe subset of `stats._trades` (e.g. only long trades). You can also tune `risk_free_rate`, used in calculation of Sharpe and Sortino ratios.

```
>>> stats = Backtest(GOOG, MyStrategy).run() >>> only_long_trades = stats._trades[stats._trades.Size >
0] >>> long_stats = compute_stats(stats=stats, trades=only_long_trades, ... data=GOOG,
risk_free_rate=.02)
```

`def cross(series1, series2)`

Return `True` if `series1` and `series2` just crossed (above or below) each other.

```
>>> cross(self.data.Close, self.sma) True
```

`def crossover(series1, series2)`

Return `True` if `series1` just crossed over (above) `series2`.

```
>>> crossover(self.data.Close, self.sma) True
```

```
def plot_heatmaps(heatmap, agg='max', *, ncols=3, plot_width=1200, filename='', open_browser=True)
```

Plots a grid of heatmaps, one for every pair of parameters in `heatmap`. See example in [the tutorial].

[the tutorial]: https://kernc.github.io/backtesting.py/doc/examples/Parameter%20Heatmap%20&%20Optimization.html#plot-heatmap # noqa: E501

`heatmap` is a Series as returned by `Backtest.optimize()` when its parameter `return_heatmap=True`.

When projecting the n-dimensional (n > 2) heatmap onto 2D, the values are aggregated by 'max' function by default. This can be tweaked with `agg` parameter, which accepts any argument pandas knows how to aggregate by.

TODO

Lay heatmaps out lower-triangular instead of in a simple grid. Like `sambo.plot.plot_objective()` does.

```
def quantile(series, quantile=None)
```

If `quantile()` is `None`, return the quantile *rank* of the last value of `series` wrt former series values.

If `quantile()` is a value between 0 and 1, return the *value* of `series` at this quantile. If used to working with percentiles, just divide your percentile amount with 100 to obtain quantiles.

```
>>> quantile(self.data.Close[-20:], .1) 162.130 >>> quantile(self.data.Close) 0.13
```

```
def random_ohlc_data(example_data, *, frac=1.0, random_state=None)
```

OHLC data generator. The generated OHLC data has basic descriptive statistics similar to the provided `example_data`.

`frac` is a fraction of data to sample (with replacement). Values greater than 1 result in oversampling.

Such random data can be effectively used for stress testing trading strategy robustness, Monte Carlo simulations, significance testing, etc.

```
>>> from backtesting.test import EURUSD >>> ohlc_generator = random_ohlc_data(EURUSD) >>>
next(ohlc_generator) # returns new random data ... >>> next(ohlc_generator) # returns new random data
...
```

```
def resample_apply(rule, func, series, *args, agg=None, **kwargs)
```

Apply `func` (such as an indicator) to `series`, resampled to a time frame specified by `rule`. When called from inside `Strategy.init()`, the result (returned) series will be automatically wrapped in `Strategy.I()` wrapper method.

`rule` is a valid Pandas offset string indicating a time frame to resample `series` to.

`func` is the indicator function to apply on the resampled series.

`series` is a data series (or array), such as any of the `Strategy.data` series. Due to pandas resampling limitations, this only works when input series has a datetime index.

`agg` is the aggregation function to use on resampled groups of data. Valid values are anything accepted by `pandas/resample/.agg()`. Default value for dataframe input is `OHLCV_AGG` dictionary. Default value for series input is the

appropriate entry from `OHLCV_AGG` if series has a matching name, or otherwise the value `"last"`, which is suitable for closing prices, but you might prefer another (e.g. `"max"` for peaks, or similar).

Finally, any `*args` and `**kwargs` that are not already eaten by implicit `Strategy.I()` call are passed to `func`.

