Spring Technical Documentation

Release 2.1.0

QuantInsti

Jan 30, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 What is Spring?

Spring is a systematic trading and research platform, powered by the Blueshift from QuantInsti[1]. It allows users to leverage systematic and quantitative approaches in their investing or trading.

Blueshift, the engine driving the platform, is a flexible, event-driven, style agnostic, trading and backtesting framework for developing systematic investment strategies in Python in a fast and reliable way. This makes developing complex (and simple) strategies easy, and moving a strategy from back-testing to live trading seamless.

Note: The platform supports Python 3.6 and newer only. We do not have any planned release for earlier versions of Python.

1.2 What it is NOT

Spring is NOT meant for any kind of high-frequency trading (HFT). Also, Spring is a platform for trading and research. It, in no way, recommends or guarantees the performance of a particular way of trading or investing.

1.3 How Spring Works

The Blueshift engine powering the platform, at its core, consists of an event loop designed to run trading algorithms and a set of API functions to support functionalities required for a trading strategy (e.g. fetching tradable instrument details, placing orders or fetching market data etc.).

A strategy code on the platform does not run directly with the Python interpreter as a regular script would. Instead, the user strategy usually defines a set of event handling functions (or class methods) that determines how the strategy should respond if any event occurs during its run.

The user strategy is read and parsed by the Blueshift engine and is run within its context, in the event loop. The core engine takes care of connecting to the broker and data source(s), running the program and keeping track of the state of the strategy (e.g. trades, orders statuses, profit and loss etc.) automatically. This allows us to focus on the trading logic of the strategy and leave the rest to the engine.

Blueshift core engine, together with user defined strategy, forms a `Complex Event Processing` system (CEP). The Blueshift engine (*engine*), once launched (in either backtesting or live trading mode), does its own initialization and enters the event processing loop. In this event loop, it continuously looks for incoming events (e.g. new market data

[1] Blueshift and QuantInsti are registered trademarks of QuantInsti Quantitative Learnings Pvt Ltd.

arrival or an order fill event etc.), updates its own state and calls the appropriate handler(s) defined by the user strategy at each loop iteration.

The strategy codes (*strategy*, a Python script or module) written by users define a set of *callback functions* that are called by the engine when the specific event occurs. In addition, users can also use the *scheduling functions* for finer control.

User defined *strategy* has no direct access to the event loop or its internal variables. Instead, all interactions with the algo are achieved through a set of `API functions` (detailed below) and by exposing two special variables -*context* and *data*. All data queries are directed through this `data` object. All algo state related queries (e.g. the cash in the account, current positions etc.) are directed through the `context` object.

This simple event loop architecture allows blueshift to run any strategies that can be coded, while keeping the interface between the backtesting and live trading consistent.

1.4 Platform APIs Overview

The APIs available on the platform can be divided into below broad categories.

- **Callback functions**
  The callback functions are the entry points to the algo event loop. These are based on the lifetime events of the strategy. See *callback APIs* for more on these. All of these callback functions accept the `context` variable as the first argument and the `data` variable as the second argument (with the exception of *initialize* and *analyze*).
- **Scheduling functions**
  Scheduling functions allow finer control of event handling by precisely defining repetitive or one-time event handlers. See *schedule APIs* for more on these. These event handlers also accept two variables - `context` and `data`.
- **Asset and Data fetching APIs**
  Use the `current` and `history` method of the *data* object to fetch current and historical data. Use the `symbol` function (see *assets*) to fetch supported assets.
- **Trading APIs**
  Use the base `order` function, or its derivatives (like `order_target` and `order_target_percent`) for placing orders. Use the `cancel_order` to cancel an open order. Use `get_open_orders` to fetch current open orders. For more details, see the *trading APIs*.
- **Pipeline APIs**
  Use pipelines for dynamically selecting and filtering the very large asset universe. See *pipeline APIs* for more details.
- **Risk management and other APIs**
  A host of risk management APIs to put constraints on the running algo - see *risk management APIs* for details. Fine tune backtest runs using *backtest model selection APIs*. Refer to *miscellaneous APIs* for various useful functions like `get_datetime` and `record`.
- **Context and Data Objects Attributes**
  Use the *context object methods* for querying the state of the running algo (e.g. querying *current portfolio and account* states). Use the *Data Object* methods for fetching current or historical data.

Danger: Backtesting and especially automated trading is far from fire-and-forget. Many things can go wrong (although we try our best to catch and highlight such cases by design). Also there are things beyond control of the platform that can go wrong. Like the strategy code going rogue or the datafeed being corrupt or the connectivity unstable. Be careful and remember to take appropriate precautions.

CHAPTER

TWO

CHANGELOG

2.1 Release 2.1.0 (Jan 16, 2022)

Highlights from release 2.1.0 are listed below

2.1.1 Breaking Changes (2.1.0)

- Realtime callback function `on_data` is now upgraded as a main callback function. Instead of adding a handler using the said function, use the function directly to handle arriving events. This also applies to `on_trade` as well. As a consequence, functions to remove such event handlers (`off_data` and `off_trade`) are now deprecated and will throw errors.
- All imports from `blueshift_library` are now deprecated and will throw errors. Point all such imports to `blueshift.library` instead.
- NSE futures and options now include `lotsize` information (previously was 1). Since assets can be traded in multiple of lotsizes only (if not 'fractional'), the trading capital must be high enough for an order to go through. For e.g. you could trade 1 unit of NIFTY futures before, but now minimum order size must be 50 (or whatever the current lotsize is).
- Error handling changes: Repeated errors will now cause the algo to exit after a threshold is breached.

2.1.2 New Features (2.1.0)

- Additional event scheduling functions `schedule_once` and `schedule_later`. See *schedule APIs* for more.
- Support for advanced algo order type. See *Advanced Algo Orders* for more.
- Option backtesting is now fully supported (for NSE dataset only).
- Built-in support for stoploss and take profit behaviour. For more on this see *Stoploss and Take-profit*.
- Add backtest support for *order modification*.
- Enriched *Order* object - now includes latency metrics as well as parent algo ID.

2.1.3 Bug Fixes and Improvements (2.1.0)

- Fix reconciliation error in case broker streaming update disconnects (force an API call).

2.2 Release 2.0.0 (Nov 2, 2021)

Warning: Blueshift 2.0 includes breaking change. See below for more details.

The new Blueshift engine improves on several aspects of strategy development. Following are major highlights

- **Breaking Change** - move away from zipline to Blueshift Python engine.
- **Breaking change** - Multi-assets, multi-fields history method now returns MultiIndex dataframes.
- **Breaking change** - Attributes for various objects (e.g. `asset`, `position`, `account`, `portfolio` etc.) may have different names. See the documentation for the attribute names to use.
- **Breaking change** - The `performance` argument to `analyze` function has new values. See the documentation for more details.
- Extended liquidity filtered universe for the US and Indian markets.
- Realistic simulation with explicit margin models.
- Realistic models for margin and slippage.
- Historical bid-ask slippage modelling of forex backtest.
- **Pipeline API** support in live trading. See *Pipeline APIs* for more details.
- **Fractional trading** - to support fractional trading, the data-type for asset quantity wherever applicable (e.g. *order.quantity* or *position.quantity*) are now `float` type instead of `int`.

2.2.1 Breaking Changes

Update your strategies to point to `blueshift` in all places that currently imports from `zipline`, as shown below

```
#### change below ....
#from zipline.finance import slippage, commission
#from zipline.api import symbol
#from zipline.pipeline import Pipeline
#### to point to blueshift
from blueshift.finance import slippage, commission
from blueshift.api import symbol
from blueshift.pipeline import Pipeline
```

Update your strategies to adapt to Pandas MultiIndex dataframes (if required). Otherwise, it may crash. See *Upgraded Data Interface*.

```
from blueshift.api import symbol

def initialize(context):
    context.universe = [symbol('AAPL'), symbol('MSFT')]

```

(continues on next page)

(continued from previous page)

```
def handle_data(context, data):
    # a multi-asset, multi-field history call
    prices = data.history(context.universe,['close','volume'],10,'1m')

    # price for AAPL
    # this is deprecated
    # appl_price = prices.minor_xs(context.universe[0])
    # use below to replace minor_xs calls.
    appl_price = prices.xs(context.universe[0])
```

For other breaking changes, carefully analyse your existing strategy code and refer to the respective documentation for supported attribute names and performance columns. If you have trouble, feel free to contact us at blueshift-support@quantinsti.com. For more on Pandas MultiIndex, see the official documentation

2.2.2 Changes in Blueshift 2.0.0

Blueshift 2.0 release moves away from `zipline` and introduces major upgrades in many areas, while retaining the existing APIs unchanged. A few improvements are highlighted below:

Extended and Updated Asset Universes

We move to a new liquidity based universe (compared to external benchmark based universe earlier) with assets size substantially expanded.

For both the US and Indian markets, we have extended the universe to include liquid ETFs. Each of these universes now tracks top 1000 liquid instruments (equities and ETFs) that trades above a certain price thresholds (avoiding penny stocks). In addition, for the Indian markets, all NSE equity futures instruments are included as well. The forex universe remains unchanged.

More Realistic Simulation Behaviour

The simulation engine is re-written to be more realistic with the introduction of an explicit *margin model*.

The default US execution now is modelled on a `Regulation T` type margin account - with 50% flat margin requirements (That is, 2x buying power). This avoids the earlier (somewhat strange) behaviour of unlimited (infinite) buying powers. For example, using a $5,000 account to take a position of more than $10,000 worth of net exposure will now throw an `insufficient fund` error and will stop the backtest, instead of continuing with unrealistic assumptions.

The execution model for the Indian market now assumes cash transactions in the equity/ ETFs segment and margin transactions in the futures segment (with default 10% `flat margin`). Forex is also explicitly modelled as margin trading with default 5% flat margin.

The default *slippage* for equities (and futures) continue to be Volume based slippage (a very common model). For forex, however, we have moved to a historical bid-ask based slippage model. This avoids the high sensitivity of forex backtest to trading cost assumptions and produces more accurate results.

In addition, we have introduced stringent and up to the minute checks for backtest account solvency. If your backtest equity is in the negative on a close of business (i.e. a margin call from the broker), the backtest will stop. In such cases, you should reduce the leverage level in your strategy and try to run again.

A couple of things you still need to be careful while testing your strategy

1. The Reg T account does not track or flag `pattern day trading`.

2. The cash equities account `does not enforce long-only`. This means on a short sell, it will credit the whole transaction value to the backtest account. You can enforce long-only behaviour by using the `set_long_only` API method.

Changes in Ordering function

With the introduction of the margin model, we have also introduced a 2% margin for `order_target_percent`. This now calculates the target value based on current portfolio value, after a 2% haircut. This is applicable only for *MARKET* orders. For LIMIT orders, no such haircut will be applied.

Also, all targeting type *order functions* will now take into account the pending amount of orders in that asset. This is in addition to considering the existing position in that asset.

Upgraded Data Interface

The *data.history* method, for multiple assets and multiple fields, will now return `Pandas MultiIndex Dataframe` (instead of the deprecated *Panel* data). The required changes in your strategy code is very minimal. Instead of accessing the data for a given asset using *prices.minor_xs(asset)*, you need to use *prices.xs(asset)*.

Using MultiIndex dataframes over Panel data has several advantages (apart from the obvious deprecation). The input data for different assets now need not be homogeneous. This is a more realistic scenario as it does not enforce an unnatural requirement of every asset in the universe to have a trade every minute of every business day.

CHAPTER

THREE

EVENT CALLBACKS IN STRATEGY

Blueshift is an event-driven engine. The core engine implements the event loop. The user strategy implements a set of callback functions (i.e. event handlers) that are called by the event loop at defined events. The whole strategy logic is structured by responding to events as and when they occur. Blueshift strategies can respond to three types of events during its run.

- **Lifetime Events**
  These are events generated by the Blueshift event loop based on the lifetime epochs of the strategy. They are summarised under the *Main Callback Functions* below.
- **Trade and Data Events**
  These are events that are triggered once an order is (potentially partially) filled or on arrival of new data points. They work differently in backtest or real-time trading. See *Trade and Data Callbacks* below.
- **Time-based Events**
  These are events scheduled by the user (using one of the *scheduling* functions). These are summarised under the *Scheduled Callback Functions* section.

3.1 Main Callback Functions

Lifetime event handlers are based on the epoch of the strategy run. This includes the following:

3.1.1 Initialize

`initialize`(*context*)

This is the first entry point function. `initialize` is called when at the beginning of an algorithm run, only once. For backtest, the call time is at midnight of the start date of the backtest run. For live trading, this is called at the start of the execution as soon as possible.

Parameters
`context` (*context* object.) – The algorithm context.

User program should utilise this function to set up the algorithm initialization (parameters, trading universe, function scheduling, risk limits etc).

Warning: A valid strategy script/ module must have this function defined.

3.1.2 Before Trading Start

`before_trading_start`(*context*, *data*)

This function is called at the beginning of every trading session (day). For backtest, this is called 30 minutes before the regular market open hour everyday. For live trading, this is called 30 minutes before the market opens each day, or just after (around a minute) of the `initialize` call at the start on the execution start day (in case it is already trading hours or less than 30 minutes remaining from the opening hour).

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`data`** (*data* object.) – The algorithm data object.

User program should utilise this function to set up daily initialization (e.g. profit and loss, model re-evaluation etc.)

Warning: User strategy should not depend on the exact time of invocation of this handler function. The only thing that is guaranteed is that it will be called only once per session and will be called before any handle data or scheduled functions.

3.1.3 Handle Data

`handle_data`(*context*, *data*)

This function is called at every clock beat, i.e. every iteration of the `event loop` (usually at every minute bar).

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`data`** (*data* object.) – The algorithm data object.

User program should utilise this function to run their core algo logic if the algorithm needs to respond to every trading bar (every minute). For algorithms (or functions) that need to respond at a lower scheduled frequency, it is more efficient to use the *scheduling* API function to handle such cases.

Note: If the clock frequency is one minute (which is the case at present on Blueshift), this function is equivalent to a scheduled function with `time_rule` as `every_nth_minute(1)`.

3.1.4 After Trading Hours

`after_trading_hours`(*context*, *data*)

This function is called at the end of every session (every day) around 5 minutes after the last trading minute (market close).

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`data`** (*data* object.) – The algorithm data object.

User program should utilise this function to do their end-of-day activities (e.g. update profit and loss, reconcile, set up for next day).

Warning: User strategy should not depend on the exact time of invocation of this handler function. The only thing that is guaranteed is that it will be called only once per session and will be called after the end of the regular market hours.

3.1.5 Analyze

`analyze`(*context*, *performance*)

This function is called only once, at the end of an algorithm run, as soon as possible.

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`performance`** (*`pandas.DataFrame`*) – The algorithm performance.

Note: The `performance` object is a DataFrame with algo performance captured at daily intervals. The dataframe timestamps are the timestamps for `after_trading_hours` calls. The columns include some of the algo account fields as well the profit and loss (pnls) metrics.

User program can implement this method to add custom analysis of backtest results.

3.2 Trade and Data Callbacks

Attention: only available for live trading, for brokers supporting streaming for data and trade updates.

These callback APIs allows the user strategy to respond to the markets events as it happens (in real-time for live mode).

3.2.1 On Data

`on_data`(*context*, *data*)

This event handler is invoked at arrival of new data points. This function is called at every clock tick, i.e. every iteration of the `event loop` (usually at every minute bar) in backtest mode. In realtime mode, this is invoked only if the broker supports real-time data streaming and a new data point has been made available. In the case of the latter, only instruments that the algo has subscribed to can trigger a call to this function.

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`data`** (*data* object.) – The algorithm data object.

Important: The *on_data* event handler is called only when triggered by the underlying broker. For that to happen, the broker must support streaming data (market data websockets or socketIO API), as well as the strategy must have subscribed to at least one instrument for streaming data. There is no explicit API to subscribe to streaming data, but *curret* or *history* will automatically trigger data subscriptions for the queried assets. If you are using this handler, make sure you have triggered a call to one of these methods to enable subscription. Without it, the handler will never be called.

Warning: The callback function must be short and quick to avoid creating a backlog. Adding long running funtions may lead to the algo crashing. Also, a temporary network disconnection will cause this event handler to stop triggering (till the connection is restored). To make sure your strategy logic is robust, you can also put the same logic inside *handle_data* to make sure the strategy logic is triggered at least once every minute, even if there is a disconnection. Of course, this depends on the specific need of a given strategy.

See also:

See available callback types `blueshift.api.AlgoCallBack`.

3.2.2 On Trade

`on_trade`(*context*, *data*)

This event handler is invoked when an order placed by the strategy is filled. This function is called when the simulator fills an order in backtest mode or paper trading mode. In realtime mode, this is invoked only if the broker supports real-time order update streaming and an order has been filled.

Parameters

- **`context`** (*context* object.) – The algorithm context.
- **`data`** (*data* object.) – The algorithm data object.

See also:

See available callback types `blueshift.api.AlgoCallBack`.

Warning: The callback function must be short and quick to avoid creating a backlog. Adding long running funtions, may lead to the algo crashing.

3.3 Scheduled Callback Functions

The scheduled event callbacks provide a way to schedule a callback function (with signature `func(context, data)`) based on a date and time based rule. There is no limit on how many callbacks can be scheduled in such a manner. But only one callback can be scheduled at each call of the `schedule_function`.

3.3.1 Schedule Once

`TradingAlgorithm.schedule_once`(*callback*)

Add a callback to be called once at the next event processing cycle. The callback must have the standard call signature of `f(context, data)`. The handler will be called in the next clock event during regular market hours, as soon as possible.

Parameters

`callback` (*`callable`*) – Callback function to run.

Note:

- This schedules the callback to run one time only. For repetitive callbacks, use `schedule_function` below
- The callback can use this function to schedule itself recursively, if needed.

3.3.2 Schedule Later

`TradingAlgorithm.`**`schedule_later`**`(`*callback*, *delay*`)`

Add a callback to be called once after a specified delay (in minutes). The callback must have the signature `f(context, data)`. The callback will be triggered during the market hour only.

Parameters

- **`callback`** (*`callable`*) – Callback function to run.
- **`delay`** (*`number`*) – Delay in minutes (can be fractional).

Note:

- This schedules the callback to run one time only. For repetitive callbacks, use `schedule_function` below
- The callback can use this function to schedule itself recursively, if needed.
- You can use a fractional number to run a function at higher frequency resolution than minute. For example specifying delay=0.1 will run the callback after 6 seconds. This is only applicable for live runs. For backtests, it will fall back to one minute minimum. The minimum delay that can be specified is 1 second.

Warning: There is no guarantee the function will be called at the exact delay, but if it is called, it will be called at least after the specified delay amount.

