Nyxar Documentation

Release

Huitao Shen, Bo Zeng

Feb 28, 2018

Contents

Nyxar (NyxTrader) is a platform for quantitative trading on 24/7 markets, with modules for data mining, strategy backtesting, paper trading, and live trading. Nyxar is particularly suitable for trading on cryptocurrency markets.

Features:

- Event based backtesting with simulative exchange. Minimal difference between live trading and backtesting algorithms.
- Multiple source of datafeed supported: mainstream cryptocurrency exchanges, quandal, csv file or pandas dataframe.
- Built-in indicators (SMA, EMA, RSI, . . .) for building your own trading strategy.
- Built-in analyzers (Sharpe ratio, drawdown, . . .), cross-validation, and hypothesis testing for benchmarking your strategy.

CHAPTER 1

Table of Contents

1.1 Quotes

1.2 BackExchange

1.2.1 Overview

BackExchange is a simulative exchange for backtesting your trading algorithms. Its API mimics that in ccxt library, which supports live trading on more than 90 mainstrem cryptocurrency exchanges. The intention is to make minimal difference between algorithms in the backtest and in the live trading.

The essential data taken by *BackExchange* are timestamped OHLCV tickers, which are fed through `Quotes` when it is first intialized. Its clock is controlled by the `Timer`. At each time bar, *BackExchange* takes and processes orders just like a real exchange.

Note:

- All tickers and balance are processed and returned with 8 decimal places.
- Order matching is in favor of buyers. For example, if there is a sell order placed at price *10.0* and a buy order placed at *10.5*, the order will be executed at *10.0*.

1.2.2 Features

Order Queue

In event based backtesting, orders placed at this time bar are processed at next, while in live trading orders are processed instantly. Therefore, all orders submitted to *BackExchange* are first cached in an order queue, and wait to be processed at the beginning of the next time bar.

Order queue raises several complication:

- As its name suggests, queued orders are processed in a first-in-first-out manner. This may cause later submitted orders rejected due to insufficient funds.
- Submitted orders can be cancelled through `BackExchange.cancel_submitted_order()`. Cancelled submitted orders will not appear in closed order book, as if order requests are never sent to the exchange.
- When an invalid order is submitted to the order queue, an exception may be raised at the moment when the order is submitted or the moment when the order is processed at the beginning of next time bar, depending on the time that the error can be detected. See *Exceptions* for more details.

See also:

More about *Order*.

List and Delist

A common pitfall in backtesting strategies is survivor bias, which can happen when assets are delisted from an exchange but are not included in the testing data. `BackExchange` supports listing and delisting assets or trading pairs by simply checking existing trading pairs at the current time bar. All currently supported trading pairs and assets can be queried through `BackExchange.fetch_markets()`.

If a previously existing trading pair doesn't exist any more, all open orders under it will be forcely closed. If a previously existing asset doesn't appear in any trading pair, it is considered as delisted. The remaining balance will be forcely withdrawn. Withdrawal history can be queried through `BackExchange.fetch_deposit_history()`.

Slippage

In real life trading, orders are usually not filled at the ticker price for various reasons. This process, called slippage, is taken care in `BackExchange` in the following aspects:

- Orders placed at this time bar is always processed at next to simulate time delay.
- Buy and sell orders can be filled at different type of prices (for example, buy orders are filled at high price and sell orders are filled at low price in the ticker). These can be set when `BackExchange` is first initialized, or changed any time through `BackExchange.buy_price` and `BackExchange.sell_price`.
- Transaction fee as fixed rate slippage. Buy orders are always filled *0.01x%* higher than the ticker price and sell orders are always filled *0.01x%* lower than the ticker price. *x* is the transaction fee rate in the unit of basis point. It can be set when `BackExchange` is first initialized, or changed any time through `BackExchange.fee_rate`.
- Slippage model. Given ticker price and any custom data as input, the slippage model determines the amount and the price to be filled for a given order. It can be set when `BackExchange` is first initialized, or changed any time through `BackExchange.slippage_model`. Nyxar provides several predefined slippage models, such as spread slippage and volume slippage. Nyxar also supports user defined slippage model. See *Slippage Model* for more details.

1.2.3 API Reference

class BackExchange(*timer, quotes[, buy_price=PriceType.Open, sell_price=PriceType.Open, fee_rate=0.05, slippage_model=SlippageBase()*)

BackExchange used for backtesting.

- timer: `Timer` class used to control the clock of BackExchange.
- quotes: `Quotes` class contains timestamped OHLCV tickers.
- buy_price: Set `buy_price`. Defaults to *'open'*.

