Quantiacs Python Toolbox Documentation

Release 2.2

Quantiacs

September 09, 2016

Contents

Table of contents:

CHAPTER 1

Support Packages

Python packages supported are:

- numpy
- Pandas
- scipy
- scikit-learn
- ta-lib
- keras
- tensorflow

CHAPTER 2

Trading System Structure

In Python, you have to define two functions: `mySettings` and `myTradingSystem`. The toolbox relies on these two functions to run your trading system. `mySettings` contains the settings structure which store all relevant information for the trading system. The toolbox allows for flexible data definitions, where you can request OPEN, HIGH, LOW, CLOSE, or VOL data for your system. A typical function definition is shown below:

```
def myTradingSystem(DATE, HIGH, CLOSE, VOL, exposure, equity, settings)
```

Your `myTradingSystem` function will be called each day of the backtesting period with the most recent data as arguments. The only requirements of `myTradingSystem` is to return an array of your trading system's market positions for the next day and the `settings` data struct. Market positions, `p`, is just an array of numbers [0 1 -1 ...] that represents the trading positions you want to take (e.g., no position, buy position, sell position).

2.1 Arguments/Parameters

Data can be requested for your trading system through the arguments of the function definition. The `myTradingSystem` function isn't required to call any arguments, the example above is just a representation of various values that can be called.

Here is a breakdown of what parameters can be loaded into the trading system:

Parameter	Description	Dimensions (rows x columns)
DATE	a date integer in the format YYYYMMDD	Lookback x 1
OPEN	the first price of the session	Lookback x # of Markets
HIGH	the highest price of the session	Lookback x # of Markets
LOW	the lowest price of the session	Lookback x # of Markets
CLOSE	the last price of the session	Lookback x # of Markets
VOL	number of stocks/contracts traded per session	Lookback x # of Markets
exposure	the realized quantities of your trading system, or all the trading positions you take	Lookback x # of Markets
equity	cumulative trading performance in each market, reflects gains and losses	Lookback x # of Markets
OI, R, RINFO	also available to be called as parameters, described in more detail under *Market Data* section	

The data is structured with the most recent information in the last index or row. Where **lookback** is simply how many historical data points you want to load in each iteration of your trading system. Given the data is daily, lookback represents the maximum number of days of market data you want to have loaded. Now the two parameters that aren't explicitly related to market data are **exposure** and **equity**. These two relate to trading in your simulated broker account. Whenever you make a trading position, that's recorded in **exposure**. And whatever the market result of your trading position is, is then recorded in **equity**. Or simply put: **equity** is what you made with each market over your **lookback** period.

CHAPTER 3

Market Positions & Trading

A unique attribute of the Quantiacs toolbox is the use of market exposure rather than trades. Instead of telling the system to buy and sell a certain number of stocks or futures contracts, you can allocate your available assets and choose how much market exposure you want in a specific stock/future. This is done by adjusting an array `p` and returning/outputing it from your trading system function. In order to figure out how big your market exposure is, Quantiacs looks at all your positions and allocates your capital proportionally. The position vector is normalized to an absolute sum of 1, so each `p` value is evaluated proportionally to every other position value.

p = 0	you don't hold asset i in your portfolio
p >= 0	you are invested long in asset i with weight p
p < 0	you are invested short in asset i with weight p

Below is a code example of setting the positions for futures:

```
settings['markets'] =  ['F_GC', 'F_ES']
% your trading algorithm
p = [2.4, -1.2]
```

Here is an example, let's say your capital is set to $1,000,000 in settings (see *Settings* section), and you decide to buy Gold (F_GC) and sell the S&P 500 (F_ES). Let's say you set your `p` vector to 2.4 in the market column representing Gold and to -1.2 for S&P 500, and take no other positions. Then the system would take your $1 million capital, and allocate it proportionally to the market exposure you just set above. In other words, 2x the capital would go towards going long Gold as it would short the S&P 500. Your market exposure in Gold would become 2.4/3.6 * 10^6 = $666,667 and our negative short position in S&P 500 would become 1.2/3.6 * 10^6 = $333,333. Then whatever your returns are the next day, your balance and market exposure would adjust accordingly (assuming the positions remained the same at 2.4 and 1.2). This representation of market exposure isn't 100% accurate of the real world, where you would have to buy and sell discrete amounts of contracts, but for large balances it should average out to roughly the same performance.