3.3.3 Schedule Function

`TradingAlgorithm.`**`schedule_function`**`(`*callback*, *date_rule=None*, *time_rule=None*`)`

Schedule a callable to be executed repeatedly by a set of date and time based rules. Schedule function can only be triggered during trading hours. The callable in the schedule function will be run before *handle_data* for that trading bar. The callback must accept two arguments - *context* and *data*.

Parameters

- **`callback`** (*`function`*) – A function with signature `f(context, data)`.
- **`date_rule`** (see *`blueshift.api.date_rules`*) – Defines schedules in terms of dates.
- **`time_rule`** (see *`blueshift.api.time_rules`*) – Defines schedules in terms of time.

Warning:

- This method can only be used within the *initialize* function. Attempting to set a scheduled callback anywhere else will raise error and crash the algo.
- The offset should be meaningful and always non-negative. For e.g. although the hours offset can be maximum 23, using such an offset is not meaningful for shorter trading hours (unless it is a 24x7 market).
- In live trading, there is no guarantee that the scheduled function will be called at exactly at the scheduled date and time. It may be delayed if the algorithm is busy with some other function. The function is guaranteed to be called no sooner than the scheduled date and time, and as soon as possible after that.

Date Rules

`class blueshift.api.date_rules`

Date rules define the date part of the rules for a scheduled function call. The supported functions are as below (further subjected to time rule). The `days_offset` parameter below (if applicable) must be `int` (positive), and it must not be greater than 3 for `week_start/week_end` and must not be greater than 15 for `month_start` and `month_end`.

- `every_day()`: called every day.
- `week_start(days_offset=0)`: *days_offet* days after the first trading day of the week.
- `week_end(days_offset=0)`: *days_offet* days before the last trading day of the week.
- `month_start(days_offset=0)`: *days_offet* days after the first trading day of the month.
- `month_end(days_offset=0)`: *days_offet* days before the last trading day of the month.
- `on(dts)`: called every day in the list *dts* (must be list of pandas Timestamps or DatetimeIndex).

Time Rules

`class blueshift.api.time_rules`

time rules defines the time part of the rules for a scheduled function call. The supported functions are as below (further subjected to date rule).

The `hours` parameter below (if applicable) must be `int`, (positive) and it must not be greater than 23. The `minutes` parameter below (if applicable) must be `int` (positive) and must not be greater than 59.

- `market_open(minutes=0, hours=0)`: called after `hours` and `minutes` offset from market open.
- `on_open(minutes=0, hours=0)`: Alias for market_open.
- `market_close(minutes=0, hours=0)`: called after `hours` and `minutes` offset before market close.
- `on_close(minutes=0, hours=0)`: Alias for market_close.
- `every_nth_minute(minutes=1)`: called every n-th minute during the trading hours.
- `every_nth_hour(cls, hours=1)`: called every n-th hour during the trading hours.
- `every_hour()`: called every hours during the trading day.
- `at(dt)`: Called at the given time *dt* (must be a datetime.time object).

3.4 Scheduling Examples

3.4.1 Repetitive Logic with Scheduling

The below code shows examples of setting up a monthly callback function, to be called on the first business day of each month, 30 minutes before the market close.

```python
from blueshift.api import schedule_function, date_rules, time_rules
from blueshift.api import get_datetime

def initialize(context):
    schedule_function(myfunc,
                      date_rule=date_rules.month_start(),
```

(continues on next page)

(continued from previous page)

```
                              time_rule=time_rules.market_open(minutes=30))

def myfunc(context, data):
    print(f'scheduled function called at {get_datetime()}')
```

3.4.2 Responsive Strategy with Scheduling

The below code shows examples of placing a limit order and then updating the order to optimize time to fill and fill price. The algo terminates once the order is executed.

```
from blueshift.api import schedule_later, schedule_once, symbol, terminate
from blueshift.api import terminate, order, update_order, get_order

def initialize(context):
    context.asset = symbol('AAPL')
    context.order_id = None
    schedule_once(myfunc) # call as soon as ready

def myfunc(context, data):
    if context.order_id is None:
        context.order_id = order(context.asset, 1)
        if not context.order_id:
            raise ValueError(f'something went wrong.')

        schedule_later(myfunc, 1) # call again after one minute
        return

    o = get_order(context.order_id)
    if o.is_open():
        px = data.current(context.asset, 'close')
        update_order(context.order_id, price=px)
        schedule_later(myfunc, 1) # call again after one minute
    else:
        # we are done
        terminate(f'order executed, terminate now.')
```

If you are using `schedule_once` or `schedule_later` recursively, carefully follow your logic and make sure the recursion ends where it needs to.

CHAPTER

FOUR

FETCHING PRICE DATA, TRACKING ALGO STATE

Blueshift callback functions are usually called with one or two arguments - `context` and `data`. These are internally maintained objects that provide useful functionalities to the algo. User strategy can use the *context* object to query about the current state of the algo, including its profit-loss, positions, leverage or other related information. For fetching price data, the *data* object is used. See below for more.

Note: The `context` and `data` variables are maintained by the platform and are made available to most callback APIs automatically. You need not instantiate these objects on your own. Also do not override their internal methods, else the algo will crash with error.

4.1 Context Object

The Blueshift engine uses the internal `context` object to track and keep the various state metrics of the running strategy up-to-date.

The `context` object is an instance of an internal class as below. User strategy code never instantiates this object. This object is created and maintained by the platform core engine and provides interfaces to the current context of the running algorithm, including querying the current `order` status and `portfolio` status. The `context` object is the first (and sometimes the only) argument to all platform callback functions.

`class blueshift.algorithm.context.AlgoContext`

The algorithm context encapsulates the context of a running algorithm. This includes tracking internal objects like blotter, broker interface etc, as well as account, portfolio and positions details (see below). The context object is also useful to store user-defined variables for access anywhere in the strategy code.

Warning: Once the context is initialised, its core attributes (i.e. non-user defined attributes) are read-only. Attempting to overwrite them will throw `AttributeError` and will crash the algo.

See also:

create and use variables

4.1.1 Context Attributes

`AlgoContext.`**`name`**

return the name (`str`) of the current algo run.

`AlgoContext.`**`mode`**

return the run mode (`enum`) of the current run.

See also:

see *Algo Modes and Other Constants* for allowed values and interpretation.

`AlgoContext.`**`execution_mode`**

return the execution mode (`enum`) of the current run.

See also:

see *Algo Modes and Other Constants* for allowed values and interpretation.

`AlgoContext.`**`trading_calendar`**

Returns the current trading calendar object.

See also:

See documentation for *Trading Calendar*.

`AlgoContext.`**`record_vars`**

The recorded var dataframe (`pandas.DataFrame`) as generated by a call to the API function `record`. The column names are recorded variable names. Variables are recorded on a per-session (i.e. daily) basis. A maximum of 10 recorded variables are allowed.

Warning: Adding recorded variables may slow down the speed of a backtest run.

See also:

See details in *`blueshift.algorithm.algorithm.TradingAlgorithm.record`*.

`AlgoContext.`**`pnls`**

Returns historical (daily) profit-and-loss information since inception. This is a `pandas.Dataframe` with the following columns:

- algo_returns: daily returns of the strategy
- algo_cum_returns: cumulative returns of the strategy
- algo_volatility: annualised daily volatility of the strategy
- drawdown: current drawdown of the strategy (percentage)

Note: The timestamp for each day is the end-of-day, except the current day with the timestamp of most recent computation.

`AlgoContext.`**`orders`**

return a list of all open and closed orders for the current blotter session. This is a `dict` with keys as order IDs (`str`) and values as *order* object.

`AlgoContext.`**`open_orders`**

return all orders currently open from the algorithm. This is a `dict` with keys as order IDs (`str`) and values as *order* object.

4.1.2 Portfolio and Account

The `context` objects provides interfaces to algo account and portfolio through attributes `context.account` and `context.portfolio` accessible from the user strategy.

`AlgoContext.`**`account`**

Return the account object (a view of the underlying trading account).

The account object has the following structure. All these attributes are read-only.

Attribute	Type	Description
margin	`float`	Total margin posted with the broker
leverage	`float`	Gross leverage (gross exposure / liquid asset value)
gross_leverage	`float`	Gross leverage (gross exposure / liquid asset value)
net_leverage	`float`	Net leverage (net exposure / liquid asset value)
gross_exposure	`float`	Gross (unsigned sum) exposure across all assets at last updated prices
long_exposure	`float`	Total exposures in long positions
short_exposure	`float`	Total exposures in short positions
long_count	`int`	Total assets count in long positions
short_count	`int`	Total assets count in short positions
net_exposure	`float`	Net (signed sum) exposure across all assets at last updated prices
net_liquidation	`float`	Sum of cash and margin
commissions	`float`	Net commissions paid (if available)
charges	`float`	Net trading charges paid (if available)
total_positions_exposure	`float`	Gross (unsigned sum) exposure across all assets at last updated prices
available_funds	`float`	Net cash available on the account
total_positions_value	`float`	Total value of all holdings

Warning: Running multiple strategies in the same account may lead to misleading values of these attributes.

`AlgoContext.`**`portfolio`**

Return the current portfolio object. Portfolio is a view of the current state of the algorithm, including positions.

The attributes (read-only) of the portfolio object are as below:

Attribute	Type	Description
portfolio_value	`float`	Current portfolio net value
positions_exposure	`float`	Present gross exposure
cash	`float`	Total undeployed cash
starting_cash	`float`	Starting capital
returns	`float`	Cumulative Algo returns
positions_value	`float`	Total value of holdings
pnl	`float`	Total profit or loss
start_date	`Timestamp`	Start date of the algo
positions	`dict`	Positions dict (see below)

The `positions` attribute is a dictionary with the current positions. The keys of the dictionary are *Asset* objects. The values are *Position* objects.

The following example shows how to access account and positions data within the strategy code.

```
def print_report(context):
    account = context.account
    portfolio = context.portfolio
    positions = portfolio.positions

    for asset in positions:
        position = positions[asset]
        print(f'position for {asset}:{position.quantity}')

    print(f'total portfolio {portfolio.portfolio_value}')
    print(f'exposure:{account.net_exposure}')

def before_trading_starts(context, data):
    print_report(context)
```

4.2 Data Object

The `data` object is the second argument to platform callback functions (where applicable). This provides an interface to the user strategy to query and fetch data.

4.2.1 Fetching Current Data

`class blueshift_data.readers.data_portal.`**`DataPortal`**

DataPortal class defines the interface for the `data` object in the callback functions. It defines two basic methods - `current` and `history`. User strategy should use these methods to query and fetch data from within a running algo.

`abstract current`(*assets*, *columns*)

Return last available price. If either assets and columns are multiple, a series is returned, indexed by assets or fields, respectively. If both are multiple, a dataframe is returned. Otherwise, a scalar is returned. Only OHLCV column names are supported in general. However, for futures and options, `open_interest` is supported as well.

Parameters

- **assets** (*asset* object or a `list` of assets.) – An asset or a list for which to fetch data.
- **columns** (`str` or a `list`.) – A field name or a list of OHLCV columns.

Returns
current price of the asset(s).

Return type
`float` (`int` in case of volume), `pandas.Series` or `pandas.DataFrame`.

> **Warning:** The data returned can be a `NaN` value, an empty series or an empty DataFrame, if there are missing data for the asset(s) or column(s). Also, the returned series or frame may not contain all the asset(s) or column(s) if such asset or column has missing data. User strategy must always check the returned data before further processing.

4.2.2 Querying Historical Data

`class blueshift_data.readers.data_portal.DataPortal`

DataPortal class defines the interface for the `data` object in the callback functions. It defines two basic methods - `current` and `history`. User strategy should use these methods to query and fetch data from within a running algo.

`abstract history`(*assets*, *columns*, *nbars*, *frequency*)

Returns given number of bars for the assets. If more than one asset or more than one column supplied, returns a dataframe, with assets or fields as column names. If both assets and columns are multiple, returns a multi-index dataframe with columns as the column names and asset as the second index level. For a single asset and a single field, returns a series. Only OHLCV column names are supported. However, for futures and options, `open_interest` is also supported.

Parameters

- **assets** (*asset* object or a `list` of assets.) – An asset or a list for which to fetch data.
- **columns** (`str` or a `list`.) – A field name or a list of OHLCV columns.
- **nbars** (*`int`*) – Number of bars to fetch.
- **frequency** (*`str`*) – Frequency of bars (either '1m' or 1'd').

Returns
historical bars for the asset(s).

Return type
`pandas.Series` or `pandas.DataFrame`.

> **Warning:** The data returned can be an empty series or an empty DataFrame, if there are missing data for the asset(s) or column(s). Also, the returned series or frame may not contain all the asset(s) or column(s) if such asset or column has missing data. In addition, for multi-indexed DataFrame, user strategy code must not assume aligned data with same timestamps for different assets (however, columns will always be aligned for a given asset). User strategy must always check the returned data before further processing.

Changed in version 2.1.0: - The *frequency* parameter now supports extended specifications, in addition to (*1d* and *1m*). This can be added as Pandas frequency format, e.g. *5T*, *30T* or *1H* for 5-minute, 30-minute and 1-hour candles respectively. This is primarily designed for live trading and using this in backtest can be very slow (as the underlying data is stored only in minute or daily format and other frequencies are resampled on-the-fly). The allowed parameters in live trading depends on the broker support, i.e. will fail if the broker does not support the particular candle frequency.

Important:

- These methods support the OHLCV ("open","high","low","close" and "volume") columns for all assets (except options), for backtests as well as live trading. For non-tradable assets (e.g. market index), "volume" may be zeros or missing. For options, open, high, low and volume fields are not available (see below for extra fields).
- For backtest, apart from OHLCV columns, "open_interest" is also available for futures and options assets. This may not be available in live trading if the broker supports streaming data but does not support open interest in streaming data.
- For backtest, when options asset(s), we can specify greeks, implied vol and ATMF forward as field names as well. Use "implied_vol" for the implied volatility levels and "atmf" for the prevailing at-the-money futures level. The supoprted greeks are "delta", "vega", "gamma" and "theta". The greek levels are computed using Black 76 model, on-the-fly, and may not be stable near the expiry. These fields may not be available in live trading (but user strategy can *compute* them easily on-the-fly).
- These methods will handle a rolling asset specifications by picking the correct asset at each timestamp for which data is returned. For example, querying for *symbol('NIFTY-ICE+100')* for last 20 minutes will return the 100 out ATM call as applicable for each of the minutes (although the underlying levels, and hence the actual strikes may vary).

The following example shows how to access current and historical data and what are the expected data types of the return values in various cases.

```python
from blueshift.api import symbol

def initialize(context):
    context.universe = [symbol("AAPL"), symbol("MSFT")]

def before_trading_start(context, data):
    # this returns a float value
    px = data.current(context.universe[0], 'close')

    # this returns an int value
    px = data.current(context.universe[0], 'volume')

    # this returns a pandas.Series with the columns in the index
    px = data.current(context.universe[0], ['open','close'])

    # this returns a pandas.Series with the assets in the index
    px = data.current(context.universe, 'close')

    # this returns a pandas.DataFrame with assets in the index
    px = data.current(context.universe, ['open','close'])

    # px1 is a Series with timestamps as index
    px1 = data.history(context.universe[0], "close", 3, "1m")

    # px2 is DataFrame with timestamp index and field names as columns
    px2 = data.history(context.universe[0], ['open','close'], 3, "1m")

```

(continues on next page)

(continued from previous page)

```
    # px3 is a DataFrame with timestamp index and assets as columns
    px3 = data.history(context.universe, "close", 3, "1m")

    # px4 is a multi-index Frame with field names as columns, asset as the second index␣
↪level
    px4 = data.history(context.universe, ["open","close"], 3, "1m")

    # to fetch all fields for an asset, use `xs`
    # this returns a regular Dataframe with columns as field names
    asset_o_price = px4.xs(context.universe[0])

    # to fetch a field for all assets, use subsetting
    # this returns a regular Dataframe with columns as assets
    close_prices = px4['close']
```

Blueshift manages data in multiple layers. Actual raw data is stored internally, through a class named `DataStore`. The core implementation of the `DataStore` class uses . The `DataStore` class provides low-level APIs to read and write data (from disk or a streaming source such as a websocket, or from an in-memory object). A high-level class `Library` handles (potentially multiple) `DataStore` instances with defined dispatch functionalities (to route a data query to appropriate `store`, among many). This `Library` class implements the `DataPortal` interface above. The algo simulation queries this `Library` instance to fetch `current` data or data `history`.

CHAPTER

FIVE

PLACING ORDERS AND OTHER API FUNCTIONS

Note: Most API functions in Blueshift are implemented as methods of the main algorithm class `TradingAlgorithm`. In the documentation below, if you see a function is documented as `TradingAlgorithm.funcname`, then the API function `funcname` can be usually imported in the strategy code from `blueshift.api` module (as `from blueshift.api import funcname`) and can be used as regular function. A reference to the current running algorithm is automatically inserted.

5.1 Assets Fetching APIs

Blueshift treats securities as *asset* objects of various kinds. To refer to a tradable security you must use an asset object. The API function `symbol` converts a security symbol to an asset object.

`TradingAlgorithm.`**`symbol`**`(`*symbol_str: str*, *dt=None*, **args*, ***kwargs*`)`

API function to resolve a symbol string to an asset.

Parameters

- **`symbol_str`** (*`str`*) – The symbol of the asset to fetch.
- **`dt`** (*`timestamp`*) – Optional datetime argument.

Returns

the *asset* object.

Raises

`SymbolNotFound` exception if no matching asset found.

`TradingAlgorithm.`**`symbols`**`(`*symbols*, *dt=None*, **args*, ***kwargs*`)`

API function to resolve a list of symbols to assets.

Parameters

`symbols` (*`list`*) – The list of symbols.

Returns

A `list` of *asset* objects.

Raises

`SymbolNotFound` exception if no matching asset found.

`TradingAlgorithm.`**`get_dated_asset`**`(`*asset*`)`

API function to fetch dated asset for a rolling asset.

Parameters

`asset` (*asset* object.) – asset to convert to corresponding dated asset.

For more on assets, see *assets*. For more on how to use symbols to refer to available instruments, see *asset symbology*.

Important: The *symbol* function will resolve a ticker name or asset symbol to an asset object representation applicable for the time at which the method was called. For example, *symbol("NIFTY-W0CE+0")* will refer to the first the current weekly ATM call at the time when the method were called. Calling the same method later can refer to a different instruments (e.g. when the underlying has moved and the ATM strike is now at a different level than earlier).

5.2 Trading API functions

Warning: While Blueshift supports running multiple strategies under a single account, this may lead to erroneous results. We recommend running a single algo on a broker account in live or paper trading.

5.2.1 Order Placing APIs

Attention: All ordering APIs can only be used during active trading hours. Else they will raise exceptions. Also, do not use the undocumented parameters in the below APIs. They are meant for internal use.

