BACKTESTING.PY

USER GUIDE

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.

Step by step

1. Import libraries:

```
import pandas as pd
import yfinance as yf
import numpy as np
import talib as talib
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
from backtesting import Strategy, Backtest
from backtesting.lib import crossover
import seaborn as sns
import pandas_datareader as pdr
```

2. Create DataFrame:

Pull data from:

Yahoo Finance using:

- df = yf.download(‘asset’, start=‘datetime’, end=‘datetime’, interval=‘1h’)

Pandas DataReader (Naver Finance, Google Analytics, etc) using:

- ts = pdr.av.time_series.AVTimeSeriesReader(‘asset’, api_key=xxxxx)
- df = ts.read()
- df.index = pd.to_datetime(df_index, format= ‘%Y-%m-%d’)

3. Be assure that “Date” column is in datetime and is the index

Use df.reset_index() and then use df.dtypes to be assure that the “date” format is in datetime; if not, use df[‘Date’] = pd.to_datetime(df[‘Date’]). After this step is done, use df.set_index(‘Date’, inplace=True) to set Date as index.

4. Define the technical indicators

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. I can use as many indicators as I want.

Example:

```
def technical_indicator():
    ‘’’
    return the technical indicator
    ‘’’
    return <use TA-Lib to calculate indicator>
```

5. Define Strategy

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.
Example:

```
Class name_strategy(Strategy):
    # if needed I can define the parameters of the indicators as “class variables” for later optimization
    n1 = x
    n2 = x
    n3 = x

    def init(self):
        # precomputes whatever indicators and signals the strategy depends on
        self.indicator1 = self.I(technical_indicator, self.data.Close, self.n1)
        self.indicator2 = self.I(technical_indicator, self.data.Close, self.n2)

    def next(self):
        # define here the trading logic: if X do this, then buy, if Y do that, sell
        if self.indicator1 == X:
            self.position.close()
            self.buy()
        elif self.indicator2 > Y:
            self.position.close()
            self.sell()
```

6. Define the backtest

The Backtest instance is initialized with OHLC data and a strategy "*class name_strategy"* (see API reference for additional options), and we begin with 10.000 units of cash and set broker's comisión to realistic 0.2%.

Example:

```
bt = Backtest(df, name_strategy, cash=xxxxx, commission=0.02, exclusive_orders=True)

stats = bt.run()

print(stats)

bt.plot()
```

bt.run() method returns a pandas Series of simulation results and statistics associated with our strategy.

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

7. Optimization

We hard-coded the lag parameters (n1,n2,n3) in the “class” above. However, the strategy mau work better with other parameters or some other modification. **We declared the parameters as optimizable by making them class variables**.

We optimize the parameters by calling bt.optimize() method with each parameter a keyword argument pointing to its pool of posible values to test. In this example, parameter 1 (n1) is tested for values in range between 5 and 30 and parameter 2 (n2) is tested for values in range between 10 and 70. Some combinations of values of the parameters are invalid, i.e. n1 should not be larger than or equal to n2. We limit admisible parameter combinations with an “ad hoc” constrait function, which takes in the parameters and returns “True” (i.e. admissible) whenever n1 is les 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.

7. Optimization

Example:

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

We can look into stats['_strategy'] to Access the Strategy instance and its optimal parameter values:

```
print(stats._strategy)
```

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

We can plot the optimization too:

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

8. 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.

Example:

print(stats.tail())

```
Expectancy [%]                                     1.98
SQN                                                1.60
_strategy                            SmaCross(n1=10,n2=15)
_equity_curve                  Equity  DrawdownPct DrawdownDura...
_trades                Size  EntryBar  ExitBar  EntryPrice  Exit...
dtype: object
```

The columns should be self-explanatory
print(stats['_equity_curve']) *# Contains equity/drawdown curves. DrawdownDuration is only defined at ends of DD periods.*
print(stats['_trades']) *# Contains individual trade data*

	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	103035.53	0.05	NaT
2013-02-26	102952.33	0.05	NaT
2013-02-27	104206.83	0.04	NaT
2013-02-28	104391.43	0.04	NaT
2013-03-01	103949.43	0.04	533 days

2148 rows × 3 columns

	Size	EntryBar	ExitBar	EntryPrice	ExitPrice	PnL	ReturnPct	EntryTime	ExitTime	Duration
0	87	20	60	114.65	185.23	6140.56	0.62	2004-09-17	2004-11-12	56 days
1	-87	60	69	184.86	175.80	788.18	0.05	2004-11-12	2004-11-26	14 days
2	96	69	71	176.15	180.71	437.61	0.03	2004-11-26	2004-11-30	4 days
3	-96	71	75	180.35	179.13	116.98	0.01	2004-11-30	2004-12-06	6 days
4	97	75	82	179.49	177.99	-145.33	-0.01	2004-12-06	2004-12-15	9 days
...	...	...	...	...	...	...	...	...	...	...
148	139	2085	2111	689.16	735.54	6447.44	0.07	2012-11-29	2013-01-08	40 days
149	-139	2111	2113	734.07	742.83	-1217.79	-0.01	2013-01-08	2013-01-10	2 days
150	136	2113	2121	744.32	735.99	-1132.29	-0.01	2013-01-10	2013-01-23	13 days
151	-136	2121	2127	734.52	750.51	-2174.91	-0.02	2013-01-23	2013-01-31	8 days
152	130	2127	2147	752.01	797.80	5952.57	0.06	2013-01-31	2013-03-01	29 days

153 rows × 10 columns

FULL API REFERENCE

Documentation

- https://kernc.github.io/backtesting.py/doc/backtesting/#gsc.tab=0

- https://pypi.org/project/Backtesting/

OTHER REFERENCES

- **For technical indicators**:

TA-Lib: https://mrjbq7.github.io/ta-lib/doc_index.html

Tulip Indicators: https://tulipindicators.org/

- **For data scrapping**:

Python-Binance: https://python-binance.readthedocs.io/en/latest/

Others: https://www.alpharithms.com/python-financial-data-491110/

https://analyticsindiamag.com/top-python-libraries-to-get-historical-stock-data-with-code/

Pandas: https://pandas.pydata.org/docs/

Trading with Python: https://sjev.github.io/trading-with-python/historicData.html

Financial Modeling Prep: https://site.financialmodelingprep.com/developer/docs#Symbols-List

- **For data calculations:**

Numpy: https://numpy.org/doc/

- **For data visualization:**

Seaborn: https://seaborn.pydata.org/

MatPlotLib: https://matplotlib.org/stable/index.html