- sell_price: Set *sell_price*. Defaults to *'open'*.
- fee_rate: Set *fee_rate*. Defaults to 0.05.
- slippage_model: Set *slippage_model*. Defaults to *SlippageBase*.

Attributes:

buy_price
The price types that all buy orders are filled at. Its value can be of one the following four strings: *'open'*, *'high'*, *'low'*, *'close'*.

sell_price
The price types that all sell orders are filled at. Its value can be of one the following four strings: *'open'*, *'high'*, *'low'*, *'close'*.

fee_rate
The fee rate imposed by the exchange on all orders in the unit of basis point. Buy orders are always filled *0.01 * fee_rate%* higher than the ticker price and sell orders are always filled *0.01 * fee_rate%* lower than the ticker price.

In practice, the fee is taken by deducting quote asset for buy orders, and base asset for sell orders. In other words, you will always receive less asset than the amount appears in the order.

slippage_model
The slippage model to determine how an order should be filled. See *Slippage Model* for more details.

User methods:

The following are user methods that resemble public APIs provided by an exchange.

fetch_timestamp()
Return the current timestamp in millisecond.

fetch_markets()
Return a tuple of dictionaries contain currently supported asset names and trading pair symbols.

fetch_ticker([*symbol=''*])
Return the OHLCV tickers of the current time bar for the given *symbol*. If *symbol* not specified, return tickers for all supported symbols.

```
>>> ex.fetch_ticker(symbol='FOO/BAR')
{'open': 1.2, 'high': 3.4, 'low': 5.6, 'close': 7.8, 'volume': 9.0}
>>> ex.fetch_ticker()
{'FOO': {'open': 1.2, 'high': 3.4, 'low': 5.6, 'close': 7.8, 'volume
↪': 900.0},
 'BAR': {'open': 9.0, 'high': 7.8, 'low': 3.5, 'close': 4.6, 'volume
↪': 120.2}, ...}.
```

The following are user methods that resemble private APIs provided by an exchange.

deposit(*asset*, *amount*)

withdraw(*asset*, *amount*)
Deposit / Withdraw *amount* of *asset* into the balance. Any negative *amount* will be cast to zero. Return successfully deposited / withdrawn amount.

fetch_balance()
Return all current balances in a dictionary.

```
>>> ex.fetch_balance()
{'FOO': {'total': 100.0, 'free': 99.5, 'used': 0.5},
 'BAR': {'total': 78.0, 'free': 78.0, 'used': 0}, ...}.
```

fetch_balance_in(*target*[, *fee=False*])
Return the total balance in the *target* asset, based on tickers at the current time bar. The method will automatically finds the most profitable way to convert an asset to *target* if there are more than one ways. A *NotSupported* exception will be raised if there exists an asset that is unable to convert to target.

If *fee=True*, the converted balance is computed by taking transaction fee into account. Defaults to *False*.

fetch_deposit_history()
Return a list of deposit and withdrawl history.

```
>>> ex.fetch_deposit_history()
[{'timestamp': 1517599560000, 'asset': 'FOO', 'amount': 100}, {
↪'timestamp': 1517599620000, 'asset': 'FOO', 'amount': -5}]
```

create_market_buy_order(*symbol*, *amount*)

create_market_sell_order(*symbol*, *amount*)
Create and submit a market buy/sell order under *symbol* of *amount* to the order queue. Return the info of placed order.

```
>>> ex.create_market_buy_order('FOO/BAR', 100)
{'id': 693461813487499546,
'datetime': '2018-02-02 14:26:00',
'timestamp': 1517599560000,
'status': 'submitted',
'symbol': 'FOO/BAR',
'type': 'market',
'side': 'buy',
'price': 0,
'stop_price': 0,
'amount': 100,
'filled': 0,
'remaining': 100,
'transaction': [],
'fee': {}}
```

create_limit_buy_order(*symbol*, *amount*, *price*)

create_limit_sell_order(*symbol*, *amount*, *price*)
Create and submit a limit buy/sell order under *symbol* of *amount* to the order queue. The limit price of the order is *price*. Return the info of placed order.

create_stop_limit_buy_order(*symbol*, *amount*, *price*, *stop_price*)

create_stop_limit_sell_order(*symbol*, *amount*, *price*, *stop_price*)
Create and submit a stop limit buy/sell order under *symbol* of *amount* to the order queue. The limit price of the order is *price*, and the stop limit price is *stop_price*. Return the info of placed order.