Note: Blueshift now supports fractional trading, i.e. order quantity need not be an integer and fractional amount is supported (if supported by the broker). Some brokers (e.g. Crypto brokers) offer exclusively fractional trading. For such cases, Blueshift will automatically treat every order as fractional. For brokers which do not exclusively offer fractional trading (e.g. Equity brokers), you must specify a keyword argument *fractional=True* in the ordering API functions to make it fractional.

Place Order

The base ordering function on Blueshift is `order`.

`TradingAlgorithm.`**`order`**(*asset*, *quantity=None*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*)

Place a new order. This is the interface to the underlying broker for ALL order related API functions.

The handling of limit and stop price specification is totally implementation dependent. In case the broker supports limit orders, limit_price will be effective.

Important:

- Orders can be placed during active trading hours only. This means only *handle_data* or *scheduled callback* functions, or appropriate trade or data callback functions are eligible to place orders.
- Order with zero implied quantity (e.g. order size less than lot-size) will silently fail. No order will be sent to the broker.
- At present only limit and market orders are supported. Stop loss specification will be ignored.
- For order using rolling assets, see the caveats *here*.

- For intraday products, order placement will be refused if the broker follows an intraday cut-off time (usually 15 mins from the end of trading day).
- Always check if the return value is None or a valid order id. For some live brokers, successful orders that result in an existing position unwind may return a `None` value. Also a rejected order or an order that failed validation checks may return a `None` value (e.g. order with 0 quantity).

Parameters

- **`asset`** (*asset* object.) – asset on which the order to be placed.
- **`quantity`** (*`int`*) – units to order (> 0 is buy, < 0 is sale).
- **`limit_price`** (*`float`*) – limit price for limit order
- **`stop_price`** (*`float`*) – Stop-loss price (currently ignored).
- **`kwargs`** – Extra keyword parameters passed on to the broker.

Returns
order ID.

Return type
str or None.

Recognized keyword arguments are *validity* and *product_type*. If specified, they must be of type *OrderValidity* and *ProductType* respectively (names are also accepted, instead of *enums*).

> **Danger:** No ordering function will check the capacity of the account to validate the order (e.g. cash, margin requirements etc.). You must check before placing any order.

Automatic Order Sizing

Apart from this there is a host of automatic order-sizing functions that allows one to place orders based on order value or by specifying the fraction of current portfolio value as order value.

`TradingAlgorithm.`**`order_value`**`(`*asset*, *value*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*`)`

Place a new order sized to achieve a certain dollar value, given the current price of the asset.

Parameters

- **`asset`** (*asset* object.) – asset on which the order to be placed.
- **`value`** (*`float`*) – dollar value (> 0 is buy, < 0 is sale).
- **`limit_price`** (*`float`*) – limit price for limit order
- **`stop_price`** (*`float`*) – Stop-loss price (currently ignored).
- **`kwargs`** – Extra keyword parameters passed on to the broker.

Returns
order ID.

Return type
str or None.

Danger: This will take into account the current price of the asset, not actual execution price. The total value of execution can exceed the specified target value.

See also:

blueshift.algorithm.algorithm.TradingAlgorithm.order

`TradingAlgorithm.`**`order_percent`**(*asset*, *percent*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*)

Place a new order sized to achieve a certain percentage of the algo net equity, given the current price of the asset. This method applies a haircut of 2% on the current portfolio value for computing the order value, in case of market order.

Parameters

- **`asset`** (*asset* object.) – asset on which the order to be placed.
- **`percent`** (*float*) – fraction of portfolio value (> 0 is buy, < 0 is sale).
- **`limit_price`** (*float*) – limit price for limit order
- **`stop_price`** (*float*) – Stop-loss price (currently ignored).
- **`kwargs`** – Extra keyword parameters passed on to the broker.

Returns

order ID.

Return type

str or None.

Danger: This will take in to account current price of the asset and current algo net equity, not actual execution price. The total value of execution can exceed the specified percent.

See also:

blueshift.algorithm.algorithm.TradingAlgorithm.order

Targeting Orders

In addition, Blueshift supports targeting order, which takes in a unit, a order value or a portfolio fraction as a target and tries to place order to achieve this target. These functions are `idempotent` and are recommended way to place orders from an algo.

`TradingAlgorithm.`**`order_target`**(*asset*, *target*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*)

Place a new order sized to achieve a position of a certain quantity of the asset, taking into account the current positions and outstanding open orders.

Parameters

- **`asset`** (*asset* object.) – asset on which the order to be placed.
- **`target`** (*int*) – units to target (> 0 is buy, < 0 is sale).
- **`limit_price`** (*float*) – limit price for limit order
- **`stop_price`** (*float*) – Stop-loss price (currently ignored).

- **kwargs** – Extra keyword parameters passed on to the broker.

Returns
order ID.

Return type
str or None.

Important: If the current position in the asset is X, and a target order is placed for Y units, and there is no outstanding open order for the asset, this will place an order for Y-X units. If X>Y, this means a sell order, if X < Y, a buy order. If X is exactly equal to Y, no actions are taken. If there are outstanding open orders, that is added to X before calculating the difference. In case of a significant delay in order update from the broker, this can compute wrong incremental units.

See also:

blueshift.algorithm.algorithm.TradingAlgorithm.order

`TradingAlgorithm.`**`order_target_percent`**(*asset*, *percent*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*)

Place a new order sized to achieve a position of a certain percent of the net account value. This method applies a haircut of 2% on the current portfolio value for computing the order value, in case of market order.

Parameters

- **asset** (*asset* object.) – asset on which the order to be placed.
- **percent** (*float*) – fraction of portfolio value to target (> 0 is buy, < 0 is sale).
- **limit_price** (*float*) – limit price for limit order
- **stop_price** (*float*) – Stop-loss price (currently ignored).
- **kwargs** – Extra keyword parameters passed on to the broker.

Returns
order ID.

Return type
str or None.

See also:

blueshift.algorithm.algorithm.TradingAlgorithm.order_percent

`TradingAlgorithm.`**`order_target_value`**(*asset*, *target*, *limit_price=0*, *stop_price=0*, *bypass_control=False*, *style=None*, ***kwargs*)

Place a new order sized to achieve a position of a certain value of the asset, taking into account the current positions and outstanding open orders.

Parameters

- **asset** (*asset* object.) – asset on which the order to be placed.
- **target** (*float*) – value to target (> 0 is buy, < 0 is sale).
- **limit_price** (*float*) – limit price for limit order
- **stop_price** (*float*) – Stop-loss price (currently ignored).
- **kwargs** – Extra keyword parameters passed on to the broker.

Returns
order ID.

Return type
str or None.

Important: This order method computes the required unit from the target and the current market price, and does not guarantee the execution price or the value after execution.

See also:

blueshift.algorithm.algorithm.TradingAlgorithm.order_value

Danger: It is highly recommended to use targeting order functions above than the basic ordering function. Basic order function, if not used correctly, may lead to sending inadvertent orders or too many orders.

Advanced Algo Orders

See *Execution Algorithms*.

5.2.2 Order management APIs

Attention: All order management APIs can only be used during active trading hours. Else they will raise exceptions. Also, do not use the undocumented parameters in the below APIs. They are meant for internal use only.

Update Order

`TradingAlgorithm.`**`update_order`**`(`*order_param*, **args*, ***kwargs*`)`

Function to update an existing open order. The parameter *order_param* can be either an Order object or an order ID. Use *limit_price* keyword for updating the price of a limit order. Other keyword support is implementation dependent.

Important: This API will try to modify an existing order, but will fail if the order is already executed.

Parameters
`order_param` – An *order* object, or a valid order ID to modify.

Cancel Order

`TradingAlgorithm.`**`cancel_order`**`(`*order_param*`)`

Function to cancel an open order not filled yet (or partially filled).

Parameters
`order_param` – An *order* object, or a valid order ID to cancel.

Danger: This function only initiates a cancel request. It does not and cannot ensure actual cancellation.

Fetch Open Orders

`TradingAlgorithm.`**`get_open_orders`**`(`*algo=True*`)`

Get a dictionary of all open orders, keyed by their id. The return value is a dict object with order IDs (str) as keys and *order* objects as values.

Returns
A dictionary of open orders.

Return type
dict

Fetch Open Positions

`TradingAlgorithm.`**`get_open_positions`**`(`*algo=True*`)`

Get all open positions. The return value is a dict, keyed by *assets* and *positions* as values.

Returns
A dictionary of open orders.

Return type
dict

Get Order by Order ID

`TradingAlgorithm.`**`get_order`**`(`*order_id*, *algo=True*`)`

Function to retrieve an order by order id.

Parameters
`order_id` (*`str`*) – A valid order id to retrieve.

Returns
The matching *order* object.

Important: Up to what history orders can be retrieved depends on broker implementation. Usually for most cases, only the closed orders placed in the current session (day), plus all open orders are available.

Check If Tradable

`TradingAlgorithm.`**`can_trade`**`(`*assets*`)`

Function to check if asset can be traded at current dt.

Parameters

`assets` (*`list`*) – List of assets to check

Returns

`True` if all assets in the list can be traded, else `False`.

Return type

bool

Square Off Position

`TradingAlgorithm.`**`square_off`**`(`*assets=None*, *algo=True*`)`

Function to square off open positions and cancel open orders. Typically useful for end-of-day closure for intraday strategies or for risk management.

If the underlying broker natively support squaring off open positions, this method will use that interface. In such cases, squaring off will only happen if there is an open position with the broker, not just with this algo. An example of such a case is two algos placing opposite orders - resulting in opposite positions in each algo, but no open positions with the broker. In such a case the square-off will do nothing. This behaviour is intentional to avoid creating unintended positions during square-off. Also, in such case, no cool-off period is enforced.

If the underlying broker does not support square-off natively, then the algo positions will be considered. In such case, a square-off may actually create a new position with the underlying broker. This will also result in a cool-off period during which a new entry into the same asset(s) will not be allowed. Additionally, during cool-off period, additional call to this method will not result in positions unwinding, but will cancel open orders. This behaviour is intentional to avoid creating unintended positions with repeated sqauare-off attempts.

If `assets` is *None*, all existing open orders will be cancelled, and then all existing positions will be squared off. If `assets` is a list or a single asset, only those positions and orders will be affected.

Important: This API will bypass any trading controls to make sure square-off orders are not blocked by such controls. Also, see above the behaviour differences between the cases where the underlying broker does or does not support a native square-off method.

Parameters

- **`assets`** (*`list`*) – A list of assets, or a single asset or None
- **`algo`** (*`bool`*) – If we should consider algo or broker positions

Danger: This function only initiates square off. It does not and cannot ensure actual square-off. Also, in case we attempt to squareoff during a cool-off period, it will be ignored (to prevent multiple squareoff attempts which may lead to unintended positions), but will attempt to cancel any open orders.

See also:

`blueshift.algorithm.algorithm.TradingAlgorithm.set_cooloff_period`

5.2.3 Stoploss and Take-profit

Add or Remove Stoploss

`TradingAlgorithm.`**`set_stoploss`**`(`*asset*, *method*, *target*, *trailing=False*, *on_stoploss=<function noop>*`)`

Set the stoploss for a position, defined by an asset. Method can be one of *PRICE*, *MOVE*, *PERCENT*. PRICE set the target as the stop level. *MOVE* will add the target to the entry price to compute stop level, whereas *PERCENT* will compute the percent move from the entry price. If *trailing* is True, stoploss is updated at each lookup frequency (i.e. every minute).

Parameters

- **`asset`** (*asset* object.) – The asset for the stoploss check.
- **`method`** (*`str`*) – The method for the stoploss check.
- **`target`** (*`float`*) – The stoploss target.
- **`trailing`** (*`bool`*) – If it is a trailing stoploss.

Important: If you are placing the entry order with a stoploss specified, do not use this function. The stoploss in the order, if supported by the broker, will automatically enable stoploss exit. This function will try to square off the position if the stoploss is hit by placing a market order. Also, after an exit, this will cause a cool-off period for that asset which will prevent further entry trade till it resets. Cool off period can be set using the *set_cooloff_period* api function. Also a stoploss, once set, will be valid till it is cancelled using *remove_stoploss* api method.

See also:

`blueshift.algorithm.algorithm.TradingAlgorithm.set_cooloff_period`

`TradingAlgorithm.`**`remove_stoploss`**`(`*asset*`)`

Remove stoploss for an asset.

Parameters
`asset` (*asset* object.) – The asset for the stoploss check.

Add or Remove Take-profit Target

`TradingAlgorithm.`**`set_takeprofit`**`(`*asset*, *method*, *target*, *on_takeprofit=<function noop>*`)`

Set the take-profit for a position, defined by an asset. Method can be one of *PRICE*, *MOVE*, *PERCENT*. PRICE set the target as the level. *MOVE* will add the target to the entry price to compute level, whereas *PERCENT* will compute the percent move from the entry price.

Parameters

- **`asset`** (*asset* object.) – The asset for the stoploss check.
- **`method`** (*`str`*) – The method for the stoploss check.
- **`target`** (*`float`*) – The stoploss target.

Important: If you are placing the entry order with a takeprofit specified, do not use this function. The takeprofit in the order, if supported by the broker, will automatically enable takeprofit exit. Also a takeprofit, once set, will be valid till it is cancelled using *remove_takeprofit* api method.

`TradingAlgorithm.`**`remove_takeprofit`**`(`*asset*`)`

Remove takeproft for an asset.

Parameters
`asset` (*asset* object.) – The asset for the stoploss check.

Important: Always place stoploss or take-profit orders using assets from the *strategy positions*, and not using the assets used to place the orders. This is because for special cases (rolling assets or margin products etc.) the order asset can be different than the asset in the resulting position.

5.3 Risk Management APIs

Attention: These set of API functions (apart from `terminate`) can only be called within *initialize* function.

A set of API functions to control trade risk and implements various limits.

`TradingAlgorithm.`**`set_allowed_list`**`(`*assets*, *on_fail=None*`)`

Defines a whitelist of assets to be ordered. Any order outside these assets will be refused (with a warning). Usually, the user script will use either this function or the blacklist function `set_do_not_order_list`, but not both.

Parameters
`assets` (*`list`*) – A list of assets.

See also:

`blueshift.algorithm.algorithm.TradingAlgorithm.set_do_not_order_list`

`TradingAlgorithm.`**`set_do_not_order_list`**`(`*assets*, *on_fail=None*`)`

Defines a list of assets not to be ordered. Any order on these assets will be refused (with a warning).

Parameters
`assets` (*`list`*) – A list of assets.

See also:

`blueshift.algorithm.algorithm.TradingAlgorithm.set_allowed_list`

`TradingAlgorithm.`**`set_long_only`**`(`*on_fail=None*`)`

Set a flag for long only algorithm. Any short-selling order (attempt to sell without owning the assets) will be refused (and a warning raised).

`TradingAlgorithm.`**`set_max_daily_size`**`(`*assets=None*, *max_quantity=None*, *max_notional=None*, *on_fail=None*`)`

Set a limit on the order size - in terms of total daily limits. If `assets` is None, it is applied for all assets, else only for assets in the list.

Parameters

- **`assets`** (*`list`*) – A list of assets for position control.
- **`max_quantity`** (*`int`*) – Maximum order quantity allowed.
- **`max_notional`** (*`float`*) – Maximum order value allowed.

> **Warning:** Specifying both max_quantity and max_notional will raise exception.

`TradingAlgorithm.`**`set_max_exposure`**`(`*max_exposure*, *on_fail=None*`)`

Set a limit on the account gross exposure. Any order that can potentially exceed this limit will be refused (with a warning).

Parameters
`max_exposure` (*`float`*) – Maximum allowed exposure.

`TradingAlgorithm.`**`set_max_leverage`**`(`*max_leverage*, *on_fail=None*`)`

Set a limit on the account gross leverage . Any order that can potentially exceed this limit will be refused (with a warning).

Parameters
`max_leverage` (*`float`*) – Maximum allowed leverage.

`TradingAlgorithm.`**`set_max_order_count`**`(`*max_count*, *on_fail=None*`)`

Set a limit on the maximum number of orders generated in a day. Any order that can exceed this limit will be refused (and will raise a warning).

Parameters
`max_count` (*`int`*) – Maximum number of orders allowed per session (day).

`TradingAlgorithm.`**`set_max_order_size`**`(`*assets=None*, *max_quantity=None*, *max_notional=None*, *on_fail=None*`)`

Set a limit on the order size - either in terms of quantity, or value. Any order exceeding this limit will not be processed and a warning will be raised.

Parameters

- **`assets`** (*`list`*) – List of assets for this control. If assets is a dict, it should be in asset:value format, and will only apply to the assets mentioned. If assets is a list, the same value wil apply to all assets. If assets is None, the control value will apply to ALL assets.
- **`max_quantity`** (*`int`*) – Maximum quantity allowed (unsigned).
- **`max_notoinal`** (*`float`*) – Maximum value at current price.

`TradingAlgorithm.`**`set_max_position_size`**`(`*assets=None*, *max_quantity=None*, *max_notional=None*, *on_fail=None*`)`

Set a limit on the position size (as opposed to order size). Any order that can exceed this position (at current prices) will be refused (and will raise a warning). If `assets` is None, it is applied for all assets, else only for assets in the list.

Parameters

- **`assets`** (*`list`*) – A list of assets for position control.
- **`max_quantity`** (*`int`*) – Maximum position quantity allowed.
- **`max_notional`** (*`float`*) – Maximum position exposure allowed.

> **Warning:** Specifying both max_quantity and max_notional will raise exception.

`TradingAlgorithm.`**`terminate`**`(`*error_message=None*, *cancel_orders=True*`)`

Exit from the current run by scheduling the algo end event. This will cause the algorithm to exit execution as soon as possible. This will also trigger the `analyze` API function as it exits.

Parameters
`cancel_orders` (*bool*) – If open orders should be cancelled.

> **Danger:** This function only initiates outstanding orders cancellation, and does NOT guarantee it.

5.4 Pipeline APIs

Pipelines are a set of APIs to systematically select universe by computing filter and factors.

`TradingAlgorithm.`**`attach_pipeline`**(*pipeline*, *name*, *chunks=None*, *eager=True*)

Register a pipeline for this algorithm. Pipeline with the same name will be overwritten.

Args:

`pipeline (obj)`: A pipeline object to register.

`name (str)`: Name of the pipeline.

`chunks (iterable)`: Iterator to specify the computation chunks.

`eager (bool)`: Compute ahead the pipeline.

Returns:

Returns the pipeline object registered.

`TradingAlgorithm.`**`pipeline_output`**(*name*)

return pipeline output for the given name.

5.4.1 Pipeline in live trading

Blueshift 2.0 introduces pipeline API support in live trading. In live trading, all data query is sourced from the broker feed. However, the pipeline data query is directed to Blueshift database (daily frequency). The pipeline is, as usual, evaluated on daily prices till one day before the evaluation date.

> **Danger:** Pipeline APIs are evaluated on the last close. If the latest data from our data vendors are not updated for some reason on a particular date, the pipeline evaluation will fail. In such case, we raise an exception and the live strategy will terminate with Error.