For example, if we have a typical moving average function `SMA(values, lookback_period)`, *hourly* data source, and need to apply the moving average MA(10) on a *daily* time frame, but don't want to plot the resulting indicator, we can do:

```
class System(Strategy): def init(self): self.sma = resample_apply( 'D', SMA, self.data.Close, 10, plot=False)
```

The above short snippet is roughly equivalent to:

```
class System(Strategy): def init(self): # Strategy exposes <code>self.data</code> as raw NumPy arrays. # Let's convert closing prices back to pandas Series. close = self.data.Close.s # Resample to daily resolution. Aggregate groups # using their last value (i.e. closing price at the end # of the day). Notice `label='right'`. If it were set to # 'left' (default), the strategy would exhibit # look-ahead bias. daily = close.resample('D', label='right').agg('last') # We apply SMA(10) to daily close prices, # then reindex it back to original hourly index, # forward-filling the missing values in each day. # We make a separate function that returns the final # indicator array. def SMA(series, n): from backtesting.test import SMA return SMA(series, n).reindex(close.index).ffill() # The result equivalent to the short example above: self.sma = self.I(SMA, daily, 10, plot=False)
```

Classes

```
class FractionalBacktest (data, *args, fractional_unit=1e-08, **kwargs)
```

A `Backtest` that supports fractional share trading by simple composition. It applies roughly the transformation:

```
data = (data * fractional_unit).assign(Volume=data.Volume / fractional_unit)
```

as left unchallenged in this FAQ entry on GitHub, then passes `data`, `args*`, and `**kwargs` to its super.

Parameter `fractional_unit` represents the smallest fraction of currency that can be traded and defaults to one satoshi. For µBTC trading, pass `fractional_unit=1/1e6`. Thus-transformed backtest does a whole-sized trading of `fractional_unit` units.

Ancestors

- Backtest

Inherited members

`Backtest`:

- `optimize`
- `plot`
- `run`

```
class MultiBacktest (df_list, strategy_cls, **kwargs)
```

Multi-dataset `Backtest` wrapper.

Run supplied `Strategy` on several instruments, in parallel. Used for comparing strategy runs across many instruments or classes of instruments. Example:

```
from backtesting.test import EURUSD, BTCUSD, SmaCross btm = MultiBacktest([EURUSD, BTCUSD], SmaCross)
stats_per_ticker: pd.DataFrame = btm.run(fast=10, slow=20) heatmap_per_ticker: pd.DataFrame =
btm.optimize(...)
```

Methods

`def optimize(self, **kwargs)`

Wraps `Backtest.optimize()`, but returns `pd.DataFrame` with currency indexes in columns.

```
heamap: pd.DataFrame = btm.optimize(...) from backtesting.plot import plot_heatmaps
plot_heatmaps(heatmap.mean(axis=1))
```

`def run(self, **kwargs)`

Wraps `Backtest.run()`. Returns `pd.DataFrame` with currency indexes in columns.

`class SignalStrategy`

A simple helper strategy that operates on position entry/exit signals. This makes the backtest of the strategy simulate a vectorized backtest. See tutorials for usage examples.

To use this helper strategy, subclass it, override its `Strategy.init()` method, and set the signal vector by calling `SignalStrategy.set_signal()` method from within it.

```
class ExampleStrategy(SignalStrategy): def init(self): super().init() self.set_signal(sma1 > sma2, sma1
< sma2)
```

Remember to call `super().init()` and `super().next()` in your overridden methods.

Ancestors

- Strategy

Methods

`def set_signal(self, entry_size, exit_portion=None, *, plot=True)`

Set entry/exit signal vectors (arrays).

A long entry signal is considered present wherever `entry_size` is greater than zero, and a short signal wherever `entry_size` is less than zero, following `Order.size` semantics.

If `exit_portion` is provided, a nonzero value closes portion the position (see `Trade.close()`) in the respective direction (positive values close long trades, negative short).

If `plot` is `True`, the signal entry/exit indicators are plotted when `Backtest.plot()` is called.

Inherited members

`Strategy`:

- `I`
- `buy`
- `closed_trades`

- data
- equity
- init
- next
- orders
- position
- sell
- trades

`class TrailingStrategy`

A strategy with automatic trailing stop-loss, trailing the current price at distance of some multiple of average true range (ATR). Call `TrailingStrategy.set_trailing_sl()` to set said multiple (`6` by default). See tutorials for usage examples.