All of this is done behind the scenes. You can see the result in **equity** and can track your chosen market exposure positions in **exposure**.

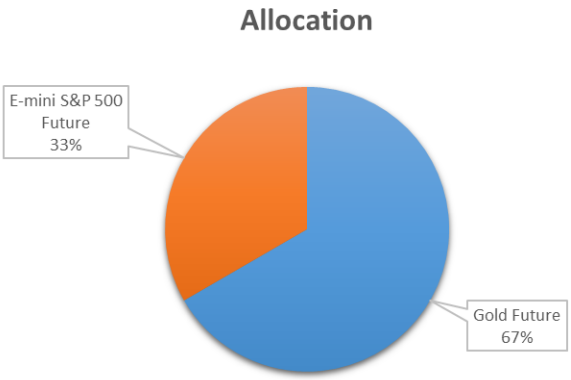

Fig. 3.1: Market allocation.

CHAPTER 4

Settings

Settings contain your environment variables and are open to being defined and extended with additional fields being added to the settings struct.

Below is an example of configuring settings:

```
def mySettings():
''' Define your trading system settings here '''

    settings= {}

    settings['markets']  = ['CASH','F_AD', 'F_BO', 'F_BP', 'F_C', 'F_CC', 'F_CD',
    'F_CL', 'F_CT', 'F_DX', 'F_EC', 'F_ED', 'F_ES', 'F_FC','F_FV', 'F_GC',
    'F_HG', 'F_HO', 'F_JY', 'F_KC', 'F_LB', 'F_LC', 'F_LN', 'F_MD', 'F_MP',
    'F_NG', 'F_NQ', 'F_NR', 'F_O', 'F_OJ', 'F_PA', 'F_PL', 'F_RB', 'F_RU',
    'F_S','F_SB', 'F_SF', 'F_SI', 'F_SM', 'F_TU', 'F_TY', 'F_US','F_W', 'F_XX',
    'F_YM']
    settings['beginInSample'] = '20120506'
    settings['endInSample'] = '20150506'
    settings['lookback']= 504
    settings['budget']= 10**6
    settings['slippage']= 0.05

    return settings
```

4.1 Default Settings

Your algorithm will always be called with the following settings by default unless you define them otherwise.

```
settings['beginInSample'] = '19900101'
settings['lookback']      = 504
settings['budget']        = 1000000
settings['slippage']      = 0.05
```

Note that you aren't required to define any settings field, but for most trading systems **lookback** will likely need to be changed from 1 if your trading system needs any historical market data. Leaving **lookback** at the default value while attempting to access market prices further back than the previous day will result in an error.

4.2 Markets

Currently Quantiacs trades in both Stock and Futures markets. A Future is a contract to deliver a specified amount of the underlying product at a certain date. You can learn more about quant trading with Futures from the video below.

Trading in both Stocks and Futures allows for a great degree of diversification and helps you to generate a trading system that is successful in many market environments. You can find the full list of available markets on Quantiacs' Markets page.

In general, you can find the contract unit of any futures by looking at the Futures Contracts Specifications. And finding the contract unit allows you to relate a futures contract to their commonly found prices such as on Yahoo Finance.

Another unique aspect of futures contracts that doesn't apply to stocks, is an expiration date on each contract. The only real difference in your trading data is that the Quantiacs toolbox automatically implements a roll over (see *Roll Overs (R & RINFO)* section below) whenever a futures contract expires, to make the pricing data the latest and most accurate.

4.3 Market Data

Note that Quantiacs market data is not updated in realtime, this means you won't be able to do live trading using the toolbox. If a market is not defined for some time period, its values are set to not a number (NaN). If data is missing within the time series (for example due to a holiday), the missing data is replaced by the last known value with zero volume.