5.5 Backtest Model Selection APIs

The following API functions allow one to customise various aspects of the backtest simulation. These functions, of course, do not apply to live trading and are ignored in `live` mode.

`TradingAlgorithm.`**`set_slippage`**(*model=None*, **args*, ***kwargs*)

Set the slippage model. For more details on available slippage models, see *Slippage Models*.

> **Warning:** This method can only be called once in the *initialize* function. Calling it anywhere else will crash the algo. This method is ignored in live trading.

Parameters
`model` (*slippage model.*) – A valid slippage model object.

`TradingAlgorithm.`**`set_commission`**(*model=None*, *charge_model=None*, **args*, ***kwargs*)

Set the commission (cost) model. For more details see the *Commissions and Cost Models* descriptions. A commission model captures trading costs that are charged by the broker or the trading venue. An optional charge_model can also be specified to capture trading charges captured by authorities (e.g. taxes and levies).

Warning: This method can only be called once in the *initialize* function. Calling it anywhere else will crash the algo. This method is ignored in live trading.

Parameters
`model` (*commission model.*) – A valid commission model object.

`TradingAlgorithm.`**`set_margin`**(*model*)

Set the margin model. For more details on available margin models, see *Margin Models*.

Warning: This method can only be called once in the *initialize* function. Calling it anywhere else will crash the algo. This method is ignored in live trading.

Parameters
`model` (*margin model.*) – A valid margin model object.

5.6 Miscellaneous API functions

`TradingAlgorithm.`**`get_datetime`**()

Get the current date-time of the algorithm context. For live trading, this returns the current real-time. For a backtest, this will return the simulation time at the time of the call.

Returns
current date-time (Timestamp) in the algo loop.

Return type
pandas.Timestamp

`TradingAlgorithm.`**`record`**(**args*, ***kwargs*)

Record a list of var-name, value pairs for each day.

Parameters
`kwargs` – the names and values to record. Must be in pairs.

Note: The recorded values are tracked within the context, as `record_vars` variable. Also, any variable record is kept as only one point per day (session). If the user script records values at multiple points in a day, only the last value will be retained. Use of this function can slow down a backtest.

See `blueshift.algorithm.context.AlgoContext.record_vars`.

`TradingAlgorithm.`**`set_benchmark`**(*asset*)

Overwrite the default benchmark asset for algo performance analysis.

Param
asset object.

Warning: This method can only be called once in the *initialize* function. Calling it anywhere else will crash the algo.

`TradingAlgorithm.`**`set_account_currency`**`(`*currency*`)`

Set the currency of the account. Only valid for backtests.

Parameters
`currency` (str or `CCY`) – A supported currency code or enumeration.

Note: for non-fx brokers, this will be ignored. All non-fx brokers assume the algo performance is maintained in the `LOCAL` currency.

`TradingAlgorithm.`**`set_cooloff_period`**`(`*period=30*`)`

Set the cool-off period following a call to the square_off function, or following an exit resulting from a take profit or stop loss.

Parameters
`period` (*`int`*) – The period (in minutes) to cool off.

`TradingAlgorithm.`**`set_algo_parameters`**`(`*param_name='params'*`)`

Declare a context attribute as algo parameters.

Parameters
`param_name` (*`str`*) – An attribute of the context variable.

Note: If the attribute does not exist, a new attribute is created and set to the value passed as the *–parameters* flag as a dictionary of key-value pairs. If it exists, it must be a dictionary, and parameter key-value pairs from the *–parameters* flag are added to the attribute (overwritten if exists).

CHAPTER

SIX

OBJECTS, MODELS AND CONSTANTS

6.1 Trading Calendar

class blueshift.protocol.**TradingCalendar**

A trading calendar maintains the trading sessions for a given trading venue (e.g. an exchange like NYSE). It has an associated timezone. It tracks sessions opening/ closing times and holidays. Strategy can access it via the context variable as `AlgoContext.trading_calendar`.

See also:

See documentation on the *Context Object*.

bump_forward(*dt*)

bump to the next date if dt is a holiday.

bump_mod_forward(*dt*)

bump to next date, unless the result is in the next month, in which case, bump to the previous date.

bump_mod_previous(*dt*)

bump to previous, unless the result is in the previous month, in which case, bump to the next date.

bump_previous(*dt*)

bump to the previous date if dt is a holiday.

property close_time

session closing time in `datetime.time` format.

current_session(*dt*)

returns the current session. Returns next if not a trading session.

is_holiday(*dt*)

check (`bool`) if it is a holiday, given a timestamp.

is_open(*dt*)

check (`bool`) for if a trading session is in progress, given a timestamp.

is_session(*dt*)

check (`bool`) if it is a trading day, given a timestamp.

last_n_sessions(*dt*, *n*, *convert=True*)

Returns last n trading sessions, including dt.

last_trading_minute(*dt*)

returns the last trading minute.

property minutes_per_day

total number of minutes (`int`) in a trading session.

property name

name (`str`) of the trading calendar.

next_close(*dt*)

returns next close session time (`pandas.Timestamp`), given a timestamp.

next_n_sessions(*dt*, *n*, *convert=True*)

Returns next n trading sessions, including dt.

next_open(*dt*)

returns next open session time (`pandas.Timestamp`), given a timestamp.

next_session(*dt*)

returns the next session.

property open_time

session opening time in `datetime.time` format.

previous_close(*dt*)

returns previous close session time (`pandas.Timestamp`).

previous_open(*dt*)

returns previous open session time (`pandas.Timestamp`) , given a timestamp.

previous_session(*dt*)

returns the previous session.

sessions(*start_dt*, *end_dt*, *convert=True*)

list all valid sessions between dates, inclusive (`pandas.DatetimeIndex`).

to_close(*dt*)

returns the close time of the curret session(`pandas.Timestamp`) , given a timestamp. Returns the next session open if the market is closed.

to_open(*dt*)

returns current open session time (`pandas.Timestamp`) , given a timestamp. Returns the next session open if the market is closed.

property tz

time-zone (`str`) of the trading calendar.

Note: Calendar objects can be imported into the strategy code from `blueshift.protocol` module. But there is no reasonable case that a user strategy may need to import this class or instantiate an instance. Use *`AlgoContext.trading_calendar`* to get a reference to the trading calendar of the running algo.

6.2 Assets

Assets are objects in Blueshift that encapsulates information about tradable instruments (e.g. equities) or non-tradable market data (e.g. a market index). User strategy code uses the *symbol* function to convert a symbol name to the corresponding asset objects. This asset object then can be passed on to market data query, order placing functions etc.

6.2.1 Asset Symbology

The symbology followed on the platform is pretty straightforward. Exchange traded equities are referred to by their respective exchange symbols. Forex assets are referred to as `CCY1/CCY2` (with the conventional meaning).

In all cases, you can optionally add the exchange reference as `EXCHANGE:SYMBOL`. Note, adding exchange is NOT required at present, as Blueshift does not support multiple exchanges for the same security currently. Also, specifying a wrong exchange name can raise the `SymbolNotFound` exception.

Note: Special characters in symbol names (e.g. '-' or '&') are replaced by an underscore ('_') in Blueshift. This means a symbol like 'M&M' should be entered in the *symbol* function like 'M_M'.

Dated and Rolling Futures and Options

For futures, the symbol is `SYM<YYYYMMDD>` for dated instruments - where `SYM` refers to the underlying symbol and the date part is the expiry date. For option the format is `SYM<YYYYMMDD>TYPE<STRIKE>`, where `TYPE` is the *option type* and can be either `CE` or `PE`. The `STRIKE` is the strike price of the option (with no leading or trailing zeros).

Referring to rolling (continuous) futures and options are simple. Instead of the expiry date, specify the expiry series. For NSE monthly futures and options, specify `-I` for the near month and `-II` for the far month. For weekly options, specify `-W{n}` instead, where n is the week offset (starting from 0 for the current week). For specifying relative strike, use + or - and specify the strike difference from the ATM (usually the near month futures). Below shows some examples.

```python
from blueshift.api import symbol

def initialize(context):
    asset0= symbol('ACC') # ACC equity on NSE
    asset1= symbol('NSE:ACC') # Specify the exchange (not needed)
    asset2= symbol('ACC20210826') # ACC Aug 2021 futures on NSE
    asset3= symbol('NIFTY20210826CE16000') # Nifty Aug 21 call at 16K strike
    asset4= symbol('NIFTY-I') # NSE Nifty first month futures
    asset5= symbol('NIFTY-ICE+100') # Nifty ATM+100 call near-month
    asset6= symbol('NIFTY-W0PE-0') # Nifty current-week ATMF put
```

You can use the rolling symbology for both data query and placing orders. See the caveat below on how rolling symbology differs for futures and options in backtesting and live trading.

Warning: Be careful when using rolling assets on Blueshift for placing orders. Futures and options rolling assets behave differently in backtesting. Positions from orders with rolling futures are tracked as the rolling futures itself, and the position is automatically rolled on expiry date. Orders for rolling options creates positions in the specific expiry and strike prevailing at the time of the order. Such positions will not be rolled automatically. In live trading, both are treated as specific instruments without any automatic roll.

6.2.2 Asset Related Constants

Below are constants related to asset definitions.

AssetClass

Asset class of the asset object.

class `blueshift.assets.`**`AssetClass`**`(`*value*`)`

An enumeration.

`EQUITY = 0`

`FOREX = 1`

InstrumentType

Instrument type of the asset object. `SPOT` assets are traded with full price paid during buying and selling. `MARGIN` assets are traded on the margin. `FUTURES` as futures derivatives on another underlying asset. `OPT` are options on another underlying asset.

class `blueshift.assets.`**`InstrumentType`**`(`*value*`)`

An enumeration.

`SPOT = 0`

`FUTURES = 1`

`OPT = 2`

`MARGIN = 3`

OptionType

Options assets can be either `CALL` or `PUT`. All options are at present assumed to be of the European type.

class `blueshift.assets.`**`OptionType`**`(`*value*`)`

An enumeration.

`CALL = 0`

`PUT = 1`

StrikeType

Asset symbology supports specifying options strikes as either absolute (ABS) or relative (to the future-implied ATM strike, REL).

class `blueshift.assets.`**`StrikeType`**`(`*value*`)`

An enumeration.

`ABS = 0`

`REL = 1`

6.2.3 Types of Asset

Different assets are modelled as objects of different classes. The base class is *MarketData*. All asset classes are derived from it. Below provides a list of supported assets.

Market Data

`class blueshift.assets.MarketData`

MarketData class encapsulates an object with which some data (pricing or otherwise) may be associated. All tradable assets are derived from this class. This can also represent other generic data series like macro-economic data or corporate fundamental data.

A MarketData object has the following attributes. All attributes are read-only from a strategy code.

Attribute	Type	Description
sid	`int`	Unique Identifier
symbol	`str`	Symbol string
name	`str`	Long name
start_date	`pandas.Timestamp`	Start date of data
end_date	`pandas.Timestamp`	End date of data
ccy	*Currency*	Asset currency
exchange_name	`str`	Name of the exchange
calendar_name	`str`	Name of the calendar

Asset

`class blueshift.assets.Asset`

Asset models a tradeable asset and derives from `MarketData`.

An Asset object has the following attributes in addition to the attributes of the parent class. All attributes are read-only from a strategy code.

Attribute	Type	Description
asset_class	*AssetClass*	Asset class
instrument_type	*InstrumentType*	Instrument type
mult	`float`	Asset multiplier (lot size)
tick_size	`int`	Tick size (reciprocal)
auto_close_date	`pandas.Timestamp`	Auto-close day
can_trade	`bool`	If tradeable
fractional	`bool`	Fractional trading supported

Warning: The `mult` attribute defaults to 1. Any order placed for an asset should include this in the order size. Order placement routine will check if the order size is a multiple of `mult`. For e.g. an asset with a lot size of 75 should use order size of 150 (`order(asset, 150)`) to place an order for 2 lots.

Note: The attribute `tick_size` is stored as reciprocal of the actual tick size. For a tick size of 0.05, the value stored will be 20. It defaults to 100 (corresponding to a tick size of 0.01).

The `auto_close_date` is the date on which the asset is automatically squared-off (if still held by the algo). This defaults to the `end_date`.

Forex

`class blueshift.assets.Forex`

Forex models for margin traded forex. This is derived from the `Asset` class. This has the following attributes in addition to the attributes of the parent class. All attributes are read-only from a strategy code.

Attribute	Type	Description
ccy_pair	`str`	currency pair (XXX/YYY)
base_ccy	`str`	Base currency (XXX)
quote_ccy	`str`	Quote currency (YYY)
buy_roll	`float`	Roll cost for a long position
sell_roll	`float`	Roll cost for a short position
initial_margin	`float`	Initial margin
maintenance_margin	`float`	Maintenance margin

Note: The `asset_class` is set to `FOREX` and the `instrument_type` is `MARGIN`.

The roll costs are for 1 unit - the overnight roll is calculated as roll cost multiplied by the quantity in open position.

Equity

Fully funded cash equity asset.

`class blueshift.assets.Equity`

Equity models exchange traded equities asset. This is derived from the `Asset` class. This has the following attributes in addition to the attributes of the parent class. All attributes are read-only from a strategy code.

Attribute	Type	Description
shortable	`bool`	If shortable
borrow_cost	`float`	Borrow cost if shortable

Note: The `asset_class` is set to `EQUITY` and the `instrument_type` is `SPOT` for equities and FUNDS for ETFs.

EquityMargin

Equity product traded on the margin.

`class blueshift.assets.EquityMargin`

Margin traded equity product.

EquityIntraday

Equity products traded on the margin which cannot be carried overnight. Usually fresh orders are restricted after a cut-off time towards the closing hours of the market.

`class blueshift.assets.EquityIntraday`

Intraday margin traded equity product.

EquityFutures

`class blueshift.assets.EquityFutures`

Equity futures models exchange traded equities futures. This is derived from the `Asset` class. This has the following attributes in addition to the attributes of the parent class. All attributes are read-only from a strategy code.

Attribute	Type	Description
underlying	`str`	Symbol of underlying asset
root	`str`	Root part of the symbol
roll_day	`int`	Roll period if rolling asset
expiry_date	`pandas.Timestamp`	Date of expiry
initial_margin	`float`	Initial margin
maintenance_margin	`float`	Maintenance margin

Note: The `asset_class` is set to `EQUITY` and the `instrument_type` is `FUTURES`. The `roll_day` parameter determines if the asset is a rolling asset (i.e. should be rolled automatically in backtest if value is greater than -1). The `roll_day` is the offset from the `expiry_date` to determine the roll date.

EquityOption

`class blueshift.assets.EquityOption`

Exchange traded equity options. This is derived from the `EquityFutures` class. This has the following attributes in addition to the attributes of the parent class. All attributes are read-only from a strategy code.

Attribute	Type	Description
strike	`float`	Option strike
strike_type	*StrikeType*	Option strike type
option_type	*OptionType*	Type of option

Note: The `asset_class` is set to `EQUITY` and the `instrument_type` is `OPT`. The parameter `strike_type` determines the offset from the ATM strike, and can be either absolute, relative from ATM (e.g. +100 or -100) or in terms of delta (multiplied by 100, e.g. 25D or -10D).

Note: All these objects can be imported into the strategy code from `blueshift.assets`.

6.2.4 Data and Simulation Models for Assets

On the platform, the data availability and default simulation behaviours depend on the asset class. By default, equities are considered fully funded instruments. That is, buying requires the full cost to be paid in cash (and shorting generates the opposite cash flow). The default simulation model is the `blueshift.finance.slippage.VolumeSlippage`. For modelling equity trading on the margin, use *EquityMargin*, which uses the same slippage model, but tracks the required margins and related cash flows through the `blueshift.finance.margin.FlatMargin` by default. *EquityFutures* uses the same default slippage and margin models.

EquityOptions slippage are computed differently. The slippage, in this case, is based on the options vega instead of volumes. The maximum transaction limit per bar is 100 lots, and the impact is calculated as a 0.5% bump on the vega (i.e. vega*implied_vol*0.005). Also, since the volatilty smile is computed from puts for below ATM and from calls for above ATM strikes, strategies exploring put and call implied volatity discrepancies at the same strike (for example, put call parity)) cannot be modelled.

The forex related assets are modelled using the `blueshift.finance.slippage.BidAskSlippage` by default.

6.3 Orders

6.3.1 ProductType

ProductType determines the handling of the position. `DELIVERY` is the usual type, where an asset is purchased by paying the full price and is held till it is sold (or auto-closed by the algo). `MARGIN` is when the asset is purchased on the margin (paying a fraction of the cost). `INTRADAY` is specific to certain geographies (e.g. NSE), it is same as margin, but the asset is automatically squared-off the same day (if not already done so), by the broker, after a cut-off time.

class `blueshift.protocol.`**`ProductType`**(*value*)

An enumeration.

`INTRADAY = 0`

`DELIVERY = 1`

`MARGIN = 2`

6.3.2 OrderType

Order types determines how the orders are to be executed. A `MARKET` order is executed at the best available price. A `LIMIT` order is put in the order book and is executed at the limit price or better. A `STOPLOSS_MARKET` order is an order that gets activated after a certain trigger (stoploss price) and becomes a market order after that. A `STOPLOSS` (or a stop-limit) order gets activated after a trigger is breached (stoploss price), and then becomes a limit order. A limit price is required for a limit order. A stoploss price is required for a stoploss market order. Specifying both is required for a stop-limit order.

class `blueshift.protocol.`**`OrderType`**(*value*)

An enumeration.

`MARKET = 0`

`LIMIT = 1`

`STOPLOSS = 2`

`STOPLOSS_MARKET = 3`

6.3.3 OrderValidity

Validity of an order. `DAY` means valid for the entire trading day (till cancelled by the algo). `IOC` or immediate-or-cancel ensures the order is filled fully or as much as possible as soon as it arrives, and the remaining part is cancelled if it cannot be filled immediately. `FOK` or fill or kill orders are filled either fully or cancelled (i.e. avoids partial fill). `AON` or all-or-none is similar to fill-or-kill, but they are not cancelled and remain in the order book - but can be filled either fully or none at all. `GTC` is valid till the order is explicitly cancelled. `OPG` (market-on-open) and `CLS` (market-on-close) are not implemented, but can be supported by a live broker.

class `blueshift.protocol.`**`OrderValidity`**`(`*value*`)`

An enumeration.

`DAY = 0`

`IOC = 1`

`FOK = 2`

`AON = 3`

`GTC = 4`

`OPG = 5`

`CLS = 6`

Note: the default validity is `DAY`.

6.3.4 OrderSide

A `BUY` order or a `SELL` order.

class `blueshift.protocol.`**`OrderSide`**`(`*value*`)`

An enumeration.