cancel_submitted_order(*order_id*)
Cacnel the submitted order in the order queue whose id is *order_id*.

cancel_open_order(*order_id*)
Cancel the open order in the open order book whose id is *order_id*.

fetch_submitted_order(*order_id*)
Return *Order.info* of the submitted order in the order queue whose id is *order_id*.

fetch_submitted_orders ([*limit=500*])
Return *Order.info* of last *limit* submitted orders in the order queue. If *limit=0*, return info of all submitted orders. *limit* defaults to *500*.

fetch_order (*order_id*)
Return *Order.info* of the order whose id is *order_id* in the open order book or closed order book.

fetch_open_orders ([*symbol=''*, *limit=500*])
Return *Order.info* of last *limit* open orders in the open order book. If *symbol* is specified, only orders under that trading symbols are returned. Otherwise all open orders will be returned. If *limit=0*, return info of all open orders. *limit* defaults to *500*.

fetch_closed_orders (*symbol*[, *limit=500*])
Return *Order.info* of last *limit* closed orders in the closed order book. Different from *fetch_open_orders()*, *symbol* must be specified.

1.2.4 Exceptions

exception NotSupported
Raised when an unsupported asset or trading pair symbol is queried.

exception InsufficientFunds
Raised when there are no enough funds to place an order. This exception will only be raised at the beginning of a time bar when the order is being processed by the exchange.

exception InvalidOrder
Raised when an invalid order is submitted. For invalid orders with negaive amount or price, this exception will be raised immediately when orders are created. For invalid orders with non-existing trading pair symbol, this exception will be raised at the beginning of the next time bar.

exception OrderNotFound
Raised when a particular order is not found (usually queried through order id) in the order book.

exception SlippageModelError
Raised when the transaction generated by the slippage model is invalid. For example, for an market order, the *transaction.amount* generated by the slippage model is not equal to *order.amount*.

1.3 Order

1.3.1 Overview

Order is part of the backtest system of Nyxar. Although users do not interact directly with *Order*, it is worthwhile to clarify some properties of it and the role it plays behind the scene.

User's trading algorithms place and query orders in the *BackExchange* through the API therein. *BackExchange* determines whether orders are filled based on the order type and the ticker price. *Transaction* are generated by the *Slippage Model* to determine how (amount and price) orders are filled.

1.3.2 API Reference

class OrderSide (*Enum*)
Enumeration of order side.

Buy

Sell

class OrderType(*Enum*)
Enumeration of order type.

Market
Market order. Market order will be filled as soon as it is accepted. Market order is also all-or-none order, meaning either the order is filled in full or an `InsufficientFunds` exception is raised.

Limit
Limit order. Limit order will only be filled when the ticker price is higher/lower than the limit price for sell/buy orders. Limit order doesn't necessarily need to be filled in full. The filled amount at each time bar is determined the by *Slippage Model*.

Asset in the unfilled part of the order is not available for trading unless the order is cancelled. In order balance can be queried as *used* in :meth::*BackExchange.fetch_balance*.

StopLimit
Stop limit order. Stop limit order will become a limit order when the ticker price is lower/higher than the stop price for sell/buy orders. Note that whether to open a stop limit order is determined only by the ticker price. Slippage model is only effective in filling the order.

class OrderStatus(*Enum*)
Enumeration of order status.

Submitted
Orders submitted to the *Order Queue* of `BackExchange`.

Accepted
Orders accepted by the exchange but is not yet open. It is only applicable to stop limit orders.

Open
Open orders that are not filled yet. It is applicable to limit and stop limit orders.

Filled
Orders that are fully filled.

Cancelled
Orders that are cancelled before fully filled.

class Order(*timestamp*, *order_type*, *side*, *quote_name*, *base_name*, *amount*, *price*, *stop_price*)

timestamp
The timestamp when the order is created.

datetime
A *datetime* object converted from `Order.timestamp`.

id
The unique id of the order that is used to query the order in order queue or order books.

status
An `OrderStatus` object represents the current order status.

type
An `OrderType` object represents order type.

side
An `OrderSide` object represents order side.

quote_name
The name of the quote asset.

base_name
The name of the quote asset.

symbol
The name of the trading pair symbol, which is *quote asset name/base asset name*.

amount
The total amount of the order.

filled
The filled amount of the order.

remaining
The remaining amount of the order. The relation *filled* + *remaining* = *amount* always holds true.

filled_percentage
The filled percentage of the order, computed as *100.0 * filled/amount*.