Remember to call `super().init()` and `super().next()` in your overridden methods.

Ancestors

- Strategy

Methods

`def set_atr_periods(self, periods=100)`

Set the lookback period for computing ATR. The default value of 100 ensures a *stable* ATR.

`def set_trailing_pct(self, pct=0.05)`

Set the future trailing stop-loss as some percent (`0 < pct < 1`) below the current price (default 5% below).

Note: Stop-loss set by `pct` is inexact

Stop-loss set by `set_trailing_pct` is converted to units of ATR with `mean(Close * pct / atr)` and set with `set_trailing_sl`.

`def set_trailing_sl(self, n_atr=6)`

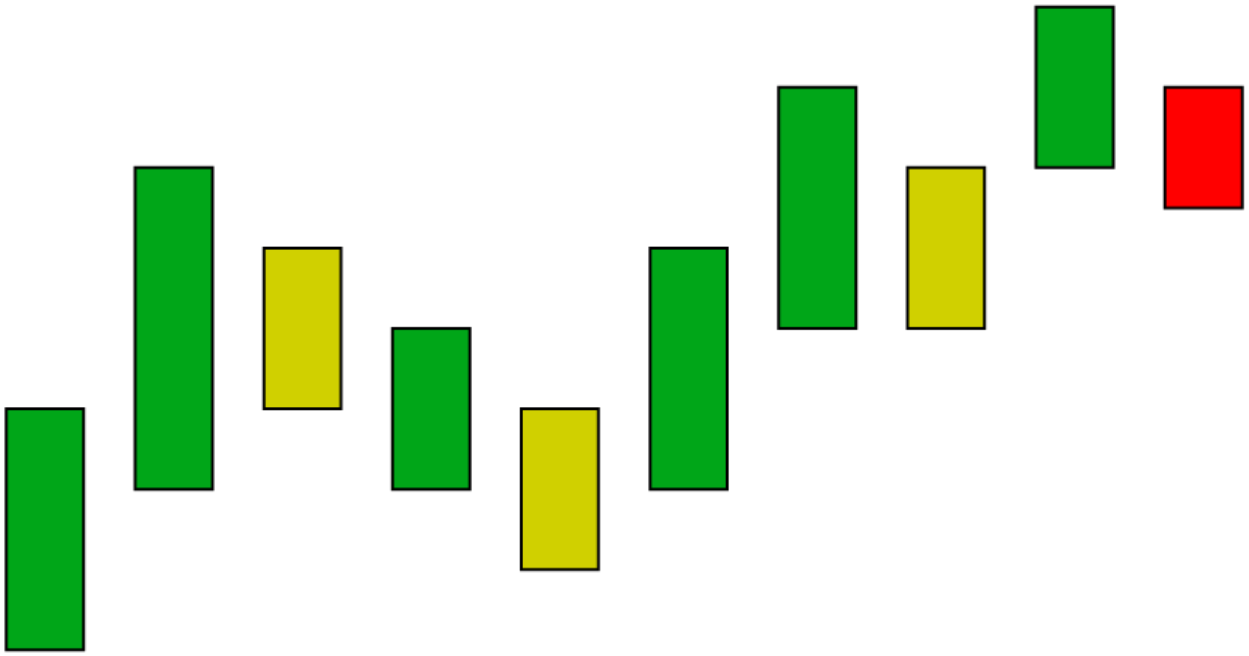
Set the future trailing stop-loss as some multiple (`n_atr`) average true bar ranges away from the current price.

Inherited members

`Strategy`:

- I
- buy
- closed_trades
- data
- equity
- init
- next
- orders
- position
- sell

- trades



Backtesting.py

Super-module

- backtesting

Global variables

- OHLCV_AGG
- TRADES_AGG

Functions

- barssince
- compute_stats
- cross
- crossover
- plot_heatmaps
- quantile
- random_ohlc_data
- resample_apply

Classes

FractionalBacktest

MultiBacktest

- optimize
- run

SignalStrategy

- `set_signal`

`TrailingStrategy`

- `set_atr_periods`
- `set_trailing_pct`
- `set_trailing_sl`