If you want to backtest across every available market, you can get a text file with a string for all of the markets here. This makes it easy to copy paste all of the available markets into your settings struct.

Here is an example of what the data looks like for a futures contract:

F_AD	Australian Dollar								
DATE	OPEN	HIGH	LOW	CLOSE	VOL	OI	P	R	RINFO
19900102	77300	77400	77020	77020	125	2559	0	0	0
19900103	76890	77030	76700	76740	1495	3215	0	0	0
19900104	77080	77610	77000	77490	932	3122	0	0	0
19900105	77050	77280	76980	76980	272	2542	0	0	0
19900108	77280	77300	77090	77200	177	2439	0	0	0
19900109	77289	77370	77210	77290	106	2379	0	0	0

And here is what the data fields look like for a stock:

AAPL	(Apple)					
DATE	OPEN	HIGH	LOW	CLOSE	VOL	P
20010102	1.067	1.0893	1.0446	1.0625	1.12E+08	0
20010103	1.0357	1.192	1.0313	1.1696	2.02E+08	0
20010104	1.2946	1.3125	1.2054	1.2188	1.84E+08	0
20010105	1.2098	1.2411	1.1473	1.1696	1.02E+08	0
20010108	1.2098	1.2098	1.1384	1.183	92568000	0
20010109	1.2009	1.2589	1.183	1.2277	1.44E+08	0

The P column is for backwards compatibility to support the Quantiacs 1.X Toolbox versions. OI represents open interest for futures contracts, and R and RINFO both provide information about futures contracts roll overs (see *Roll Overs (R & RINFO)* section below).

4.4 Loading Market Data

Whenever you run `runts`, it will automatically download the necessary market data. When backtesting across new markets, or a new sample size, the toolbox will automatically download the corresponding market data if it hasn't been downloaded before.

To manually initiate this process, you can use the command `loadData`. You can find a full breakdown of `loadData` under *Reference* section. The main arguments `loadData` needs are a list of markets you want downloaded, and the time period (sample size). An example use case would be:

```
quantiacsToolbox.loadData(marketList = allMarkets, dataToLoad = ['DATE','OPEN','CLOSE'], beginInSampl
```

4.5 Roll Overs (R & RINFO)

Futures, as opposed to Stocks, come in single contracts with an expiration (delivery) date. This requires that we treat futures contracts slightly different than stocks in the backtester. Since there is an expiration to the contract, we have to sell the contract before the expiry and buy a different contract (of the same underlying) that expires further in the future (this is called 'rolling' a contract). There are extra costs and uncertainties associated with this.

Rolling explains why the plot of the prices of the time series (as shown on the website) is not necessarily what you get when you buy and hold that commodity. The differences between the price plot and the trading result are higher for commodities and lower for financial futures, since the cost of carry for a Stock Index Future or a Government Bond is usually very low.

In the market data files (found in the *tickerData* folder of the toolbox), R and RINFO columns address roll overs. The data column R contains the roll announcement - the contract maturity of the new contract (i.e. the contract we're rolling into) in the format yyyymm. RINFO is the roll difference in the time series data. At a roll we back-adjust the data in the lookback window by RINFO to keep the time series data steady. We also adjust the performance by the roll amount since the price difference between the two contracts at the same time is not a win or a loss that can be traded. So our raw data are not continuous contracts, but single contracts.

Here is an example of rollover data from F_AD.txt:

DATE	OPEN	CLOSE	R	RINFO
20150902	70120.0000	70250.0000	0	0.0000
20150903	70360.0000	70100.0000	0	0.0000
20150904	70080.0000	69230.0000	201512	0.0000
20150908	68820.0000	69930.0000	0	-290.0000
20150909	69840.0000	69840.0000	0	0.0000
20150910	69500.0000	70480.0000	0	0.0000

Roll overs are all done automatically in `runts`, and because of this on-the-fly rolling method you always get:

1. The true Dollar value of the commodity at that point in time - at least for the last data point, i.e. the last row of the CLOSE matrix.
2. A steady course with no disruptions/gaps because of rolls.

