Backtesting.py Quick Start User Guide¶

This tutorial shows some of the features of *backtesting.py*, a Python framework for backtesting trading strategies.

Backtesting.py is a small and lightweight, blazing fast backtesting framework that uses state-of-the-art Python structures and procedures (Python 3.6+, Pandas, NumPy, Bokeh). It has a very small and simple API that is easy to remember and quickly shape towards meaningful results. The library *doesn't* really support stock picking or trading strategies that rely on arbitrage or multi-asset portfolio rebalancing; instead, it works with an individual tradeable asset at a time and is best suited for optimizing position entrance and exit signal strategies, decisions upon values of technical indicators, and it's also a versatile interactive trade visualization and statistics tool.

Data¶

You bring your own data. Backtesting ingests *all kinds of OHLC data* (stocks, forex, futures, crypto, ...) as a pandas.DataFrame with columns `'Open'`, `'High'`, `'Low'`, `'Close'` and (optionally) `'Volume'`. Such data is widely obtainable, e.g. with packages:

- pandas-datareader,
- Quandl,
- findatapy,
- yFinance,
- investpy, etc.

Besides these columns, **your data frames can have additional columns which are accessible in your strategies in a similar manner**.

DataFrame should ideally be indexed with a *datetime index* (convert it with `pd.to_datetime()`); otherwise a simple range index will do.

In [1]:

```
# Example OHLC daily data for Google Inc. from backtesting.test import GOOG GOOG.tail()
```

Loading BokehJS ...

Out[1]:

	Open	High	Low	Close	Volume
2013-02-25	802.30	808.41	790.49	790.77	2303900
2013-02-26	795.00	795.95	784.40	790.13	2202500
2013-02-27	794.80	804.75	791.11	799.78	2026100
2013-02-28	801.10	806.99	801.03	801.20	2265800
2013-03-01	797.80	807.14	796.15	806.19	2175400

Strategy¶

Let's create our first strategy to backtest on these Google data, a simple moving average (MA) cross-over strategy.

Backtesting.py doesn't ship its own set of *technical analysis indicators*. Users favoring TA should probably refer to functions from proven indicator libraries, such as TA-Lib or Tulipy, but for this example, we can define a simple helper moving average function ourselves:

In [2]:

```
import pandas as pd def SMA(values, n): """ Return simple moving average of `values`, at each step taking into account `n` previous values. """ return pd.Series(values).rolling(n).mean()
```

A new strategy needs to extend `Strategy` class and override its two abstract methods: `init()` and `next()`.

Method `init()` is invoked before the strategy is run. Within it, one ideally precomputes in efficient, vectorized manner whatever indicators and signals the strategy depends on.

Method `next()` is then iteratively called by the `Backtest` instance, once for each data point (data frame row), simulating the incremental availability of each new full candlestick bar.

Note, *backtesting.py* cannot make decisions / trades *within* candlesticks — any new orders are executed on the next candle's *open* (or the current candle's *close* if `trade_on_close=True`). If you find yourself wishing to trade within candlesticks (e.g. daytrading), you instead need to begin with more fine-grained (e.g. hourly) data.

In [3]:

```
from backtesting import Strategy from backtesting.lib import crossover class SmaCross(Strategy): # Define the two MA lags as *class variables* # for later optimization n1 = 10 n2 = 20 def init(self): # Precompute the two moving averages self.sma1 = self.I(SMA, self.data.Close, self.n1) self.sma2 = self.I(SMA, self.data.Close, self.n2) def next(self): # If sma1 crosses above sma2, close any existing # short trades, and buy the asset if crossover(self.sma1, self.sma2): self.position.close() self.buy() # Else, if sma1 crosses below sma2, close any existing # long trades, and sell the asset elif crossover(self.sma2, self.sma1): self.position.close() self.sell()
```

In `init()` as well as in `next()`, the data the strategy is simulated on is available as an instance variable `self.data`.

In `init()`, we declare and **compute indicators indirectly by wrapping them in `self.I()`**. The wrapper is passed a function (our `SMA` function) along with any arguments to call it with (our *close* values and the MA lag). Indicators wrapped in this way will be automatically plotted, and their legend strings will be intelligently inferred.

In `next()`, we simply check if the faster moving average just crossed over the slower one. If it did and upwards, we close the possible short position and go long; if it did and downwards, we close the open long position and go short. Note, we don't adjust order size, so *Backtesting.py* assumes *maximal possible position*. We use `backtesting.lib.crossover()` function instead of writing more obscure and confusing conditions, such as:

In [4]:

```
%%script echo def next(self): if (self.sma1[-2] < self.sma2[-2] and self.sma1[-1] > self.sma2[-1]): self.position.close() self.buy() elif (self.sma1[-2] > self.sma2[-2] and # Ugh! self.sma1[-1] < self.sma2[-1]): self.position.close() self.sell()
```

In `init()`, the whole series of points was available, whereas **in `next()`, the length of `self.data` and all declared indicators is adjusted** on each `next()` call so that `array[-1]` (e.g. `self.data.Close[-1]` or `self.sma1[-1]`) always contains the most recent value, `array[-2]` the previous value, etc. (ordinary Python indexing of ascending-sorted 1D arrays).