`BUY = 0`

`SELL = 1`

6.3.5 OrderStatus

Status of an order. `COMPLETE` means fully executed. `OPEN` means partially (including 0) executed and still active. `REJECTED` means the order is no longer active, and is rejected by the broker. `CANCELLED` means the order is no longer active and is cancelled by the algo. A rejected or cancelled order can be partially filled.

class blueshift.protocol.**OrderStatus**(*value*)

An enumeration.

COMPLETE = 0

OPEN = 1

REJECTED = 2

CANCELLED = 3

6.3.6 Order

class blueshift.protocol.**Order**

Order object encapsulates all details about orders sent by the algo to the broker. It also tracks the order fill. Orders are automatically tracked by internal trackers. Strategy code can query its attributes to check various details and fill status.

An order object has the following attributes. All attributes are read-only from a strategy code.

Attribute	Type	Description
oid	`str`	Order ID
asset	*Asset*	Asset of the position
quantity	`float`	Net quantity at present
product_type	*ProductType*	Product type
order_type	*OrderType*	Order type
order_validity	*OrderValidity*	Order validity
disclosed	`float`	Total amount disclosed
price	`float`	Limit price
trigger_price	`float`	Stoploss trigger price
stoploss_price	`float`	Limit for stop-limit order
filled	`float`	Amount filled so far
pending	`float`	Pending (quantity-filled)
average_price	`float`	Average price of fill
side	*OrderSide*	Order side (buy/sell)
status	*OrderStatus*	Order status
timestamp	`pandas.Timestamp`	Timestamp of last update
fractional	`bool`	If a fractional order
price_at_entry	`float`	Mid price at entry time
create_latency	`float`	Latency (milli) to create
exec_latency	`float`	Latency (milli) to place

Note: The `oid` is the field through which the platform tracks an order (which can be different from the broker or the exchange IDs).

is_open()

Is the order still open.

is_buy()

Is the order a buy order.

6.4 Positions

6.4.1 PositionSide

PositionSide captures the original side of the position - `LONG` if a long position, `SHORT` otherwise.

`class blueshift.protocol.`**`PositionSide`**`(`*value*`)`

An enumeration.

LONG = 0

SHORT = 1

6.4.2 Position

`class blueshift.protocol.`**`Position`**

A position reflect the current economic interest in an asset, including associated profit and losses accumulated over the lifetime of such position. Positions are maintained by the algo blotter (though a dedicated tracker) and are updated on each trade or as and when prices change.

A position object has the following attributes. All attributes are read-only from a strategy code.

Attribute	Type	Description
asset	*Asset*	Asset of the position
quantity	`float`	Net quantity at present
buy_quantity	`float`	Total buying quantity
buy_price	`float`	Average buy price
sell_quantity	`float`	Total sell quantity
sell_price	`float`	Average selling price
pnl	`float`	Total profit or loss
realized_pnl	`float`	Realised part of pnl
unrealized_pnl	`float`	Unrealized (MTM) part of pnl
last_price	`float`	Last updated price
last_fx	`float`	Last known FX conversion rate
underlying_price	`float`	Underlying price
timestamp	`pandas.Timestamp`	Timestamp of last update
value	`float`	Holding value of this position
cost_basis	`float`	Entry cost of this position
margin	`float`	Margin posted (if available)
product_type	*ProductType*	Product type
position_side	*PositionSide*	Position side (Long/Short)
fractional	`bool`	If a fractional order

See also:

See *Asset* for details on asset object. See *Orders* for details on order objects.

if_closed()

If this position is closed out.

6.5 Simulation Models

6.5.1 Slippage Models

class blueshift.finance.slippage.**SlippageModel**

Slippage model implements the backtest simulation. The core method *simulate* takes in an order and simulates the fill and the fill price.

simulate()

Takes in an order (may be already partially filled), and returns a trade with executed price and amount, as well as the fair mid. The difference between the execution price and the fair mid is the slippage.

Parameters

- **order** (*order* object.) – Order object to process.
- **dt** (*pandas.Timestamp*) – Timestamp for the order.

Returns

A tuple of volume, price, fair mid and max volume.

Return type

(float, float, float, float)

class blueshift.finance.slippage.**NoSlippage**

Trade simulation based on OHLCV data without any slippage. The max volume executed at each trading bar is capped at *max_volume* fraction (defaults to 0.02, i.e. 2% of available volume at each bar) of the available volume at the bar. Setting *max_volume* to 0 will disable any cap and the full pending amount will be assumed to trade successfully in a single trade. The impact cost is always zero, i.e. the order is executed at the available 'close' price on that bar.

Parameters

max_volume (*float*) – maximum volume that can be executed at a bar.

class blueshift.finance.slippage.**BidAskSlippage**

Trade simulation based on OHLCV data with bid-ask spread. The max volume executed is capped at *max_volume* fraction of the available volume at the bar. The impact cost equals to half the bid-ask spread. Setting`max_volume` to 0 will disable any cap and the full pending amount will be assumed to trade successfully in a single trade.

Parameters

max_volume (*float*) – maximum volume that can be executed at a bar.

Warning: This is supported only for cases where the dataset includes the 'bid' and 'ask' data. At present, only the Forex data set is suitable for this slippage model.

class blueshift.finance.slippage.**VolumeSlippage**

Trade simulation based on OHLCV data with volume based slippage. In this model, the max volume that can be executed is capped at a fixed percent of the volume at the bar. The price impact is modelled based on the below.

$$\Delta P = s/2 + \alpha.\sigma.(\frac{Q}{V})^{\beta}$$

where:

ΔP = impact, s = spread, α = cost coefficient, σ = historical volatility, Q = traded volume, V = available volume, β = cost exponent

Parameters

- **max_volume** (*float*) – maximum participation (defaults to 0.02).
- **cost_coeff** (*float*) – cost coefficient (defaults to 0.002).
- **cost_exponent** (*float*) – cost exponent (defaults to 0.5).
- **spread** (*float*) – constant spread (defaults to 0).
- **spread_is_percentage** (*bool*) – If false, spread is treated as absolute (defaults False).
- **use_vol** (*bool*) – If false, vol is set to 1.0 (defaults to False).
- **vol_floor** (*float*) – Floor value of vol (defaults to 0.05), ignored if use_vol is False.

See also:

For more on the model, see .

`class blueshift.finance.slippage.FixedSlippage`

Trade simulation based on OHLCV data with a fixed slippage. The max volume executed is capped at *max_volume* fraction of the available volume at the bar. Slippage is half the spread.

Parameters

- **spread** (*float*) – bid-ask spread (constant).
- **max_volume** (*float*) – maximum volume that can be executed at a bar.

`class blueshift.finance.slippage.FixedBasisPointsSlippage`

Trade simulation based on OHLCV data with a fixed slippage expressed in basis points (100th of a percentage point). The max volume executed is capped at *max_volume* fraction of the available volume at the bar. Slippage is half the spread. The actual spread applied is arrived at as $spread * close/10000$, where $close$ is the close price at that bar.

Parameters

- **spread** (*float*) – bid-ask spread (constant in basis points).
- **max_volume** (*float*) – maximum volume that can be executed at a bar.

6.5.2 Margin Models

`class blueshift.finance.margin.MarginModel`

Margin model calculates margins required (or released) for an order execution. For a fully funded trade, (e.g. cash equities) the margin is 0. For margin trading or derivatives (futures and options), each trade adds to or releases a certain margin. Margins are charged from the available cash in the portfolio. On release, it is added back to portfolio cash. Each order, before execution, is checked for sufficient funds in the account. If sufficient funds are not available, the order is rejected.

All margin models derive from this class and concretely implement two methods, *calculate* and *exposure_margin*. The method *calculate* is used to determine margin requirements for a transaction. The method *exposure_margin* is used to calculate daily MTM settlements at the end-of-day calculation.

Note: Selection of margin models has no impact on fully funded assets (e.g. cash equities).

`calculate()`

Calculate the cashflow and margin for a transaction. The algo account must have sufficient cash to cover for the transaction to happen.

Parameters

- **`order`** (*order* object.) – Order object to process.
- **`quantity`** (*`float`*) – Amount to trade.
- **`price`** (*`float`*) – Traded price.
- **`position`** (*`float`*) – Current position in the asset with sign.
- **`timestamp`** (*`pandas.Timestamp`*) – Current timestamp.
- **`last_fx`** (*`float`*) – FX rate for conversion.
- **`underlying_px`** (*`float`*) – Underlying price for option.
- **`pos`** (*position* object.) – Position object to process.

Returns
A tuple of cashflow, margin.

Return type
(float, float)

`exposure_margin()`

Compute the exposure margin, given an asset and position in that asset. This is used for end-of-day mark-to-market settlement computation and is settled against the account. If the account is short of the required cash, the account is frozen for further trading.

Parameters

- **`asset`** (*asset* object.) – Asset in which margin to calculate.
- **`exposure`** (*`float`*) – current exposure in the asset.
- **`timestamp`** (*`pandas.Timestamp`*) – current timestamp

Returns
exposure margin

Return type
float

`class blueshift.finance.margin.`**`NoMargin`**

No margin. This model allows any amount of leverage for unfunded products (e.g. forex or derivatives).

`class blueshift.finance.margin.`**`FlatMargin`**

Flat margin on total exposure as percentage. The parameter *intial_margin* is used to compute the fraction of the exposure required to be posted as margin for a transaction. This means *1/margin* is the leverage offered. The default value is 10% (i.e. 10x leverage). The parameter *maintenance_margin* is applied for computation of margin required for carrying an overnight position. It defaults to the same value as the *initial_margin* - that is, no extra margin is charged for overnight positions.

Parameters

- **`intial_margin`** (*`float`*) – Initial margin for a trade.

- **maintenance_margin** (*float*) – Maintenance margin.

`class blueshift.finance.margin.`**`RegTMargin`**

RegT margin. This is derived from *FlatMargin* class with set values of *intial_margin* at 0.5 (50%) and *maintenance_margin* the same as the *initial_margin* for carrying an overnight position.

`class blueshift.finance.margin.`**`VarMargin`**

Value-at-risk margin on total exposure - also known as portfolio margin. On blueshift, this is applied to each asset separately based on a multiplier of recent historical volatility. The formula is *max(vol, vol_floor)*vol_factor*. The default value of *vol_factor* is 3 (approximating a 99% VaR for normally distributed returns). The parameter *vol_floor* defaults to 0.05 (i.e. a minimum 5% volatility). The default value of lookback for historical volatility computation is 20. Volatilities are computed based on daily returns (daily volatility).

Parameters

- **vol_factor** (*float*) – Factor (z-score) for the probability threshold.
- **vol_floor** (*float*) – Minimum volatility level (percentage).
- **vol_lookback** (*float*) – Lookback to compute historical volatility.

6.5.3 Commissions and Cost Models

`class blueshift.finance.commission.`**`CostModel`**

CostModel defines the broking commission/ brokerage cost modelling (plus any exchange fees and/or tax/ regulatory charges, which defaults to zero and cannot be modified by the user). A concrete implementation must define the *calculate* method.

Parameters

- **commissions** (*float*) – Brokerage commission - interpretation depends on implementation.
- **cost_cap** (*float*) – Max possible brokerage.
- **cost_floor** (*float*) – Minimum brokerage levied.
- **cost_on_sell_only** (*bool*) – If brokerage only on sell leg.
- ****kwargs** – For compatibility, see below.

Keyword Arguments

- *cost* (`float`) – if supplied, overwrites the *commission* parameter.
- *min_trade_cost* (`float`) – if supplied, overwrites the *cost_floor* parameter.

`calculate()`

Calculate the commission and charges for the transaction. Commission is the fees deducted by the broker for this particular transaction. Charges are any exchange fees and/ or government or regulatory charges on top.

Parameters

- **order** (*order* object.) – Order object to process.
- **quantity** (*float*) – Traded amount.
- **price** (*float*) – Traded price.
- **last_fx** (*float*) – FX rate for conversion (defaults to 1.0).

Returns
A tuple of commission, charges.

Return type
(float, float)

`rollcost()`

Costs for rolling positions overnight. This is usually applicable for rolling margin trading positions (e.g Forex CFDs).

Parameters
`position` (*dict*) – Open positions.

Returns
A tuple of costs, margin.

Return type
(float, float)

Note: the input is a dictionary of current open positions - keyed by the *assets* and values are *position* objects. The *cost* in the returned tuple is the cost to charge for the roll, and *margin* is the overnight margin to settle.

`class blueshift.finance.commission.NoCommission`

Zero commission and trading charges.

`class blueshift.finance.commission.PerDollar`

Brokerage costs based on total value traded with cap and floor. This is derived from *CostModel* and takes in the same parameters. The parameter *commissions* (or *cost*) is multiplied with the traded value (quantity times the price) to determine the cost.

`class blueshift.finance.commission.PerShare`

Brokerage costs based on total quantity traded with cap and floor. This is derived from *CostModel* and takes in the same parameters. The parameter *commissions* (or *cost*) is multiplied with the traded quantity to determine the cost.

`class blueshift.finance.commission.PerOrder`

Flat brokerage costs per order. This is derived from *CostModel* and takes in the same parameters. The parameter *commissions* (or *cost*) is the flat rate per order. If an order results in multiple trades (corresponding to multiple fills), the charge is applied only once (per order).

`class blueshift.finance.commission.PipCost`

Brokerage costs based on total quantity traded with cap and floor. This is derived from *CostModel* and takes in the same parameters. The parameter *commissions* (or *cost*) is multiplied with the traded quantity to determine the cost. In addition, this also implements the *rollcost* method to calculate the overnight funding cost of carrying over the position.

Note: This cost model is suitable for Forex assets only.

6.6 Miscellaneous Constants

6.6.1 Currency

`blueshift.api.`**`CCY`**

alias of `Currency`

6.6.2 Algo Modes and Other Constants

`class blueshift.api.`**`AlgoMode`**`(`*value*`)`

Track the current running mode of algo - live or backtest. `BACKTEST` for backtesting mode and `LIVE` for live mode.

`BACKTEST = 'BACKTEST'`

`LIVE = 'LIVE'`

`PAPER = 'PAPER'`

`EXECUTION = 'execution'`

`class blueshift.api.`**`ExecutionMode`**`(`*value*`)`

Track the current execution mode of a live algo. `AUTO` stands for automatic sending of orders to the broker. `ONECLICK` means the user must confirm the order before it is send to the broker.

`AUTO = 'AUTO'`

`ONECLICK = 'ONECLICK'`

`class blueshift.api.`**`AlgoCallBack`**`(`*value*`)`

An enumeration.

`DATA = 'DATA'`

`TRADE = 'TRADE'`

Note: All these objects can be import into the strategy code from `blueshift.api`.

These constants are useful when doing different processing based on the algo mode or execution mode. Query the *`AlgoContext.mode`* for the mode of the current running algo. Query the *`AlgoContext.execution_mode`* attribute from within strategy code to ascertain the current execution mode. Example:

```python
from blueshift.api import AlgoMode

def initialize(context):
    if context.mode == AlgoMode.LIVE:
        print('some extra debug prints only during live trading ')
```

CHAPTER

SEVEN

BUILT-IN LIBRARY

The platform has an exclusive and extensive built-in library. These library (*blueshift.library*) provide a select set of functionalities unique to the platform, or wrapped from other packages for ease of use on the platform. These functions are divided in categories as follow:

- **Technicals**
  Includes technical indicators and automatic technical pattern identification functionalities.
- **Statistical**
  Include perceptually important points and change points algorithms. Also include a collection of functions that wraps various functionalities from other useful packages.
- **Pipelines**
  A useful collection of ready-to-use pipeline filtering and factoring functions.
- **Timeseries**
  A collection of functions and models useful for timeseries analysis and timeseries transformations.
- **Machine learning**
  A useful collection of ready-to-use machine learning functionalities that wraps various other useful packages.
- **Models**
  A useful collection of statistical and pricing models.
- **Execution Algorithms**
  A selection of useful execution algorithms that can be directly used in a strategy to place orders.

The features and functionalities of the blueshift library functions are frequently updated and revised.

7.1 Technical Indicators

You can import all the technical indicators supported by the TA-Lib module from *blueshift.library.technicals.indicators*.

Note: Use the uppercase name (as defined by TA-Lib) for functions that return vectorized computation. Use the lowercase name (with the same signature) for functions that return the last observation. The latter is useful for writing event-driven strategy.

The TA-Lib functions, when imported from *blueshift.library.technicals.indicators*, can automatically identify required columns from pandas DataFrame and have additional error handling. See examples below.

```python
import talib as ta
from blueshift.library.technicals.indicators import ADX, adx

def initialize(context):
    ...

def signal_function(asset, price):
    # we assume price is a dataframe with OHLC columns

    # to call the TA-Lib function, we must extract the required
    # columns
    x1 = ta.ADX(price.high, price.low, price.close)

    # not needed for the blueshift library version, the required
    # columns will be automatically extracted
    x2 = ADX(price)

    # but we can still use the TA-Lib signature if we want to
    # both will work
    x3 = ADX(price.high, price.low, price.close)

    # X1, X2 and X3 above are pandas Series. To get the last computed
    # value, use the lowercase version. X4 is a float.
    x4 = adx(price)
```

7.1.1 Additional Indicators

Apart from the TA-Lib indicators, a few additional indicators are available as below.

`blueshift.library.technicals.indicators.`**`MA_XOVER`**`(`*real*[, *ltma=?*, *stma=?*, ***kwargs=?*]`)`

Moving Average Cross-over

Inputs:
real: (any ndarray)

Parameters:
ltma: 20 stma: 5

Outputs:
real

`blueshift.library.technicals.indicators.`**`EMA_XOVER`**`(`*real*[, *ltma=?*, *stma=?*, ***kwargs=?*]`)`

Exponential Moving Average Cross-over

Inputs:
real: (any ndarray)

Parameters:
ltma: 20 stma: 5

Outputs:
real

`blueshift.library.technicals.indicators.`**`BOLLINGER_BAND_DIST`**`(`*real*[, *timeperiod=?*, *nbdevup=?*, *nbdevdn=?*, *matype=?*]`)`

Bollinger Bands Distance From Upper (percentage)

Inputs:
real: (any ndarray)

Parameters:
timeperiod: 5 nbdevup: 2 nbdevdn: 2 matype: 0 (Simple Moving Average)

Outputs:
real

`blueshift.library.technicals.indicators.`**`HEIKIN_ASHI`**`(`*real*[, *precision=?*]`)`

Heikin-Ashi

Returns heikin-ashi candles. The input price must be a dataframe with 'open', 'high', 'low' and 'close' columns. The 'volume' column, if present, will be added to the returned dataframe.

Args:
price(dataframe): input OHLC (or OHLCV) prices.

Returns:
Dataframe (OHLC or OHLCV depending on the input).

`blueshift.library.technicals.indicators.`**`ICHIMOKU_CLOUD`**`(`*real*[, *timeperiod1=?*, *timeperiod2=?*, *timeperiod3=?*, *timeperiod4=?*]`)`

Ichimoku Cloud

Returns Ichimoku cloud lines in this order - *conversion*, *base*, *spanA*, *spanB* and *lagging*.