4.6 Why Only Daily Data

Quantiacs only supports daily historical market data for several reasons. The first is that our investors want scalable strategies that can manage hundreds of millions rather than just hundreds of thousands. As limit orders can only be filled during those times of the session, in which the market trades below the limit, we'd only have a fraction of the session to execute these orders. Naturally this leads to a much lower capacity of the trading strategy. Additionally,

if we'd allow limit orders we would have to account for partial fills in the backtest, which could make the backtest results no longer representative in extreme cases.

Secondly, we are a Commodity Trading Advisor registered with the NFA, and we have to comply with the rules of our regulators. We have to protect our institutional clients from front-running, arbitrage and other potentially criminal activities. It's impossible to protect investors trading third party strategies on 1 minute bars. On end of day data we can ensure their protection from criminal activities.

We have to separate the strategic part of the trading system (its logic of when to buy what) strictly from the actual order execution and risk management, that's handled by us (and might actually involve the use of leverage, limit orders, stop loss orders etc.).

4.7 Sample Size

By default, the system will load market data for all dates available, so the backtest will run across the entire 25+ years of historical market data. Alternatively, you have the ability to define the specific start and end dates for your backtests through *beginInSample* and *endInSample* respectively. Both fields follow the format of YYYYMMDD.

4.8 Budget

Although you can change your budget to any size, it is good to test it at $1 million because that would provide it with the proper scale to effectively trade futures in the real world. Moreover, good trading strategies will show similar results whether they're traded at $1 million or $10 million.

Our backtester, no matter the budget allocated, assumes the ability to purchase non-discrete or fractional amounts of contracts. In reality this is not possible, however, it allows the trading strategy to be evaluated without significant deviation caused by budgets. Since futures generally have a very large contract size, there would be a big difference between real and intended allocations at lower capital sizes.

For example, if you attempted to manage your algorithm with 500k and had the following target allocation:

Market	Allocation	Cash in market	Price of 1 contract
F_ES	0.5	0.5 * 500k = 250k	104k
F_SI	0.2	0.2 * 500k = 250k	79k
F_GC	0.1	0.1 * 500k = 50k	118k
F_TY	0.1	0.1 * 500k = 50k	126k
F_FV	0.1	0.1 * 500k = 50k	119k

Again because of the large contract sizes of futures (and the fact that it is impossible to buy half contracts) a naïve discrete representation would give you 2 contracts F_ES, 1 contract F_SI, and ignore the rest. Thus the real exposure would be:

Market	Allocation	Price of 1 contract
F_ES	2 * 104 / 500 = 0.416	104k
F_SI	1 * 79 / 500 = 0.158	79k
F_GC	0	118k
F_TY	0	126k
F_FV	0	119k
CASH	0.426	

Realistically, any institution would put down at least $1 million to trade futures with. So our non-discrete trading positions turn out to be a better representation of real life trading situations.

4.9 Trading Costs

When writing your trading system, all trading costs are based off **slippage** (see *Slippage* section below), for example setting it to 0 will test your system without any trading costs. Trading costs can have a significant effect on the performance of a trading algorithm. The two main contributors to trading costs are commissions and slippage. Commissions are fees charged by the exchange and the broker. You cannot avoid them. In most cases they are quite low compared to amount of your trade. Slippage is the price at which you expected or placed your order and the price at which your order was actually filled. Factors like the liquidity and the volatility contribute to the slippage as well as the volume you want to trade. A good estimate for slippage is the daily range, therefore slippage can be estimated ex post.

In our backtesting toolbox we use a very simple yet conservative approach to estimate slippage and commissions: We take 5% of the daily range as the trading costs. This computes as (HIGH - LOW) * 0.05. This covers the assumption, that you'll have more slippage on days with larger market moves, than on days with smaller. This approximation might overestimate the real trading costs. In this case, it is better to overestimate than underestimate.