Note: `self.data` and any indicators wrapped with `self.I` (e.g. `self.sma1`) are NumPy arrays for performance reasons. If you prefer pandas Series or DataFrame objects, use `Strategy.data.<column>.s` or `Strategy.data.df` accessors respectively. You could also construct the series manually, e.g. `pd.Series(self.data.Close, index=self.data.index)`.

We might avoid `self.position.close()` calls if we primed the `Backtest` instance with `Backtest(..., exclusive_orders=True)`.

Backtesting¶

Let's see how our strategy performs on historical Google data. The `Backtest` instance is initialized with OHLC data and a strategy *class* (see API reference for additional options), and we begin with 10,000 units of cash and set broker's commission to realistic 0.2%.

In [5]:

```
from backtesting import Backtest bt = Backtest(GOOG, SmaCross, cash=10_000, commission=.002) stats = bt.run() stats
```

Out[5]:

```
Start 2004-08-19 00:00:00 End 2013-03-01 00:00:00 Duration 3116 days 00:00:00 Exposure Time [%] 94.27 Equity Final [$] 56263.52 Equity Peak [$] 56309.06 Commissions [$] 10563.95 Return [%] 462.64 Buy & Hold Return [%] 607.37 Return (Ann.) [%] 22.47 Volatility (Ann.) [%] 37.41 CAGR [%] 14.99 Sharpe Ratio 0.60 Sortino Ratio 1.14 Calmar Ratio 0.66 Alpha [%] 450.62 Beta 0.02 Max. Drawdown [%] -33.93 Avg. Drawdown [%] -6.16 Max. Drawdown Duration 830 days 00:00:00 Avg. Drawdown Duration 50 days 00:00:00 # Trades 93 Win Rate [%] 52.69 Best Trade [%] 56.92 Worst Trade [%] -16.83 Avg. Trade [%] 1.76 Max. Trade Duration 121 days 00:00:00 Avg. Trade Duration 32 days 00:00:00 Profit Factor 1.99 Expectancy [%] 2.29 SQN 1.58 Kelly Criterion 0.21 _strategy SmaCross _equity_curve Equ... _trades Size EntryB... dtype: object
```

`Backtest.run()` method returns a pandas Series of simulation results and statistics associated with our strategy. We see that this simple strategy makes almost 600% return in the period of 9 years, with maximum drawdown 33%, and with longest drawdown period spanning almost two years ...

`Backtest.plot()` method provides the same insights in a more visual form.

In [6]:

```
bt.plot()
```

Out[6]:

GridPlot(id = 'p1349', …)

Optimization¶

We hard-coded the two lag parameters (`n1` and `n2`) into our strategy above. However, the strategy may work better with 15–30 or some other cross-over. **We declared the parameters as optimizable by making them [class variables](#)**.

We optimize the two parameters by calling `Backtest.optimize()` method with each parameter a keyword argument pointing to its pool of possible values to test. Parameter `n1` is tested for values in range between 5 and 30 and parameter `n2` for values between 10 and 70, respectively. Some combinations of values of the two parameters are invalid, i.e. `n1` should not be *larger than* or equal to `n2`. We limit admissible parameter combinations with an *ad hoc* constraint function, which takes in the parameters and returns `True` (i.e. admissible) whenever `n1` is less than `n2`. Additionally, we search for such parameter combination that maximizes return over the observed period. We could instead choose to optimize any other key from the returned `stats` series.

In [7]:

```
%%time stats = bt.optimize(n1=range(5, 30, 5), n2=range(10, 70, 5), maximize='Equity Final [$]', constraint=lambda param: param.n1 < param.n2) stats
```

```
CPU times: user 128 ms, sys: 43.3 ms, total: 171 ms Wall time: 1.66 s
```