Args:
price(dataframe): input OHLC (or OHLCV) prices.

timeperiod1(int): timeperiod for conversion line.

timeperiod2(int): timeperiod for base line.

timeperiod3(int): timeperiod for leading span B.

timeperiod4(int): timeperiod for lagging span.

Returns:
Tuple of real.

`blueshift.library.technicals.indicators.`**`TREND_STALL`**`(`*high*, *low*, *close*[, *bandwidth=?*, *timeperiod2=?*, *threshold=?*]`)`

Trend Stall indicator checks the stalling of the momentum in the price based on ADX indicator and then determines if it is a stalling of bullish or bearish trend based on ROC. If ADX has peaked out and ROC is positive (negative), it signifies stalling of a bullish (bearish) trend and has signal value -1 (+1). Ideally this signal needs further confirmation. The *bandwidth* is used to smooth the raw ADX signal and the extreme points are determines based on a neighbourhood of 2*bandwidth+1 points. Note: the timeperiod must be greater than bandwidth.

Returns Trend Stall signals.

Inputs:
price: (HLC Dataframe)

Parameters:
bandwidth=5 timeperiod: 14 threshold: 0.001

Outputs:
real

`blueshift.library.technicals.indicators.`**`TREND_SET`**`(`*high*, *low*, *close*[, *bandwidth=?*, *timeperiod2=?*, *threshold=?*]`)`

Trend Set indicator checks the beginning of a new trend in the prices based on ADX indicator and then determines if it is a starting a bullish or bearish trend based on ROC. If ADX has bottomed out and ROC is positive (negative), it signifies start of a bullish (bearish) trend and has signal value +1 (-1). Ideally this signal needs further confirmation. The *bandwidth* is used to smooth the raw ADX signal and the extreme points are determines based on a neighbourhood of 2*bandwidth+1 points. Note: the timeperiod must be greater than bandwidth.

Returns Trend Stall signals.

Inputs:
price: (HLC Dataframe)

Parameters:
bandwidth=5 timeperiod: 14 threshold: 0.001

Outputs:
real

These functions follow the same naming convention, i.e. the uppercase function names return vectorized output, and lower case for the last observation. An exception is *HEIKIN_ASHI* (returns DataFrame) which has no lowercase implementation.

7.2 Technical patterns

`blueshift.library.technicals.`**`find_support_resistance`**(*x*, *type_='pip'*, *R=None*, *scale=[1.75, 2, 2.5, 3, 5, 7]*, *tolerance=0.001*)

Find support(s) and Resistance(s) lines based on either Fibonacci or the ‘PIP’ method.

The parameter *R* is used in the case of the ‘PIP’ support and resistance method. This is the same parameter used to find the perceptually important points and should be in the form of 1+x, where x is the percentage move determining a peak or trough. For e.g. to use 2% move, use R = 1.02. R value will be automatically determined based on the input series, if set to None.

The parameter *tolerance* is used to determine the support lines. To consider a collection of peak (trough) points forming a single resistance (support) line, each point must not deviate by more than the tolerance value from a linear fit.

Args:

x (Dataframe or Series): Input data.

type_ (str): Can be either ‘fibonacci’ or ‘pip’

R (float): The returns ratio for the PIP method.

scale(float): Multiple of volatility for PIP identification.

tolerance (float): Tolerance for finding support lines.

Returns:
A list of lines object of type *Line*. For the pip method, always a pair of lines are returned. For the Fibonacci method, 6 lines, corresponding to the Fibonacci levels are returned.

`blueshift.library.technicals.`**`search_chart_patterns`**(*data*, *pattern*, *R=None*, *scale=[1.75, 2, 2.5, 3, 5, 7]*, *tolerance=None*, *find_all=False*, *adjust_trend=False*)

function to find important points based patterns. This is made for daily returns. For other frequencies, the default values of R may not be suitable. Scale range is used only when R is None or empty list.

Args:

data (frame or series): input price data.

pattern(obj or str): A pattern definition object.

R(float): R range for searching important points.

scale(float): Scale range for searching important points.

tolerance(float): For fine-tuning pattern matching.

find_all(bool): Return all or the last pattern in the sample.

adjust_trend(bool): If true, de-trend the points before matching.

Returns:

List. A list of pattern objects (empty list if no match found).

7.3 Statistical Functions

`blueshift.library.statistical.`**`find_imp_points`**`()`

Find perceptually important points (see https://www.cs.cmu.edu/~eugene/research/full/search-series.pdf)

Note: PIPs are an effort to algorithmically derive a set of important points as perceived by a human to describe a time series. This typically can be a set of minima or maxima points or a set of turning points which are important from a feature extraction perspective. Traditional technical analysis - like technical pattern identification - relies heavily on PIPs. In addition, a set of PIPs can be used to compress a time series in a very useful way. This compressed representation then can be used for comparing segments of time series (match finding) or other purposes.

Args:

x(frame or series): Input price data.

R(float): (1+x) to identify PIP with minimum x move.

scale(float): Multiple of volatility for PIP identification.

Returns:

Tuple. The first element is a list of indices for the minimum points, second is the same for maximum points. The third element of the tuple returns the compressed data (with only the peak and valleys).

`blueshift.library.statistical.`**`find_trends`**`(`*x*, *type_='price'*, *Q=10*, *minseglen=10*, *penalty=2*`)`

Find change point in price levels or variance. This implements a unique change point analysis for non-stationary time series to identify multiple changes in the deterministic linear trends. The implementation is based on identifying change in simple regression coefficients (with penalty) and extends to multiple change point identification using the popular binary segmentation methodology.

Args:

x (Dataframe or Series): Input pricing data.

type_ (str): Can be either “price” or “variance”.

Q (int): The maximum number of segments.

minseglen (int): The minimum length of a trend segment.

penalty (int): A penalty specifications usually between 2 to 10.

Returns:
A list of lines (of *LineType.trends* type).

`blueshift.library.statistical.`**`get_hmm_state`**`(`*x*, *covariance_type='full'*, *n_iter=100*`)`

Market classification based on hidden market model. The state with the lowest return is 0, and highest is 2. If we have only a single state identified, it is assigned state 1. If only two states are identified, they are assigned 0 and 2. Else 0, 1 and 2 based on the conditional expected returns.

Args:
` x (Series)`: Input pricing data.

covariance_type (str): Passed on to the underlying `hmm.GaussianHMM call.

n_iter (int): Passed on to the underlying *hmm.GaussianHMM* call.

Returns:
A Series with the inferred state of the market.

`blueshift.library.statistical.`**`hedge_ratio`**`(`*Y*, *X*`)`

Returns the ADF p-value and regression coefficient (without intercept) of regression of y (dependent) against x (explanatory).

Args:
Y (series or ndarray or list): input y series X (series or ndarray or list): input x series

Returns:
Tuple. p-Value of Augmented Dickey Fuller test on the regression residuals, and the regression coefficient.

`blueshift.library.statistical.`**`z_score`**`(`*Y*, *X=None*, *lookback=None*, *coeff=None*`)`

Given two series Y and X, and a lookback, computes the latest z-score of the regression residual (ratio of deviation from the mean and standard deviation of the residuals).

Note:
X and Y must be of equal length, lookback must be less than or equal to the length of these series.

Args:
Y (series or ndarray or list): input y series X (series or ndarray or list): input x series lookback (int): lookback for computation. coeff (float): regression coefficient.

Returns:
z-score of the regression residuals.

7.4 Timeseries Functions and Models

`blueshift.library.timeseries.`**`intraday_seasonality_func`**`(`*series*, *period=None*, *calendar=None*, *bandwidth=1.0*, *infer_frequency=True*, *drop_first_minute=True*`)`

Generate the intraday seasonality function. The input series is resampled to the target frequency, and a smoothed intraday seasonality is estimated. The returned function takes in an intraday series index (pandas DatetimeIndex) and computes the seasonality factors to be applied. The reciprocals of these factors can be used to de-seasonalize. The *bandwidth* parameter is passed on to *scipy.ndimage.gaussian_filter1d* function for kernel smoothing.

Args:

`series (pd.Series)`: Input series.

`period (str)`: Valid pandas period string for resampling.

`calendar (TradingCalendar)`: Calendar for session filtering.

`bandwidth (float)`: Bandwidth for smoothing.

`infer_frequency (bool)`: Infer input frequency (must be intraday).

Returns:
A callable of signature *f(index)* that accepts an input *DatetimeIndex* and returns the seasonality factor weight (not normalized) for each timestamp in the index.

> **Warning:** The input data as well as the input index to the returned function must have intraday frequency, which must be higher than the frequency implied by *period*.

`blueshift.library.timeseries.`**`deseasonalize`**(*series*, *seasonality_func*, *period*, *minutes_per_day=None*, *calendar=None*)

Given a *seasonality_func*, deasonalize an input series (multiplicative). The *seasonality_func* must be obtained using *intraday_seasonality_func*. The number of trading minutes per day (implied from the calendar object) should be divisible by the period.

Args:

`series (Series)`: Input timeseries to deseasonalize.

`seasonality_func (callable)`: See *intraday_seasonality_func*.

`period (str)`: A valid pandas period string (e.g. “5T”).

`calendar (TradingCalendar)`: A valid trading calendar.

Returns:
Deseasonalized input timeseries.

> **Warning:** Input series must be indexed by DatetimeIndex and should have an intraday frequency.

`blueshift.library.timeseries.`**`reseasonalize`**(*series*, *seasonality_func*, *period*, *calendar*)

Given a *seasonality_func*, re-seasonalize an input series (previously deseasonalized). The *seasonality_func* must be obtained using *intraday_seasonality_func* (and should be the same to deseasonalize it before). The number of trading minutes per day (implied from the calendar object) should be divisible by the period.

Args:

`series (Series)`: Input timeseries to deseasonalize.

`seasonality_func (callable)`: See *intraday_seasonality_func*.

`period (str)`: A valid pandas period string (e.g. “5T”).

`calendar (TradingCalendar)`: A valid trading calendar.

Returns:
Deseasonalized input timeseries.

> **Warning:** Input series must be indexed by DatetimeIndex and should have an intraday frequency.

class `blueshift.library.timeseries.`**`OnlineAutoARIMA`**(*max_coeffs=5*, *period=None*, *calendar=None*)

An online auto-arima model (compatible with sklearn) that uses *blueshift.library.timeseries.auto_arima* function to fit an initial model. The model can be updated with new incoming data by calling the *update* method

Args:

max_coeffs (int): Maximum coefficents (AR + MA).

`blueshift.library.timeseries.`**`auto_arima`**(*series*, *max_terms=5*)

Auto-ARMA model for timeseries. Parameter *max_terms* determine the total coefficients to estimates (MA + AR). This function search all combinations given a *max_terms* and selects the mode with the minimum AIC. The input series must be stationary (else raises exception). Also the selected model must have residuals with durbin-watson stats in the range of 1.95 to 2.05 (else raises exception). Returns the selected model.

Args:

series (Series): Input time series.

max_terms (int): Maximum coefficients (MA + AR).

Returns:

statsmodels.tsa.arima.model.ARIMA - the estimated ARIMA model.

Raises:

blueshift.errors.ModelError in case the model fit fails.

`blueshift.library.timeseries.transform.`**`resample`**(*df*, *period*, *fill_value=None*, *fill_method='ffill'*, *limit=None*, *default_func=<function mean>*, *calendar=None*)

Base function to resample pandas dataframe or series to different periods.

Args:

`df (dataframe or serues)`: Input time-series.

`period (str)`: Target period string.

`fill_value (number or dict)`: Fill value in *fillna* (*None*).

`fill_method (str)`: Deafault fill method in *fillna* (*ffill*).

`limit (int)`: max number to fill with method (*None*).

`default_func (function)`: Function to aggregate with (*numpy.mean*).

`calendar (TradingCalendar)`: Calendar for filtering non-trading minutes (*None*).

Returns:

Series or DataFrame. Aggregated and na-filled. If calendar is specified, returned series will include only trading hours and sessions.

Note: In addition to this function, a host of derived functions to convert to specific frequencies are available. They are *to_yearly*, *to_quarterly*, *to_monthly*, *to_weekly*, *to_daily*, *to_hourly*, *to_minutes30*, *to_minutes15*, *to_minutes10* and *to_minutes5*. All this accepts the first argument as a Series (or Frame) and compute the resampling based on the above defaults. Functions with period equal to or lower than daily (i.e. from *to_daily* to *to_minute5*) also accepts a calendar object for filtering trading days and trading minutes.

`blueshift.library.timeseries.transform.`**`endpoints`**(*df*, *on*)

get a DatetimeIndex resampled from the input series based on the period specified by *on*.

Args:

`df (dataframe or serues)`: Input time-series.

`on (str)`: Period specification.

Returns:
pandas.DatetimeIndex. Timestamp of the last observation for each period.

`blueshift.library.timeseries.transform.`**`split`**`(`*df*, *on*`)`

Split a dataframe by time periods and get a list of data frames for further analysis.

Args:
`df (dataframe or series)`: Input time-series.

`on (str)`: Period specification.

Returns:
List. A list of dataframe (series) after the split.

`blueshift.library.timeseries.transform.`**`period_apply`**`(`*df*, *on*, *func*`)`

Apply aggregation on by periods. The parameter *on* must be a valid Pandas period string. The function must accept a series and produce a single salar value.

Args:
`df (dataframe or serues)`: Input time-series.

`on (str)`: Period specification.

`func (str, function, dict)`: Method specification

Returns:
Dataframe. Aggregated dataframe.

Note: In addition to this base function, a host of useful derivatives are also available, namely *period_mean*, *period_max*, *period_min*, *period_median*, *period_prod*, *period_sum*, *period_std*, *period_var*. These accepts the input series and the *on* parameter. The functions applied to each of these cases are obvious from the function names.

`blueshift.library.timeseries.transform.`**`rollapply`**`(`*df*, *width*, *func*, *y=None*, *by=0*, *by_column=True*, *fill=True*, *partial=False*, *align='right'*, *coredata=False*, ***kwargs*`)`

Roll apply for Pandas, with R-like functionalities. If by_column is False, entire dataframe subset for the window will be available for the user supplied function `func`, otherwise `func` is applied separately to each column.

Note:
User supplied function (`func`) must return a scalar. The function must accept an input array/frame/series and optional kwargs. If *y* is not *None*, the second argument must be *y*.

Args:
`df (dataframe or series)`: Input time-series.

`width (int)`: Width of the window.

`func (function)`: A Function that returns a scalar.

`y (dataframe or series)`: Optional second input time-series.

`by (int)`: Rows to skip (will be filled with NA).

`by_column (bool)`: Applies func to each column if True

`fill (bool)`: Fill missing values if true.

`partial (bool)`: See R rollapply documentation.

`align (bool)`: See R rollapply documentation.

`coredata (bool)`: Feeds the numpy array to func if true.

Returns:
Dataframe or series. Returns results of applying func.

Note: In addition to this base function, a list of derivatives are available as well, namely *run_sum*, *run_prod*, *run_min*, *run_max*, *run_mean*, *run_median*, *run_std*, *run_var*, *run_skew*, *run_kurt*, *run_cov*, *run_corr*. These functions accept an input series (or frame) and a window size (*n*). If the third parameter *cumulative* is True, the result if cumulative application with minimum size n (all values before becoming NaN).

`blueshift.library.timeseries.transform.`**`cumulative_apply`**(*df*, *func*, *min_period=1*, *by_column=True*, *fill=True*, *partial=False*, *align='right'*, *coredata=False*, ***kwargs*)

R-style reduce for Pandas, with R-like functionalities. If by_column is False, entire dataframe subset for the window will be available for the user supplied function `func`, otherwise `func` is applied separately to each column.

Note:
User supplied function (`func`) must return a scalar.

Args:
`df (dataframe or series)`: Input time-series.

`func (function)`: A Function that returns a scalar.

`min_period (int)`: Mimimum width of the window.

`by_column (bool)`: Applies func to each column if True

`fill (bool)`: Fill missing values if true (locf).

`partial (bool)`: Ignored.

`aligh (bool)`: See R rollapply documentation.

`coredata (bool)`: Feeds the numpy array to func if true.

Returns:
Dataframe or series. Returns results of applying func.

Note: For the derivative functions, see *roll_apply*.

7.5 Statstical and Pricing Models

`blueshift.library.models.bs.`**`bs_plain_vanilla_option`**(*atmf*, *strike*, *imp_vol*, *time*, *option_type*, *annualization_factor*)

Black 76 plain vanilla European options pricing. Parameter *time* can be a float (in years to expiry) or a pandas Timedelta object or a Timestamp (tz-aware). Parameter *option_type* can be either a *OptionType* enumeration or one of [“CALL”, “PUT”].

Inputs:
atmf: (float) strike: (float) imp_vol: (float) time: (float) option_type: (str) annualization_factor: (float)

Outputs:
real

`blueshift.library.models.bs.`**`bs_implied_vol`**`(`*atmf*, *strike*, *price*, *time*, *option_type*, *annualization_factor*`)`

Black 76 plain vanilla European options implied volatility. Parameter *time* can be a float (in years to expiry) or a pandas Timedelta object or a Timestamp (tz-aware). Parameter *option_type* can be either a *OptionType* enumeration or one of [“CALL”, “PUT”].

Inputs:
atmf: (float) strike: (float) price: (float) time: (float) option_type: (str) annualization_factor: (float)

Outputs:
real

`blueshift.library.models.bs.`**`bs_plain_vanilla_greek`**`(`*atmf*, *strike*, *imp_vol*, *time*, *option_type*, *annualization_factor*`)`

Black 76 plain vanilla European options greek. Parameter *time* can be a float (in years to expiry) or a pandas Timedelta object or a Timestamp (tz-aware). Parameter *option_type* can be either a *OptionType* enumeration or one of [“CALL”, “PUT”]. Supported greeks are one of [‘DELTA’, ‘VEGA’,’THETA’ and ‘GAMMA’].

Inputs:
atmf: (float) strike: (float) imp_vol: (float) time: (float) option_type: (str) greek: (str) annualization_factor: (float)

Outputs:
real

7.6 Machine Learning Functions

`blueshift.library.ml.`**`estimate_random_forest`**`(`*df*`)`

Estimate random forest regression for input DataFrame, assuming the last column to be the predicted variable, and everything else are predictors.

Args:
df (DataFrame): Merged frame of X and Y of training set.

Returns:
A random forest fitted model based on the input dataframe.

`blueshift.library.ml.`**`predict_random_forest`**`(`*regr*, *df*`)`

Forecast using a fitted model, assuming the last row in the input DataFrame are the observations to be predicted and the last but one column are the predictors in the model.

Args:
regr (object): A model object for prediction

df (DataFrame):

7.7 Pipeline Functions

`blueshift.library.pipelines.`**`select_universe`**`(`*lookback*, *size*`)`

Returns a custom filter object for volume-based filtering.