price
The limit price of the order. Applicable for limit order and stop limit order. For market order, it defaults to *0*.

stop_price
The stop limit price of the order. Applicable for stop limit order. For other order types, it defaults to *0*.

transactions
A list of *Transaction* that accounts for the filled amount of the order.

fee
A dictionary of fees taken by the exchange.

```
>>> order.fee
{'FOO': 0.05}
```

info
A dictionary of order info.

```
>>> order.info
{'id': 4920631724339456104,
 'datetime': '2018-02-02 14:26:00',
 'timestamp': 1517599560000,
 'status': 'filled',
 'symbol': 'FOO/BAR',
 'type': 'limit',
 'side': 'buy',
 'price': 0.000954,
 'stop_price': 0,
 'amount': 100,
 'filled': 100,
 'remaining': 0,
 'transaction': [{'datetime': '2018-02-02 15:44:00', 'timestamp':␣
↪1517604240000, 'price': 0.00095367, 'amount': 100}],
 'fee': {'FOO': 0.05}}
```

open()

accept()

cancel()

generate_transaction(*amount*, *price*, *timestamp*)

execute_transaction(*transaction*)

pay_fee(*asset*, *amount*)

class Transaction(*quote_name*, *base_name*, *price*, *amount*, *side*, *timestamp*)

Attributes of *Transaction* are very similar to those in *Order*. In fact, *Order* is inherited from *Transaction* with more attributes and methods.

timestamp

datetime

id

side

quote_name

base_name

symbol

amount

price

info

```
>>> tx.info
{'datetime': '2018-02-02 15:44:00', 'timestamp': 1517604240000, 'price': 0.
↪00095367, 'amount': 100}
```

1.4 Slippage Model

1.4.1 Overview

In real life trading, orders are usually not filled at the ticker price for various reasons. In order to make backtest results reliable, Nyxar takes slippage into account at various levels (see *Slippage*). In particular, slippage models are responsible to simulate market impact of the order. Nyxar has two builtin slippage models, and users can easily create their own more sophisticated slippage models.

By default, *BackExchange* doesn't use any slippage model. To set up slippage model, assign *BackExchange.slippage_model* to be an instance of slippage model class.

```
from Nyxar import VolumeSlippage, SpreadSlippage, SpreadVolumeSlippage
ex.slippage_model = VolumeSlippage(tradable_rate=2.5)
ex.slippage_model = SpreadSlippage(bidask=data, spread_rate=50)
ex.slippage_model = SpreadVolumeSlippage(bidask=data, spread_rate=50, tradable_rate=2.
↪5)
```

Orders will be automatically processed with the slippage model.

1.4.2 Slippage Models

Volume Slippage

Volume slippage model uses the volume data provided by *BackExchange*. At each time bar, at most *tradable_rate%* of total volume can be filled *per order*. The remaining amount of the order will be processed at next time bar. By default, *tradable_rate=2.5*.

Volume slippage model is only applicable to limit or stop limit orders. Market orders will always be filled in full. Avoid placing large market orders with volume slippage model.

Spread Slippage

Spread slippage model uses additional bid-ask spread data provided by the user through `BidAsks`. All buy/sell orders are filled at price additional *spread_rate% * spread* higher/lower. By default, *spread_rate=50*.

Spread slippage model is applicable to all order types.

Volume-Spread Slippage

Volume-spread slippage is simply a combination of the volume slippage model and spread slippage model.

Custom Slippage

Users can define their own slippage model by defining a child class of *SlippageBase*.

class SlippageBase(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
User should overwrite *__init__()* method doing necessary data feeding or initialization.

generate_tx(*price*, *amount*, *order_type*, *order_side*, *symbol*, *ticker*, *timestamp*)

- *price*: The original tentative ticker price for the order to fill at.
- *amount*: The remaining amount in the order to fill.
- *order_type*: Order type as a *OrderType* enumeration class.
- *order_side*: Order side as a *OrderSide* enumeration class.
- *symbol*: Trading pair symbol of the order.

This method should return a tuple *(tx_price, tx_amount)* which represents a tentative transaction. In the transaction, *tx_amount* is filled at price *tx_price*. *BackExchange* will check if the tentative transaction will actually happen (for example, if *tx_price* is in the range of the limit price of the limit order), and will generate the transaction for you.

However, it is user's responsibility to make sure *(tx_price, tx_amount)* is valid. For example, *tx_amount* == *amount* for market orders. Otherwise *SlippageModelError* will be raised by *BackExchange*.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

O

P

Q

R

S

T

W