Out[7]:

```
Start 2004-08-19 00:00:00 End 2013-03-01 00:00:00 Duration 3116 days 00:00:00 Exposure Time [%] 98.14 Equity Final [$] 77829.05 Equity Peak [$] 84982.19 Commissions [$] 30771.04 Return [%] 678.29 Buy & Hold Return [%] 687.99 Return (Ann.) [%] 27.22 Volatility (Ann.) [%] 43.21 CAGR [%] 18.05 Sharpe Ratio 0.63 Sortino Ratio 1.28 Calmar Ratio 0.61 Alpha [%] 614.80 Beta 0.09 Max. Drawdown [%] -44.55 Avg. Drawdown [%] -5.81 Max. Drawdown Duration 1558 days 00:00:00 Avg. Drawdown Duration 50 days 00:00:00 # Trades 152 Win Rate [%] 50.00 Best Trade [%] 61.36 Worst Trade [%] -19.98 Avg. Trade [%] 1.32 Max. Trade Duration 83 days 00:00:00 Avg. Trade Duration 21 days 00:00:00 Profit Factor 1.83 Expectancy [%] 1.75 SQN 1.28 Kelly Criterion 0.13 _strategy SmaCross(n1=10,n... _equity_curve Equ... _trades Size Entry... dtype: object
```

We can look into `stats['_strategy']` to access the Strategy *instance* and its optimal parameter values (10 and 15).

In [8]:

```
stats._strategy
```

Out[8]:

```
<Strategy SmaCross(n1=10,n2=15)>
```

In [9]:

```
bt.plot(plot_volume=False, plot_pl=False)
```

Out[9]:

GridPlot(id = 'p1618', …)

Strategy optimization managed to up its initial performance *on in-sample data* by almost 50% and even beat simple buy & hold. In real life optimization, however, do **take steps to avoid overfitting**.

Trade data¶

In addition to backtest statistics returned by `Backtest.run()` shown above, you can look into *individual trade returns* and the changing *equity curve* and *drawdown* by inspecting the last few, internal keys in the result series.

In [10]:

```
stats.tail()
```

Out[10]:

```
SQN 1.28 Kelly Criterion 0.13 _strategy SmaCross(n1=10,n2=15) _equity_curve Equity DrawdownPct DrawdownDurat... _trades Size EntryBar ExitBar EntryPrice Exit... dtype: object
```

The columns should be self-explanatory.

In [11]:

```
stats['_equity_curve'] # Contains equity/drawdown curves. DrawdownDuration is only defined at ends of DD periods.
```

Out[11]:

	Equity	DrawdownPct	DrawdownDuration
2004-08-19	10000.00	0.00	NaT
2004-08-20	10000.00	0.00	NaT
2004-08-23	10000.00	0.00	NaT
2004-08-24	10000.00	0.00	NaT
2004-08-25	10000.00	0.00	NaT
...	...	...	...
2013-02-25	76348.73	0.10	NaT
2013-02-26	76287.29	0.10	NaT
2013-02-27	77213.69	0.09	NaT
2013-02-28	77350.01	0.09	NaT
2013-03-01	77829.05	0.08	1558 days

2148 rows × 3 columns

In [12]:

```
stats['_trades'] # Contains individual trade data
```

Out[12]:

	Size	EntryBar	ExitBar	EntryPrice	ExitPrice	SL	TP	PnL	Commission	ReturnPct	EntryTime	ExitTime	Duration	Tag	Entry_SMA(C,10)	Exit_SMA(C,10)	Entry_SMA(C,15)	Exit_SMA(C,15)
0	87	20	60	114.42	185.23	None	None	6108.33	52.14	0.61	2004-09-17	2004-11-12	56 days	None	107.40	181.07	105.75	183.32
1	-86	60	69	185.23	175.80	None	None	748.88	62.10	0.05	2004-11-12	2004-11-26	14 days	None	181.07	173.56	183.32	173.14
2	95	69	71	175.80	180.71	None	None	398.71	67.74	0.02	2004-11-26	2004-11-30	4 days	None	173.56	173.18	173.14	174.55
3	-95	71	75	180.71	179.13	None	None	81.73	68.37	0.00	2004-11-30	2004-12-06	6 days	None	173.18	176.58	174.55	175.51
4	96	75	82	179.13	177.99	None	None	-178.01	68.57	-0.01	2004-12-06	2004-12-15	9 days	None	176.58	175.15	175.51	176.58
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
147	-90	2056	2085	740.13	687.78	None	None	4454.48	257.02	0.07	2012-10-16	2012-11-29	44 days	None	752.66	667.39	753.99	664.45
148	104	2085	2111	687.78	735.54	None	None	4670.99	296.05	0.07	2012-11-29	2013-01-08	40 days	None	667.39	718.50	664.45	719.00
149	-103	2111	2113	735.54	742.83	None	None	-1055.41	304.54	-0.01	2013-01-08	2013-01-10	2 days	None	718.50	724.62	719.00	721.51
150	101	2113	2121	742.83	735.99	None	None	-989.56	298.72	-0.01	2013-01-10	2013-01-23	13 days	None	724.62	724.32	721.51	726.41
151	-100	2121	2127	735.99	750.51	None	None	-1749.30	297.30	-0.02	2013-01-23	2013-01-31	8 days	None	724.32	738.20	726.41	735.12

152 rows × 18 columns

Learn more by exploring further examples or find more framework options in the full API reference.