Args:
lookback (int): lookback window size *size (int)*: Top n assets to return.

Returns:
A custom filter object

```
# from library.pipelines.pipelines import select_universe

pipe = Pipeline()
top_100 = select_universe(252, 100)
pipe.set_screen(top_100)
```

`blueshift.library.pipelines.`**`average_volume_filter`**`(`*lookback*, *amount*`)`

Returns a custom filter object for volume-based filtering.

Args:
lookback (int): lookback window size *amount (int)*: amount to filter (high-pass)

Returns:
A custom filter object

```
# from library.pipelines.pipelines import average_volume_filter

pipe = Pipeline()
volume_filter = average_volume_filter(200, 1000000)
pipe.set_screen(volume_filter)
```

`blueshift.library.pipelines.`**`filter_assets`**`(`*func=None*`)`

Returns a custom filter object to filter assets based on a user supplied function. The function must return *True* for assets that are selected and *False* for assets to be filtered out. It should accept a single argument (an asset object).

Args:
func (callable): A function for filtering.

Returns:
A custom filter object

```
# from library.pipelines.pipelines import filter_assets
# from blueshift.assets import Equity
# context.universe = [symbol(AAPL), symbol(MSFT)]

pipe = Pipeline()
# filter out non-Equity assets
func = lambda asset:isinstance(asset, Equity)
asset_filter = filter_assets(func)
pipe.set_screen(asset_filter)
```

`blueshift.library.pipelines.`**`filter_universe`**`(`*universe*`)`

Returns a custom filter object to filter based on a user supplied list of assets objects. This is useful where we still want to use the underlying pipeline computation facilities, but want to specify assets explicitly.

Args:
universe (list): A list of asset objects to keep.

Returns:
A custom filter object

```
# from library.pipelines.pipelines import filter_universe
# context.universe = [symbol(AAPL), symbol(MSFT)]

pipe = Pipeline()
universe_filter = filter_universe(context.universe)
pipe.set_screen(universe_filter)
```

`blueshift.library.pipelines.`**`exclude_assets`**`(`*universe*`)`

Returns a custom filter object to filter based on a user supplied list of assets objects to exclude.

Args:
universe (list): A list of asset objects to exclude.

Returns:
A custom filter object.

```
# from library.pipelines.pipelines import filter_universe
# context.exclude = [symbol(AAPL), symbol(MSFT)]

pipe = Pipeline()
exclude_filter = filter_universe(context.exclude)
pipe.set_screen(exclude_filter)
```

`blueshift.library.pipelines.`**`returns_factor`**`(`*lookback*, *offset=0*`)`

Returns a custom factor object for computing simple returns over a period (*lookback*).

Args:
lookback (int): lookback window size *offset (int)*: offset from the end of the window

Returns:
A custom factor object.

```
# from library.pipelines.pipelines import returns_factor
pipe = Pipeline()
momentum = returns_factor(200)
pipe.add(momentum,'momentum')
```

`blueshift.library.pipelines.`**`filtered_returns_factor`**`(`*lookback*, *filter_*, *offset=0*`)`

Returns a custom factor object for computing simple returns over a period (*lookback*), with a volume filter applied. Equivalent to separately applying *period_returns* and *average_volume_filter* above.

Args:
lookback (int): lookback window size *filter_ (CustomFilter)*: a custom volume filter *offset (int)*: offset from the end of the window

Returns:
A custom factor object.

```
# from library.pipelines.pipelines import average_volume_filter, period_returns2

pipe = Pipeline()
```

(continues on next page)

(continued from previous page)

```
volume_filter = average_volume_filter(200, 1000000)
momentum = filtered_returns_factor(200,volume_filter)
pipe.add(momentum,'momentum')
```

`blueshift.library.pipelines.technical_factor`(*lookback*, *indicator_fn*, *indicator_lookback=None*)

A factory function to generate a custom factor by applying a user-defined function over asset returns.

Args:

lookback (int): lookback window size *indicator_fn (function)*: user-defined function *indicator_lookback (int)*: lookback for user-defined function.

Returns:

A custom factor object applying the supplied function.

Note:

The *indicator_fn* must be of the form f(px, n), where px is numpy ndarray and lookback is an n. Also the *lookback* argument above must be greater than or equal to the other argument *indicator_lookback*. If *None* it is set as the same value of *lookback*.

```
# from library.pipelines.pipelines import technical_factor

pipe = Pipeline()
rsi_factor = technical_factor(14, rsi)
pipe.add(rsi_factor,'rsi')
```

7.8 Execution Algorithms

These execution algorithms are designed to be used with the *order* to place advanced algorithmic orders. These advanced order objects are different from the simpler *Order* object, but follow similar interface and attributes with equivalent meaning.

`class blueshift.library.algos.PassiveAggressiveOrder`(*asset: Asset*, *quantity: int*, *limit_price: float*, *timeout: int = 30*, *convert_to_market: bool = False*, ***kwargs*)

Algo Order that places a limit order, and waits for the duration of the order as specified (*timeout*). At the end of the specified duration, if the order is still not complete, we cancel the limit order, and optionally replace the remaining part to a market order.

Args:

asset (Asset): Asset for the order.

quantity (int): Quantity for the order.

limit_price (number): Limit price for the order.

timeout (int): Timeout in minutes before cancellation.

convert_to_market (bool): Convert to market order if not filled.

Optional keywords arguments as applicable for ordering functions can be passed on as well. A positive quantity is a buy order.

`class blueshift.library.algos.AdaptiveOrder`(*asset: Asset*, *quantity: int*, *price_offset: float*, *timeout: int = 30*, *delay: float = 1*, *convert_to_market: bool = True*, *offset_decay=None*, ***kwargs*)

Adaptive Order is a passive-aggressive style where we first place a limit order off by *offset* amount of from the best bid (offer) for a buy (sell) order (in the favour of the direction of the order). For the duration of the strategy (*timeout*), we periodically (*delay*) look at the fill and update the order limit price - by adjusting the bid (offer) at the current market level. If by the end of *timeout* the period, the order is still not filled (or partially filled), we cancel the initial limit order and optionally replace it with a market order with the remaining amount. If *offset_decay* is supplied, it must be a positive number less than 1, and for each order modification, the current price offset is multiplied by this value. A low offset decay moves fast to the best bid/offer and a high value moves to the best bid/offer slowly. Omit this parameter to keep the same offset for all order modification requests.

Note: Fractional values are allowed for *delay*.

Args:

asset (Asset): Asset for the order.

quantity (int): Quantity for the order.

price_offset (number): Offset from the best bid or offer.

timeout (int): Timeout in minutes before cancellation.

delay (number): Number of minutes to wait before update.

convert_to_market (bool): Convert to market order if not filled.

Optional keywords arguments as applicable for ordering functions can be passed on as well. A positive quantity is a buy order.

`class blueshift.library.algos.`**`MarketIfTouched`**(*asset:* Asset, *quantity:* *int*, *target_price:* *float*, *timeout:* *int* = *30*, *delay:* *float* = *1*, ***kwargs*)

Conditional order waits for a specific condition to meet before placing the order (at limit or at market). The algo waits for a *timeout* number of minutes before the condition is met, else gets cancelled. It accepts a callable (*condition*) of the signature *f(context, data)* and must evaluate to True if the condition is fulfilled. The parameter *delay* determines the delay (minutes) between two successive condition checks. For Market-If-Touched orders, if the specified *target_price* is reached, a market order is triggered.

Note: Fractional values are allowed for *delay*.

Args:

asset (Asset): Asset for the order.

quantity (int): Quantity for the order.

target_price (float): Price that triggers the order.

timeout (int): Timeout in minutes before cancellation.

delay (number): Number of minutes between consecutive checks.

Optional keywords arguments as applicable for ordering functions can be passed on as well. A positive quantity is a buy order.

`class blueshift.library.algos.`**`LimitIfTouched`**(*asset:* Asset, *quantity:* *int*, *target_price:* *float*, *limit_price:* *float*, *timeout:* *int* = *30*, *delay:* *float* = *1*, ***kwargs*)

Conditional order waits for a specific condition to meet before placing the order (at limit or at market). The algo waits for a *timeout* number of minutes before the condition is met, else gets cancelled. It accepts a callable (*condition*) of the signature *f(context, data)* and must evaluate to True if the condition is fulfilled. The parameter *delay* determines the delay (minutes) between two successive condition checks. For Limit-If-Touched orders, if the specified *target_price* is reached, a limit order is triggered at the *limit_price*.

Note: Fractional values are allowed for *delay*.

Args:

asset (Asset): Asset for the order.

quantity (int): Quantity for the order.

target_price (float): Price that triggers the order.

limit_price (number): Limit price for the order.

timeout (int): Timeout in minutes before cancellation.

delay (number): Number of minutes between consecutive checks.

Optional keywords arguments as applicable for ordering functions can be passed on as well. A positive quantity is a buy order.

`class blueshift.library.algos.IcebergOrder`(*asset: Asset, quantity: int, slices: int, order_type: ['market', 'limit'] = 'market', price_offset: float = 0, timeout: int = 30, delay: float = 1, cancel_on_timeout: bool = False, **kwargs*)

Iceberg order breaks up a large order to a number of (*slices*) smaller orders and place each of them one by one. The individual order can be either regular or adaptive (*order_type*). For regular order, if *limit_price* is specified, each order is placed as limit order at the same price. Else for regular order, it becomes a market order. For adaptive orders, *price_offset* is required - the offset (in our favour, can be negative) to apply to the current best bid (buy or offer (sell). If the whole order is not complete by the time specified by *timeout* (in minutes), the pending orders are cancelled if *cancel_on_timeout* is True, else they are left as is. The *delay* parameters determines the minimum delay between successive orders. The next order is sent if the previous order is filled AND the delay time has elapsed. If delay is 0, the next order is sent as soon as the previous one is filled.

Args:

asset (Asset): Asset for the order.

quantity (int): Quantity for the order.

slices (int): A callable to evaluate the condition.

order_type (str): Order type, can be either 'market' or 'limit'.

price_offset (number): Limit price offset for the order.

timeout (int): Timeout in minutes before cancellation.

cancel_on_timeout (bool): Cancel pending orders if timed-out.

Optional keywords arguments as applicable for ordering functions can be passed on as well. A positive quantity is a buy order.

An example of use case is shown below

```python
from blueshift.api import symbol, schedule_function, date_rules, time_rules
from blueshift.api import get_datetime, order, terminate, schedule_once

from blueshift.library.algos import IcebergOrder
```

(continues on next page)

(continued from previous page)

```python
def initialize(context):
    asset = symbol('ACC')

    algo = IcebergOrder(asset, 50, 10, order_type='market', timeout=300)
    context.algo = algo

    schedule_once(place_order)
    schedule_function(strategy,
                      date_rules.every_day(),
                      time_rules.every_nth_minute(1))

def place_order(context, data):
    order(context.algo)

def strategy(context, data):
    if not context.algo.is_open():
        msg = f'{get_datetime()}:exiting strategy, algo execution terminated.'
        terminate(msg)
```

In the above example, the strategy instantiates an Iceberg order and places the order as soon as possible. Once the order is placed, it periodically checks the status, and once completed, exits the strategy.

7.9 Library Objects

class blueshift.library.common.**Signal**(*value*)

An enum for trading signal. You can use the values of these enums for placing order directly. Although all three of *EXIT*, *LONG_EXIT* and *SHORT_EXIT* has numerical value of 0, they still are distinct enumerations. No value is defined for the member *NO_SIGNAL*, do not attempt to use its value for any computation.

LONG_ENTRY = 1

LONG_ENTRY_STRONG = 2

SHORT_ENTRY = -1

SHORT_ENTRY_STRONG = -2

LONG_EXIT = 0

SHORT_EXIT = 0

EXIT = 0

NO_SIGNAL = NO_SIGNAL

class blueshift.library.common.**LineType**(*value*)

Enum for line types generated by *find_support_resistance* as well as *find_trends* functions.

SUPPORT = 'support'

RESISTANCE = 'resistance'

`TREND = 'trend'`

class `blueshift.library.common.`**`Line`**(*points*, *type_*, *score=0*, *fit=True*)

A class to capture important points technical support or resistance lines. The param *points* is a series (of points that makes up the line). The parameter *type_* can be either string "support" or "resistance" or of Enum type SupportType.

`type`

Line type of this line.

`line`

Returns the pandas Series that represents the line.

`score`

User defined score, or the R-squared fit for the given points.

`slope`

The slope of the line.

`intercept`

The intercept of the line.

`is_breakout`(*level*, *dt*)

Given the current level and current date (date-like or timestamp), find if we have a break-out. This is only applicable for line types SUPPORT and RESISTANCE. A break is breaking below support or breaking above resistance.

`get_level`(*dt*)

Given a date-like or timestamp, find the implied level from the line.

`plot`()

Use Pandas *plot* method to plot the underlying line.

class `blueshift.library.common.`**`Pattern`**(*name*, *points*, *lines*, *level*, *aspect*)

A class to capture important points based patterns. This has the following attributes

- *name*: is the name of the pattern type,
- *points*: are the price points (important points) that make the pattern.
- *lines*: A list of lines that define the pattern.
- *level*: is usually the level that is watched for breach.
- *aspect* is any (usually dimension-less) feature of the particular pattern that is deemed important.

The *lines* parameters are a list of *Line* objects (may be an empty list for patterns purely defined by important points).

`plot`()

Use Pandas *plot* method to plot the pattern.

CHAPTER

EIGHT

ERRORS AND EXCEPTIONS

Errors and exceptions, including from user strategy, are caught by the platform engine and handled based on their category and recoverability. If an error is recoverable, Blueshift logs the error details and continues to run. If it is irrecoverable, Blueshift executes the exit processes (graceful exit, saving data etc.). Errors are usually classified based on their types and sources, as described below.

- **User Strategy Errors**
  Any errors generated from user strategy code are considered fatal and no attempt is made to recover.
- **Recoverable Errors**
  Errors due to a drop in connection, or from broker API (e.g. invalid or illegal orders, bad data from broker etc.) are considered recoverable. If a recoverable error is raised, the event loop stops the current iteration at that point, and starts the next iteration as usual in the next cycle. If the error continues to repeat, after a certain threshold, it is upgraded to irrecoverable error.
- **Irrecoverable Errors**
  Any errors during the Blueshift event loop initialization is considered irrecoverable in a fail-fast manner (including connection errors, API errors or validation failures). Once the event loop starts normal cycles, any unexpected error (e.g. disk full, or server crash) or any error upgraded (as described above) is considered fatal.

8.1 Error Handling in Strategy

It is usually a good idea to handle possible errors in the strategy itself, rather than leaving it to the platfrom. Strategy code can use the Python try-except block to achieve the same. Below are a list of useful errors that can be handled from within the strategy.

`class blueshift.errors.`**`ValidationError`**`(`**args*, ***kwargs*`)`

Validation failed. Raise this exception to flag invalid inputs or parameters.

`class blueshift.errors.`**`SymbolNotFound`**`(`**args*, ***kwargs*`)`

Symbol requested does not exist. Usually raised when *symbol* function fails.

`class blueshift.errors.`**`ServerError`**`(`**args*, ***kwargs*`)`

Received an error response from broker data or trade API.

`class blueshift.errors.`**`IllegalRequest`**`(`**args*, ***kwargs*`)`

Illegal parameters for API request.

`class blueshift.errors.`**`BrokerError`**`(`**args*, ***kwargs*`)`

Something went wrong while connecting to the broker or fetching data over broker API.

`class blueshift.errors.`**`APIError`**`(`**args*, ***kwargs*`)`

Broker server sent an error response, either because of invalid or illegal input parameters, or the server failed to respond temporarily.

`class blueshift.errors.`**`BrokerConnectionError`**`(`**args*, ***kwargs*`)`

Error in connecting to broker servers.

`class blueshift.errors.`**`InsufficientFund`**`(`**args*, ***kwargs*`)`

Insufficient fund in account. Could not complete the transaction.

`class blueshift.errors.`**`TradingControlError`**`(`**args*, ***kwargs*`)`

A trading risk control check failed. See *Risk Management APIs*.

`class blueshift.errors.`**`BadDataError`**`(`**args*, ***kwargs*`)`

Received malformed data from broker server.

`class blueshift.errors.`**`NoDataForAsset`**`(`**args*, ***kwargs*`)`

Data query extended beyond available start date. See *data.history*.

`class blueshift.errors.`**`HistoryWindowStartsBeforeData`**`(`**args*, ***kwargs*`)`

Data query in pipelines extended beyond available start date. See *Pipeline APIs*.

`class blueshift.errors.`**`NoSuchPipeline`**`(`**args*, ***kwargs*`)`

Pipeline requested is not registered. See *Pipeline APIs*.

In addition to error handling, it is highly recommended that strategy code also applies input data sanity checks.

CHAPTER

NINE

HOW-TOS AND EXAMPLES

9.1 How to code a trading strategy on Spring

On the platform you can use the full power of Python to code your strategy logic. To do that, you follow - roughly - the following steps

- **Have a clearly defined strategy logic**
  The platform will run your strategy as you have coded it. Make sure your strategy logic clearly identifies all scenarios and has an appropriate logical flow.
- **Identify instruments, input data and variables**
  Clearly identify the assets that you are going to trade, the data that is required to generate trade entry/ exit and any variables you need to track.
- **Identify the events handlers that suit the strategy**
  The platform offers a number of ways to respond to the market with different choice of event handlers. Choose the one that suits your case the best
- **Initialize your strategy properly**
  Use the *initialize* to make sure your strategy has a proper starting state. For example define your trading assets, as well as initialise the variables you need to track. You can optionally `parameterise` your strategy here.
- **Write efficient and robust strategy**
  Use the event handlers from step 3 to write down the strategy. Separate the parts of the logic in individual functions (so that it is easy to debug and easy to tweak). Fetch data only once in each of the strategy iterations (instead of in each function where this data is used) and pass on to different functions. Choose the order placing functions correctly, depending on your strategy logic. Also, validate data to check for missing values and stale data.

Below are some guidelines on various steps involved in the process of writing an effective strategy.

9.2 What is the Python support on Spring

The platform support all legal Python code (version 3.6 or higher) subject to a few restrictions. It has a comprehensive collection of *white-listed* modules that you can import as usual.

package	use case
bisect	An useful array sorting package.
cmath	Provides access to mathematical functions for complex numbers.
cvxopt	Package for convex optimization.
cvxpy	A “nice and disciplined” interface to cvxopt.
datetime	For manipulating dates and times in both simple and complex ways.
functools	Higher-order functions and operations on callable objects.
hmmlearn	For unsupervised learning and inference of Hidden Markov Models.
hurst	for analysing random walks and evaluating the Hurst exponent.
arch	ARCH and other tools for financial econometrics.
keras	A deep learning API running on top of TensorFlow.
math	Provides access to the mathematical functions defined by the C standard.
numpy	Package for scientific computing with Python.
pandas	High-performance, easy-to-use data structures and data analysis tools.
pykalman	Implements Kalman filter and Kalman smoother in Python.
pytz	Allows accurate and cross platform timezone calculations.
random	Random number generators for various distributions.
scipy	Efficient numerical routines for scientific computing.
sklearn	For machine learning in Python.
statsmodels	For statistics in Python.
talib	For technical analysis in Python.