4.9.1 Slippage

Slippage is the difference between the price at which you expected or placed your order and the price at which your order was actually filled. The following factors contribute to the slippage. The liquidity of the market: Higher liquidity results in lower slippage. In very liquid markets your positions are filled almost immediately. In an illiquid market, the order execution could cost significant time, in this time the price might move against you. You will notice that the impact of slippage on your trading system depends on how frequently you trade and how much return each trade generates. If you trade often and have trades with smaller returns per trade, slippage will be an issue. If you don't change the size of your exposure often, slippage will be almost irrelevant for your results. These factors contribute to slippage:

- Your trading volume: The more shares you want to buy, the longer the order execution takes. The longer the order execution takes, the further the fill price might be.
- The bid-ask spread: This is the difference in the price quoted for an immediate buy (ask) and the price quoted for an immediate sale (bid). To get an order filled, you usually have to cross the spread. This is typically on the far side for you. If you want to sell 100 shares of Stock X you need to find a buyer for them. If there is one in the orderbook you will find him at the other side of the spread. Large bid/ask spreads lead to a high slippage.
- The volatility of the market: For the sake of simplicity, let's define volatility as the average change of price per unit of time. Thus, if the volatility is high, it's evident that slippage will be higher in volatile markets since prices tend to move more while your order is executed.

4.10 Extensibility and Custom Fields

The best part of settings is the ability to add custom fields to the settings struct.

```
settings['anotherField'] =  some_value
```

The only way to retain custom data from your trading system across multiple instances is the settings struct. Let's say you build a custom indicator, and you want to save the values it generates and make them available to your trading system. Remember that your trading system is just one big function that is called again for each new day of market data. In other words, nothing within your trading system (except the market positions) is saved across multiple dates. Settings are an exception to this rule, and they remain intact during the entire backtest. This allows you to record custom values and datatypes by adding your own fields to settings.

CHAPTER 5

Evaluating Your System

5.1 Backtesting

Start testing your system by running python command:

```
python path/to/trading_system.py
```

The only requirement is to have the following at the end of your python file to make the trading system file self-executable.

```
# Evaluate trading system defined in current file.
if __name__ == '__main__':
        import quantiacsToolbox
        results = quantiacsToolbox.runts(__file__)
```

Alternatively, you can always run the trading system straight from Python console:

```
> import quantiacsToolbox
> returnDict = quantiacsToolbox.runts('/Path/to/your/TradingSystem.py')
```

5.2 Trading Results

`runts` loads the market data and calls your trading system for each day of the backtest with the most recent market data. It simulates the equity curve for your output values `p`. The toolbox will return a dictionary of performance values and a plot of your system's performance.

`returnDict` contains the following fields:

Field	Data Type	Description
returnDict['tsName']	string	a string that contains the name of your trading system
returnDict['fundDate']	list	a list of the dates available to your trading system
returnDict['fundTradeDates']	list	a list of the,dates traded by your trading system
returnDict['fundEquity']	list	a list defining the equity of your portfolio
returnDict['marketEquity']	list	a list defining the equity earned in each market
returnDict['marketExposure']	list	a list containing the equity made from each market in settings['markets']
returnDict['errorLog']	list	a list describing any errors made in evaluation
returnDict['runtime']	float	a float containing the time required to evaluate your system
returnDict['evalDate']	int	an int that describes the last date of evaluation for your system (in YYYYMMDD format)
returnDict['stats']	dict	a dict containing the performance statistics of your system
returnDict['settings']	dict	a dict containting the settings defined in mySettings
returnDict['returns']	list	a list defining the market returns of your trading system

CHAPTER 6

Submitting a System

To submit your trading strategy to the Quantiacs marketplace simply click the button 'Submit Trading System' at the bottom of the evaluation GUI.

To submit a system to the Quantiacs server, simply run the submit function with the filepath to your system and the name you wish to use for your system. Submission can be done by running following commands in Python console:

```
> import quantiacsToolbox
> quantiacsToolbox.submit('/Path/to/your/TradingSystem.py','mySystemName')
```

Alternatively, you can upload a tradingsystem via the Upload button on the Website.

CHAPTER 7

Optimization

To compute several different parameterizations of the same system we can use the `optimize` function. To use this feature, place a comment next to the parameter you wish to scan. Within the comment place the parameter domain and step-size between two pound signs (similar to the Matlab vector notation).

The optimizer takes each set of optimization parameters and runs a backtest to return the results (Sharpe Ratio, volatility, etc.) for each parameter value. This is essentially a brute-force method, so keep in mind computation time significantly increases when testing many variables at once or a large range of parameters. Nonetheless, the optimizer is useful for automating a test of several parameters when you're looking for the best fit variable. The overall format looks like this:

```
emaPeriod = 30 #[20:5:90]#
```

In the above example, we see that the variable 'emaPeriod' will be tested from 20 to 90 in increments of 5, and will return the performance results from each run. This also allows you to extract any specific dictionary result that is found under Trading Results (see *Trading Results* section).

CHAPTER 8

Performance

What makes your system a good system? Universally defining the quality of a system is impossible, but a good first approximation is the Sharpe Ratio. The Sharpe Ratio compares the realized performance against the risk (or volatility) taken to achieve that performance. Every investor would prefer systems with high performance and low risk. This is why a higher Sharpe Ratio is usually better.

A good way to assess your trading strategy is to separately backtest it against an in-sample and out-of-sample set of historical data. For example, you can first backtest your strategy against 2/3 of the available historical data (the in-sample), optimize/test various parameters, then backtest it against the remaining 1/3 (out-of-sample) historical data. This allows you to get a sense of your trading system's true performance on the out-of-sample data because it won't have any of the optimization bias from your in-sample backtest.

8.1 Overfitting

Overfitting is the natural enemy of trading algorithms. With enough parameters, it's easy to fit a model perfectly to the past. Mathematically: You can always fit n data points perfectly with a polynomial of degree n-1. In other words, you can build extremely complicated algos that perfectly retell the past but have no clue what happens next. So, here are a few rules of thumb on how to avoid overfitting:

- Don't use too many parameters. You'll have a hard time to find the best solution in the parameter space. If you manage to isolate an effect with only a few parameters, it might be more stable.
- Test your hypothesis. Don't use all the data for the development of your algorithm. Save some for a test once you're done. You'll have a much higher chance of success on live market data, if your system doesn't fail your out-of-sample test.
- Avoid putting in knowledge of future market events. This means not stopping your strategy from trading during the 2008 recession, or using Apple as the stock for a long-only strategy since we already know it's historically a great choice for buying and holding.
- Use walk forward analysis. This is a bit more complex but involves testing your strategy over a small period of time, optimizing it, then testing it on another equal period of time that's out of sample. The intent is to mimic real world trading of the strategy.

CHAPTER 9

Reference

9.1 computeFees

quantiacsToolbox.**computeFees**(*equityCurve*, *managementFee*, *performanceFee*)

Computes equity curve after fees.

Args:

equityCurve (list, numpy array) : a column vector of daily fund values

managementFee (float) : the management fee charged to the investor (a portion of the AUM charged yearly)

performanceFee (float) : the performance fee charged to the investor (the portion of the difference between a new high and the most recent high, charged daily)

Returns:

returns an equity curve with the fees subtracted. (does not include the effect of fees on equity lot size)

9.2 fillnans

quantiacsToolbox.**fillnans**(*inArr*)

Fills in (column-wise) value gaps with the most recent non-nan value. Fills in value gaps with the most recent non-nan value. Leading nan's remain in place. The gaps are filled in only after the first non-nan entry.

Args:

inArr (list, numpy array)

Returns:

returns an array of the same size as inArr with the nan-values replaced by the most recent non-nan entry.

9.3 fillwith

quantiacsToolbox.**fillwith**(*field*, *lookup*)

Replaces nan entries of field, with values of lookup.

Args:

field (list, numpy array) : array whose nan-values are to be replaced

lookup (list, numpy array) : array to copy values for placement in field

Returns:

returns array with nan-values replaced by entries in lookup.

9.4 loadData

quantiacsToolbox.**loadData**(*marketList=None*, *dataToLoad=None*, *refresh=False*, *beginInSample=None*, *endInSample=None*, *dataDir='tickerData'*)

Prepares and returns market data for specified markets. Prepares and returns related to the entries in the dataToLoad list. When refresh is true, data is updated from the Quantiacs server. If inSample is left as none, all available data dates will be returned.

Args:

marketList (list): list of market data to be supplied

dataToLoad (list): list of financial data types to load

refresh (bool): boolean value determining whether or not to update the local data from the Quantiacs server.

beginInSample (str): a str in the format of YYYYMMDD defining the begining of the time series *endInSample* (str): a str in the format of YYYYMMDD defining the end of the time series

Returns:

dataDict (dict): mapping all data types requested by dataToLoad. The data is returned as a numpy array or list and is ordered by marketList along columns and date along the row.

9.5 plotts

quantiacsToolbox.**plotts**(*tradingSystem*, *equity*, *mEquity*, *exposure*, *settings*, *DATE*, *statistics*, *returns*, *marketReturns*)

Plots equity curve and calculates trading system statistics

Args:

equity (list): list of equity of evaluated trading system

mEquity (list): list of equity of each market over the trading days

exposure (list): list of positions over the trading days

settings (dict): list of settings

DATE (list): list of dates corresponding to entries in equity

9.6 runts

quantiacsToolbox.**runts**(*tradingSystem*, *plotEquity=True*, *reloadData=False*, *state={}*, *sourceData='tickerData'*)

Backtests a trading system. Evaluates the trading system function specified in the argument tsName and returns the struct ret. runts calls the trading system for each period with sufficient market data, and collets the returns of each call to compose a backtest.

Args:

tsName (str): Specifies the trading system to be backtested

plotEquity (bool, optional): Show the equity curve plot after the evaluation

reloadData (bool,optional): Force reload of market data

state (dict, optional): State information to resume computation of an existing backtest (for live evaluation on Quantiacs servers). State needs to be of the same form as ret

Returns:

a dict mapping keys to the relevant backesting information: *trading system name, system equity, trading dates, market exposure, market equity, the errorlog, the run time, the system's statistics, and the evaluation date*.

keys and description:

'tsName' (str): Name of the trading system, same as tsName

'fundDate' (int): All dates of the backtest in the format YYYYMMDD

'fundEquity' (float): Equity curve for the fund (collection of all markets) 'returns' (float): Marketwise returns of trading system

'marketEquity' (float): Equity curves for each market in the fund

'marketExposure' (float): Collection of the returns p of the trading system function. Equivalent to the percent expsoure of each market in the fund. Normalized between -1 and 1 'settings' (dict): The settings of the trading system as defined in file tsName

'errorLog' (list): list of strings with error messages

'runtime' (float): Runtime of the evaluation in seconds

'stats' (dict): Performance numbers of the backtest 'evalDate' (datetime): Last market data present in the backtest

9.7 stats

quantiacsToolbox.**stats**(*equityCurve*)

Calculates trading system statistics. Calculates and returns a dict containing the following statistics - sharpe ratio - sortino ratio - annualized returns - annualized volatility - maximum drawdown

- the dates at which the drawdown begins and ends
- the MAR ratio
- the maximum time below the peak value (the dates at which the max time off peak begin and end)

Args:

equityCurve (list): the equity curve of the evaluated trading system

Returns:

statistics (dict): a dict mapping keys to corresponding trading system statistics (sharpe ratio, sortino ration, max drawdown...)

9.8 submit

quantiacsToolbox.**submit**(*tradingSystem*, *tsName*)

Submits trading system to Quantiacs server.

Args:

tradingSystem (file, obj, instance): accepts a filepath, a class object, or class instance.

tsName (str): the desired trading system name for display on Quantiacs website.

Returns:

returns True if upload was successful, False otherwise.

9.9 updateCheck

quantiacsToolbox.**updateCheck**()

Checks for new version of toolbox.

Returns:

returns True if the version of the toolbox on PYPI is not the same as the current version returns False if version is the same

CHAPTER 10

Indices and Search

- genindex
- search

CHAPTER 11

Contact Us

Questions, comments, concerns or just want to say hello?

Contact us at office@quantiacs.com

Visit us at Quantiacs

Q