This covers a range of useful modules from technical indicators to advanced machine learning programs. If you attempt to import and use any other packages not listed here, you will get an import error.

There are a few other restrictions as listed below.

- certain built-in functions (e.g. *type*, *dir* etc.) are restricted on the platform.
- identifier (variables, functions etc.) names should not start or end with underscore.
- while you can use the `print` function in your strategy code, the maximum output is restricted.
- async programming and generator functions are not allowed on the platform. Also looping with *while* is banned, use a for loop instead.

9.3 How to create and use variables

The strategy code is a collection of functions that are called by the Blueshift event loop at appropriate times. You can use the normal Pythonic way to create and use local variables for use within each individual function. For accessing the same variables across functions, we recommend using the *context* variable. Since this is a Python object, you can add attributes to it to store your variable. Also since this variable is passed in all the *event callbacks*, you can access this variable and its attributes in all functions. This makes it a superior way to pass around variables across your strategy functions (instead of using, say, global or module-level variables).

Note: There are certain restrictions on variable names that you can use. Apart from being a legal Python identifier, it also must not start or end with underscore (‘_’). In addition, there are some built-in *attributes* of the context variable and user variable name should not clash with them (else the strategy will crash with errors).

See the point on asset fetching below to see an example.

9.4 How to fetch assets in strategy code

Use the *symbol* API function to convert an asset ticker or symbol to an *asset* object. This object can then be used in any API functions (e.g. to place order or fetch data) that require an *asset* object as an input. To use this function, import it from the `blueshift.api` module in your strategy code.

```
from blueshift.api import symbol, order_target
from blueshift.api import get_datetime

def initialize(context):
    # convert the ticker "TCS" to the TCS asset
    # we can also shorten it by direct assignment
    # context.asset = symbol('TCS')
    asset = symbol('TCS')
    context.asset = asset

def handle_data(context, data):
    # maintain 1 unit position in TCS stock
    order_target(context.asset, 1)
```

For more on what symbol to use for an asset, please see *symbology*.

Note, here we are using the *context* object as a store of strategy variables (the asset(s) to trade in this case). We can use the *context* object for all variables our strategy needs to track.

9.4.1 Fetching Equity Futures instruments

Strategy code can fetch futures instruments as either dated or rolling assets. For dated instruments, specify the ticker as `SYM<YYYYMMDD>`, where *SYM* is the underlying symbol. For rolling futures, use `SYM-I` for the first futures (near-month) and `SYM-II` for the far-month.

```
from blueshift.api import symbol

def initialize(context):
    acc_dated_futures = symbol('ACC20210826')
    acc_first_futures = symbol('ACC-I')
    acc_second_futures = symbol('ACC-II')
```

Important: Rolling futures will always be resolved to dated futures in live trading and the positions will be tracked in terms of the dated futures. For backtesting, positions are tracked in terms of rolling futures. Also placing order with rolling assets may be restricted after a cut-off period each trading day if the underlying broker requires it. The cut-off time is typically 15 minutes before the market close.

9.4.2 Fetching Equity Options instruments

Fetching options instruments are similar to futures. Use the symbology `SYM<YYYYMMDD>TYPE<STRIKE>` to fetch a specific option, where *SYM* is the underlying *<YYYYMMDD>* is the expiry date, *TYPE* is the option type (can be either `CE` or `PE` for call and put respectively) and *STRIKE* is the strike price without any leading or trailing zeros. For rolling options, replace the expiry with expiry identifier. For strikes specified in terms of offset, replace the *STRIKE* part with offset specifications as described in the *symbology*.

```
from blueshift.api import symbol

def initialize(context):
    # Nifty Aug 21 call at 16000 strike
    asset1 = symbol('NIFTY20210826CE16000')
    # Nifty near-month ATMF+100 call
    asset2 = symbol('NIFTY-ICE+100')
    # Nifty current-week ATMF put
    symbol('NIFTY-W0PE-0')
```

9.5 How to place orders

Use the *ordering functions* for placing orders. The first argument must be an asset object. Blueshift offers a number of ways to place orders, including auto-sizing and targeting orders.

We recommend *target order* functions for placing orders from a strategy. The family of targeting order functions work by checking the current positions and outstanding orders for the asset at the time of placing order, and place orders for incremental amounts, if any, to achieve the specified target. The target can be in terms of units, or percent of the current portfolio value, or total value of the required position in the specified asset. An example is given below.

```
from blueshift.api import symbol, order_target

def initialize(context):
    context.asset = symbol('TCS')

def handle_data(context, data):
    # maintain 1 unit position in TCS stock
    order_target(context.asset, 10)
```

In the above example, the *handle_data* function is called every minute, which in turn calls the *order_target* API function. The first time this function is called, a new order for 10 stocks of TCS is placed. The next time this function is called (and in any subsequent calls), the target, i.e. 10 stocks of TCS in our algo positions is already achieved. So the incremental quantity required is 0, and hence no further orders are sent out to the broker anymore, as long as the target position is maintained. This works very differently if we did not use a target function. For example, if we used simply the basic *order* function, for each call (initial or subsequent) a fresh order of 10 units will be sent to the broker.

Let's look at another example.

```
from blueshift.api import symbol, order_target_value, schedule_function
from blueshift.api import date_rules, time_rules

def initialize(context):
    context.asset = symbol('TCS')
    schedule_function(rebalance, date_rules.every_day(),
        time_rules.market_close(hours=2, minutes=30))
```

(continues on next page)

(continued from previous page)

```python

def rebalance(context, data):
    # maintain INR 10,000 position in TCS stock
    order_target_value(context.asset, 10000)
```

In this example, the rebalance function is called everyday, 2.5 hours before the market close. For the first time, this will place an order worth INR (broker currency) 10,000 of TCS shares. In subsequent calls, if the market price of TCS shares remain unchanged, no further orders will be sent. If the prices go down, the positions will fall below 10,000 and to maintain the target, algo will send buy orders to achieve 10,000 in value. If the prices go up and the opposite will happen (sell orders).

Order targeting in terms of portfolio percent works similarly, but to safeguard against market movement and order failing due to lack in buying power, a haircut (usually 2% of the current portfolio value) is applied before calculating the required quantities. For example, if the portfolio value is $10,000 and an order target percent of 0.25 is specified, the computed target value will be $10,000 (*portfolio value*) X 0.98 (*haircut*) X 0.25 (*target*) or $2450. From there it will follow the order target value behaviour as above. Note, a value or percent target does not guarantee the value of the resulting positions or the execution price.

Important: We recommend using targeting functions for placing order, unless there is a strong reason not to. This reduces the chance of an order machine-gunning (sending the same order many times over, due to bugs in strategy logic).

Important: An order, when filled (partially or fully), results in a position. The asset used in the order function may be different from that of the position it creates. This is true for rolling futures and options. For futures, backtest will create the same (rolling) asset positions, but in live trading, they will be in dated futures. For option positions are always dated instruments. For equities, they are usually the same assets, but if you specify *product_type* (for e.g. *margin*), that may create a different asset for the position (*EquityMargin*, for e.g.).

9.6 How to fetch price data for signal generation

Use the *data* object for fetching historical or current data points. See examples below.

```python
from blueshift.api import symbol

def initialize(context):
    context.assets = [symbol('TCS'), symbol('WIPRO')]

def handle_data(context, data):
    prices = data.history(context.assets, 'close', 100, '1m')
    for asset in context.assets:
        sig = generate_signal(prices[asset])

def generate_signal(price):
    sig = 0
    # apply your data analysis logic here
    # note, in this case price is a pandas series with the
    # closing price of the asset
    return sig
```

Note, although we are using price data for each asset in the *generate_signal* function (a custom function we created), the data query is done in one place, and for all assets together. Also, since we are using only the 'close' price, we queried only for that field. This is an efficient way to query data (instead of calling *data.history* for each asset separately inside the *for* loop or inside the *generate_signal* function). For more on how to query data see *data.current* and *data.history*.

Warning: Note, the *current* and the *history* method returns different types of objects based on the types of the input arguments. The returned object type is the simplest possible, depending on the number of assets, number of fields queried and whether we asked for current or historical data. see the function documents for the expected returned data type.

9.7 How to write strategy code

Blueshift is an event driven engine. Use the *event callbacks* to write your strategy logic. Your strategy should always include the *initialize* function (otherwise it is **NOT** a valid Blueshift strategy). Based on your underlying trading logic, you have a number of options to arrange your strategy flow. Below are some examples, that assume we have a signal function as below that checks the asset prices and determines if a trade to be initiated or not.

```python
import talib as ta # import the ta-lib for RSI calculation

def signal_func(asset, price):
    # TODO: enter your trading logic here. The `price`
    # parameter is assumed to be a pandas series with closing
    # price for the assets at 1 minute candles. Below example
    # shows a simple RSI based entry logic and assume the asset
    # is shortable - i.e. margin equities or F&Os
    rsi = ta.RSI(price, 14)
    if rsi < 30:
        return 1  # buy signal
    elif rsi > 70:
        return -1 # sell signal
    else:
        return 0  # neutral signal
```

In the above example, the signal function evaluates a simple RSI based entry condition.

Danger: Note the above signal function does not trigger only on cross-over but for the entire duration the condition is true. For example, it will trigger a buy signal as long as RSI<30, not just the first time it crosses below 30. If you trigger a basic *order*, it will generate a fresh order each time the signal function is evaluated (not just when the cross-over happens). The appropriate order function in this case is *targeting functions* Alternatively, you can modify the above function to trigger only on cross-over (by remembering the last RSI value in the strategy code, for e.g. storing it as a context variable attribute).

9.7.1 Strategy that trades periodically

Strategies that run (check trading signal and enter a position) on a periodic basis are best handled by the *schedule_function*. Assume our strategy checks for entry/ exit every 5 minutes. We can code that as shown below

```python
import talib as ta
from blueshift.api import symbol, schedule_function
from blueshift.api import date_rules, time_rules

def initialize(context):
    context.freq = 5
    context.quantity = 1
    context.assets = [symbol('TCS'), symbol('WIPRO')]

schedule_function(rebalance, date_rules.every_day(),
        time_rules.every_nth_minute(context.freq))

def rebalance(context, data):
    prices = data.history(context.assets, 'close', 50, '1m')
    for asset in context.assets:
        price = prices[asset]
        signal = signal_func(asset, price)
        order_target(asset, signal)

def signal_func(asset, price):
    rsi = ta.RSI(price, 14)
    if rsi < 30:
        return 1  # buy signal
    elif rsi > 70:
        return -1 # sell signal
    else:
        return 0  # neutral signal
```

Note that we have initialised the strategy in the *initialize* function that defines the stocks we want to trade and also the trade frequency and the trade size. Secondly, we have split the logic in functions - for this simple case, only two (*rebalance* and *signal_func*). Finally, we are querying data efficiently, only once per iteration (per trade frequency) and fetching data for all assets at one go.

9.7.2 Strategy that trades conditionally

Sometimes, we may have to enter or exit based on condition or state of the algo. We tweak the above RSI strategy for this example: we still use the same signal function, but want to enter once (and hold), and only in one stock (whichever triggers the RSI condition first). We can code this strategy as follows.

```python
import talib as ta
from blueshift.api import symbol, schedule_once, schedule_later
from blueshift.api import date_rules, time_rules

def initialize(context):
    context.freq = 5
    context.quantity = 1
    context.traded = False
    context.assets = [symbol('TCS'), symbol('WIPRO')]
```

(continues on next page)

(continued from previous page)

```
    schedule_once(rebalance)

def rebalance(context, data):
    if context.traded:
        # do nothing if already traded
        return

    # not traded, check for RSI signal
    prices = data.history(context.assets, 'close', 50, '1m')
    for asset in context.assets:
        price = prices[asset]
        signal = signal_func(asset, price)

        if signal !=0:
            # if an entry signal, place the order and mark
            # traded, break out of the for loop
            order_target(asset, signal)
            context.traded = True
            break

    if not context.traded:
        # if not traded, schedule itself again to run in 5 minutes
        schedule_later(rebalance, context.freq)

def signal_func(asset, price):
    rsi = ta.RSI(price, 14)
    if rsi < 30:
        return 1  # buy signal
    elif rsi > 70:
        return -1 # sell signal
    else:
        return 0  # neutral signal
```

Note that in the above example, we use a combination of *schedule_once* and *schedule_later* to run the *rebalance* function conditionally. This capability gives a powerful way to express your strategy logic.

9.8 How to check order status

There are roughly two ways to do that. We can either use the *get_open_orders* to fetch a dict (keyed by order IDs) of all orders currently open (i.e. not completed, cancelled, or rejected). Else, we can use the *get_order* function to fetch an order by its order ID. Check the *status* attribute of the *order* object to know its status. See *OrderStatus* to know how to interpret it.

9.9 How to check open positions

See *here* for more details and code sample.

9.10 How to use stoploss and take-profit

On Blueshift, adding a *stoploss* or a *take-profit* target is just a convenience API function that automatically checks the price level at the frequency of the event loop (i.e. one minute). Additionally, it also enforces a cool-off period (typically 30 minutes). Example below shows how to add a stoploss and take-profit to our original RSI strategy above (shows only the relevant part).

```
from blueshift.api import set_stoploss, set_takeprofit

def rebalance(context, data):
    prices = data.history(context.assets, 'close', 50, '1m')
    for asset in context.assets:
        price = prices[asset]
        signal = signal_func(asset, price)
        order_target(asset, signal)
        set_stoploss(asset, 'PERCENT', 0.01) # stoploss of 1%
        set_takeprofit(asset, 'PERCENT', 0.01) # stoploss of 1%
```

If we are exiting a position by a signal (i.e. not triggered by a stoploss or a take-profit exit), it is recommended to remove the corresponding stoploss and take-proft targets as well. This makes the algo run more efficiently.

Warning: The asset to place order and the asset to track the stoploss or take-profit must be consistent. See the caveat under *placing trades* for more. It is usually safer to query positions and then place stoploss or take-proft on the assets from the positions dictionary.

9.11 How to parameterise my strategy

On Spring, you can parameterise your strategy so that you can launch backtests or live executions with dynamic input at the time of launch (instead of hard-coding them in the strategy). This is done in two steps.

First make sure you have put all your parameters in a dictionary named *params* and have set it as an attribute of the *context* variable in the *initialize* function. Just after that, call the *set_algo_parameters* function to set the *params* dictionary as your parameters definition for the strategy. Use appropriate default values for your parameters while defining the dictionary.

```
from blueshift.api import set_algo_parameters

def initialize(context):
    # strategy parameters
    context.params = {'my_param1':0, 'my_param2':42}
    set_algo_parameters('params')
```

The *set_algo_parameters* API call binds the context attribute *params* as strategy parameters. You can use any other variable name as well, but *params* is recommended for easier tracking and understanding.

In the second step, define your parameters while creating the strategy on the plaform. Once both are done successfully, you will get options to select parameters while launching your strategy on the platform. Be careful to exactly match the name of your parameters while creating your strategy parameters (else the default values will be seen by the strategy).

Important: If you are defining your strategy that accepts parameters as above, it is highly recommended that you add validation in each of the parameter values before using them in the strategy. This is because the parameters entered during launch and passed by the platform to your strategy can potentially be corrupted (mis-format, bad inputs etc.)

9.12 Good practices to follow for strategy building

Algorithmic trading can be advantageous as machines are faster than humans. But they are faster when things go wrong as well. It is of utmost importance that we take proper steps to make our strategies fault-tolerant. Below is a (non-exhaustive) set of points to keep in mind before you take your strategy live.

- **Strategy is designed to be fault-tolerant for corrupt input data**
  At the base level, check if the data you have received is very much different from the last datapoints your strategy had. Most exchanges usually follow a market volatility control for publicly traded securities. These limit stock price movements and also enforce cool-off periods in such circumstances. If your algo received some extreme data points, it is highly likely they are wrong. Even if they are true, the market volatility control mechanism probably has already been triggered. If your strategy is not particularly designed to exploit these situations, it is a good practice to pause any trading activity till saner data arrive.
- **Strategy has necessary risk controls in place**
  This is, again, an absolute must. At the minimum, it should control the max number of orders it can send (to control machine gunning), the max size of each order (machines do fat fingers too) and a kill switch (a percentage loss below which it should stop automatically). Blueshift® has all these features, and then some more. You can put controls based on the maximum position size, maximum leverage or even declare a white-list (black-list) of assets that the algo can (cannot) trade.
- **Checking/ cancelling pending open order before placing new orders**
  This one is an absolute must for live trading. A best practice is usually to cancel all open orders before placing fresh orders, or, updating the existing orders as per the algo signals. Else it is easy to end up with a machine gun order scenario - the algo firing up and queuing up orders faster than they can be processed. See the code snippet below. We use the *get_open_orders* and *cancel_order* API functions to handle pending orders, **before** placing fresh orders.
- **Order placing and signal generations are isolated into specific functions**
  A strategy that places orders from multiple functions can mess up really fast. Make sure all your orders are placed only through a specific function. In such a design, chances of unforeseen mis-behaviours are considerably less.
- **Strategy is not over-sensitive to latency**
  Orders will be sent and processed over regular internet. So any latency sensitive strategies (like cross exchange arbitrage or market-making) are bound to suffer. Also internet connections are prone to interruptions or even complete outages. The strategy should be robust to such scenarios.

9.13 Things to avoid while writing a strategy

- **A strategy generating orders at a very high rate**

  These are more prone to instability, and also perhaps lose more in round trip trading costs than they make. A hair-trigger signal generation method may result in such a scenario. So make sure your signal generation is robust and expected holding periods are consistent with points above.

- **A strategy that triggers orders continuously for the same prevailing condition**

  If you are trading based on some technical indicators, say *RSI*, you usually want to place an order when the indicator crosses a threshold (a change of state). The intention is not to generate orders constantly as long as the indicator stays below (or over) that threshold (a state). If you are using a state-based signal, make sure the order functions are targeting in nature (e.g. the ubiquitous *order_target_percent*), not absolute (e.g. *order*). In case you are using absolute orders, make sure your signal generation is based on change of state, not the state itself

- **A strategy trading close to the account capacity**

  Margin calls can put an automated strategy out of gear. Always ensure the account is funded adequately, so that the algo runs in an expected way.

Risk management and monitoring makes all the difference between blowing out the bank roll and making handsome profit. The above points will take care of some aspects of risk management, especially in automated trading set-up. But bank-roll management, position sizing etc, are still some points that need to be deliberated carefully before taking a strategy live. Also it is absolutely necessary we keep a close watch on the algo performance.

CHAPTER TEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX