Cosine Documentation

Dotun Rominiyi

Oct 15, 2018

Contents

- GitHub repository
- GitHub example repository

Cosine is a crypto exchange trading algo framework. It provides a modular framework for implementing custom algorithmic trading strategies, across either single or multiple execution venues, with support for multiple concurrent pricing feeds from a variety of market pricing sources.

Cosine can support synchronous and asynchronous execution, and is designed with scalability in-mind via a multiprocess architecture.

CHAPTER 1

Contents

1.1 Quickstart

Note: All code starting with a `$` is meant to run on your terminal. All code starting with a >>> is meant to run in a python interpreter, like ipython.

1.1.1 Installation

Cosine can be installed (preferably in a *virtualenv*) using `pip` as follows:

```
$ pip install cosine-crypto
```

Note: If you run into problems during installation, you might have a broken environment. See the troubleshooting guide to *Set up a clean environment*.

Installation from source can be done from the root of the project with the following command.

```
$ pip install .
```

1.1.2 Using Cosine

Once cosine has been installed, it's relatively easy to get setup to run it. Before that however, it's best to get familiar with the architecture and more information can be found via the *Overview* section of this documentation.

Once you're ready you can begin by implementing code similar to the following.

```
import argparse
from cosine.core.algo import CosineAlgo


def main():
    parser = argparse.ArgumentParser(description='Cosine Algo')
    parser.add_argument("-c", "--config",
                        help="Config YAML file required to setup the algo.")
    parser.add_argument("-e", "--env", choices=['DEV', 'TST', 'PRD'],
                        help="The execution environment.")
    parser.add_argument("-a", "--appname", default='Cosine Algo',
                        help="The name of the application.")
    parser.add_argument("-lf", "--logfile",
                        help="The name of the log file. Do not change this unless you␣
↪know what you're doing.")
    parser.add_argument("-lv", "--loglevel",
                        choices=['DEBUG', 'INFO', 'WARNING', 'ERROR', 'FATAL', 'CRITICAL
↪'],
                        help="Log level.")

    parser.print_help()
    args = parser.parse_args()

    app = CosineAlgo(cmdline_args=args)
    app.run()


if __name__ == "__main__":
    main()
```

In the above code we perform the following actions:

1. First we import the argparse module to capture the required command line arguments Cosine cares about.
2. Then we import the `CosineAlgo` class, the main application class we need to instantiate to launch the algo.
3. In our main function we construct the argument list for consumption and parse it.
4. We then instantiate the `CosineAlgo` class into an object, which we then proceed to run to start the algo execution.

1.1.3 Command Line Arguments

Here's the set of command line arguments that cosine looks for.

Note: Cosine will by default check the provided `cmdline_args` keyword parameter of the `CosineAlgo` constructor, for the required parameters and if not provided, fall back to the environment variable equivalent and then a

default if none has been provided.

Command Line Argument	Environment Variable	Description	Type
–config	COSINE_CFG	The path to the configuration file required	str
–appname	COSINE_APP	The name of the application	str
–env	COSINE_ENV	The execution environment	str
–logfile	COSINE_LOGFILE	The path to the file to provide logging into	str
–loglevel	COSINE_LOGLVL	The filtered log level	str

1.1.4 Configuring The Algo

Cosine provides a comprehensive set of configurations for setting up and customising the algo. See the *Configuration Management* section for more details.

1.1.5 Custom Strategies

Cosine supports the implementation and configuration of custom strategies for use with the algo framework. See the *Strategies* section for more details.

1.1.6 Custom Connectivity

Cosine supports the implementation and configuration of connectivity to a multiple custom venues for execution, as well as to multiple pricing feeds concurrently. See the *Venues* section to learn more about building and using custom exchange connectivity. Also check out the *Pricing Feeds* section to learn more about building consuming custom feeds to new pricing sources.

1.2 Overview

1.2.1 Architecture

Cosine was designed with both scalability and modularity in mind. At The top level is the `CosineAlgo` a class which represents top level application construct. Within this are several constructs providing varying layers of functionality, of which all of that are provided to the core strategy (a derivative of `CosineBaseStrategy`) for orchestration of all trading activities. Here's the list of constructs provided:

- `CosineBaseFeed` - The base class representing an interface for subscribing to market pricing information, typically from an external data source. Current the `CryptoCompareSocketIOFeed` has been provided to support pricing feeds across a range of instruments.
- `CosinePairInstrument` - The class representing a single tradable instrument, representing an asset pair. Or more specifically, a tradable asset, denominated in a secondary quote currency asset type. Both assets derive from `CosineTradableAsset` base type.
- `CosineBaseVenue` - The base class representing an interface for connectivity to a market execution venue. It's expected that such venues would provide a range of markets for trading a set of instruments, as well as providing on-exchange account management of trader balances of inventory. This interface should expose all required access to this functionality and the interface is designed to support either synchronous (blocking) exchange interaction or asynchronous (non-blocking).
- `CosineBasePricer` - The base class representing an interface to an internal or external source of pricing. Cosine is designed to support sophisticated pricing engines that may consume a wide variety of pricing sources and leverage statistical pricing models to determine theoretical pricing for a given market, or provide pricing leans as a means of integrated risk management within execution of the algo.
- `CosineOrderWorker` - The class which provides order-working management (e.g. place resting orders, amend orders based on pricing changes and cancel orders) for a specified instrument, against a specified venue.

1.2.2 Event Loop

Cosine is designed to run via an internal event loop. The main process which runs the `CosineAlgo` instance, kicks off the set of configured venues and feeds (which may individually create separate processes) as well as creating the strategy. At which point the algo can run in one of two configurations: `system.EventLoop: feed` which uses the `feed.Primary:` configuration to determine which configured and initialised `CosineBaseFeed` to use to drive the event loop. Alternatively the `system.EventLoop: timer` configurations sets up a periodic timer which kicks the event loop on a regular interval.

```
# in CosineAlgo
def run(self):

    # setup the algo...
    self.setup()

    # initiate the update loop...
    if self._cfg.system.EventLoop == "feed":

        # here we tick the strategy every time the primary pricing feed ticks...
        self._run_on_primary_feed()
    else:
        # here we tick the strategy every time the timer ticks...
        self._run_on_timer()

    # clean up the algo and exit...
    return self.teardown()
```

If the `system.EventLoop: feed` is selected, whenever a pricing update comes in from the pricing feed, the pricing cache is updated (see the *Pricing Feeds* section for more details) and the `CosineAlgo` instance's `_tick_main()` method is called. This call is rate limited by the `debounce` utility decorator, based on the `system.EventLoopThrottle:` configuration value, which represents a minimal inter-arrival rate in seconds.

The `_tick_main()` represents the main workhouse of the algo, which processing the following business logic in-order:

- Update any configured auxiliary (i.e. non-primary) pricing feeds to build an up-to-date set of pricing caches (feeds subscribe to message updates which are queued and processed on the main process)
- Update all the configured venues (if any have asynchronous components, they will queue inbound events and these update calls will consume the queued events and update order worker states accordingly)
- Update the strategy (this will process any strategy level business logic, see the *Strategies* section for more details)

1.2.3 Multiprocessing

Cosine is designed to operate within a multi-process setup. The core `CosineAlgo` provides a framework class called `CosineProcWorkers`, which manages and orchestrates all processes for running the various services required by the algo framework, such as pricing, feeds, exchange connectivity and strategy logic. For example, the `BlockExMarketsVenue` (a derived class of `CosineBaseVenue`) provides two main connectivity protocols: a set of synchronous requests to control order execution and to query order, market, instrument and pricing information. As well as an asynchronous event feed for subscribing to pricing changes order state updates and executions, to publish events that are queued and picked up by the main process for reactive processing (i.e. order-working). In-order to achieve this, the `BlockExMarketsVenue` implements a `CosineProcWorker` derived class (`BlockExMarketsSignalRWorker`), which handles the blocking subscription for async updates in a separate process, and events that come in are republished into an event queue for processing on the main process by the initiating `CosineOrderWorker`.

The `CosineProcWorkers` instance can be leveraged by any custom strategy, feed or connectivity layer module, that requires processing to be done in a separate process, so as to not block the main process event loop. This provides streamlined execution of the algo, and was designed this way to lay the foundations for maximising performance (e.g. for HFT algo development), alleviating the costs around frequent context switching as well as the risks of thread/process starvation inherent with a cooperative multi-threading (aka green threads or co-routines - i.e. Asyncio concurrency) approach. This also alleviates the performance problems inherent in python multithreading due to the Global Interpreter Lock (or GIL).

1.2.4 Configuring The Algo

Cosine provides a comprehensive set of configurations for setting up and customising the algo. See the *Configuration Management* section for more details.

1.2.5 Custom Strategies

Cosine supports the implementation and configuration of custom strategies for use with the algo framework. See the *Strategies* section for more details.

1.2.6 Custom Connectivity

Cosine supports the implementation and configuration of connectivity to a multiple custom venues for execution, as well as to multiple pricing feeds concurrently. See the *Venues* section to learn more about building and using custom exchange connectivity. Also check out the *Pricing Feeds* section to learn more about building consuming custom feeds to new pricing sources.

1.3 Configuration Management

1.3.1 Configuration Loading

Cosine uses YAML-based configuration to configure the setup of the algo framework. This includes things like pricing feeds to initialise, the strategy to run, available venues to connect to and more. The configuration file must be provided upfront and you can inform Cosine how to locate it in one of three ways:

- Direct (Cosine will look for a config file path in the `config` attribute of the `cmdline_args` dict argument passed to the `CosineAlgo` constructor)
- Environment (Cosine with otherwise check for the path in the `COSINE_CFG` environment variable if it exists)
- Implicit (Cosine will otherwise look for a file called `config.yaml` in the current working directory of the executed algo)

1.3.2 Configurable Attributes

Here's the full set of available configurations (complete with group headings per the YAML structure):

```
# all system-level configurations
system:
    EventLoop: <val>                                # the event loop configuration mode,
↪"feed" or "timer"
    EventLoopThrottle: <val>                        # event loop rate limit in seconds
    network:                                        # general network level configuration
        ssl:                                        # SSL related configuration
            CertFile: <val>                         # [optional] path to the SSL␣
↪certificate authority cert file

# general order worker related configurations
orders:
    ActiveDepth: <val>                              # active depth on each side of book␣
↪respectively (bid and ask)

# set of configured venues (with their contextual configurations) to initialise for␣
↪use with the order workers
venues:
    cosine.venues.bem:                              # [optional] the fully qualified␣
↪module path of the BlockExMarketsVenue (CosineBaseVenue derivative) class to load +␣
↪configure
        Username: <val>                             # [venue-specific] the username of␣
↪the trader account to authenticate against
        Password: <val>                             # [venue-specific] the password of␣
↪the trader account to authenticate against
        APIDomain: <val>                            # [venue-specific] the top-level␣
↪domain of the BEM venue
        APIID: <val>                                # [venue-specific] the dedicated␣
↪APIID for the BEM venue
        ConnectSignalR: <val>                       # [venue-specific] tells BEM whether␣
↪to subscribe to the async signalR feed or not, "true" or "false"

# the set of configured instruments to work markets in. Order workers will be created␣
↪against each of these on the relevant venue(s)
```

(continues on next page)

(continued from previous page)

```
instruments:
- "XTN/EUR"
- "RCC/EUR"
- "ETH4/EUR"

# the set of configured pricing feeds to connect and subscribe to for market data␣
↪consumption
feeds:
    cosine.pricing.cryptocompare:                # [optional] the fully qualified␣
↪module path of the CryptoCompareSocketIOFeed (CosineBaseFeed derivative) class to␣
↪load + configure
        type: <val>                              # [feed-specific] the type of␣
↪connection ("stream" only for this feed)
        endpoint: <val>                          # [feed-specific] the websockets/
↪socket.io endpoint hostname to connect to
        port: <val>                              # [feed-specific] the port to␣
↪connect to
        framework: <val>                         # [feed-specific] the framework for␣
↪connectivity
        triangulator: <val>                      # [feed-specific] the REST endpoint␣
↪to use to pull triangulation info for implying pricing for pairs with no direct␣
↪subscription
        triangulator_throttle: <val>             # [feed-specific] the rate limit for␣
↪running triangulation queries in seconds
        instruments:                             # the set of instruments to␣
↪subscribe to
            "XTN/EUR":
                Ticker: <val>                    # ticker re-mapping for the base/top-
↪level currency, e.g. "BTC"
                BaseCCY: <val>                   # [optional] forces the feed to e.g.␣
↪if the value is "ETH" for an RCC/EUR pair, subscribe to RCC/ETH and then run␣
↪triangulation on each price tick to calculate the RCC/EUR price
            "RCC/EUR": {}
            "ETH4/EUR": {}

# [optional] the configured primary feed, such that when "system.EventLoop: feed",␣
↪this CosineBaseFeed derivative will be configured to drive the main event loop
feed:
    Primary: cosine.pricing.cryptocompare        # primary feed to drive the main␣
↪event loop

# the set of configured pricers to pipeline for processing pricing data. Can be used␣
↪to consume raw price feed data and generate theoretical pricing or other price-
↪derived values
pricers:
    Default: cosine.pricing.pricers.nullpricer   # a comma-separated list of pricer␣
↪modules to load and pipeline in-order for pricing generation
    settings:                                    # the set of pricer-specific␣
↪configurations
        cosine.pricing.pricers.nullpricer: {}    # [pricer-specific] pricer␣
↪configuration

# the configuration for the configured strategy to run
strategy:
    type: cosine.strategies.noddy_floater        # the strategy module to load and␣
↪run under the algo. This contains the core business logic of the algo
    settings:                                    # the set of strategy-specific␣
↪settings configurations
```

(continues on next page)

(continued from previous page)

```
        cosine.strategies.noddy_floater:          # [optional] the noddy_floater␣
↪strategy settings
            Spread: <val>                         # [strategy-specific] the % spread␣
↪to maintain around the spot mid-price, e.g. 0.20
            MaxSpread: <val>                      # [strategy-specific] the maximum %␣
↪spread based on dynamic widening of quotes, e.g. 0.50
            instrument_settings:                  # [strategy-specific] instrument␣
↪specific strategy settings
                "XTN/EUR":
                    MinVol: <val>                 # [strategy-specific] minimum volume␣
↪per quoted price step
                    MaxVol: <val>                 # [strategy-specific] maximum volume␣
↪per quoted price step
                "RCC/EUR": {}
                "ETH4/EUR": {}
```

See above.

1.4 Pricing Feeds

1.4.1 Structure

Cosine's pricing feeds all inherit from the `CosineBaseFeed` class and are responsible for providing all market data consumption connectivty for the algo framework. Feeds are designed to be modular, where an algo can be configured to connect to multiple feed sources simultaneously to obtain pricing for different instruments, or even for the same ones but providing different data sets perhaps. Feeds are setup via the configuration file (See the *Configuration Management* section for more details) with all of their custom feed-specific attributes defined. These config attributes will be automatically set as member attributes on the class instance on construction via the `CosineAlgo`. Once a feed instance has been created, the `CosineAlgo` calls the feed's `setup()` method, which first initialises the pricing cache based on the set of configured instruments, and then it will attempt to initialise the feed connection. If the `system.EventLoop: feed` configuration is set and the specific feed instance is the primary feed, then the feed will run inline in the current process. If not then the feed will run in a separate worker process managed by the `CosineProcWorkers` worker pool.

The `CryptoCompareSocketIOFeed` has been provided as an initial pricing feed implementation, which supports websockets based pricing updates, with REST based pricing triangulation on pricing ticks, for price triangulation of instrument pairs where required (i.e. if there is no direct pricing feed provided for this pair by cryptocompare.com). Feel free to check out the code for this feed implementation to familiarise yourself with the structure.

1.4.2 Developing A Custom Feed

Custom feeds can be implemented easily, by inheriting from `CosineBaseFeed` and overriding the `run()` method. The `CryptoCompareSocketIOFeed` provides an example implementation of how to achieve this.

Once your custom feed has been written and tested, you can leverage it in one of two ways:

- Submit your feed implementation to the main open source repository as a pull request (recommended if applicable)
- Or leverage it locally as part of your algo implementation.

To achieve the latter is fairly simple. Assuming you have an algo project setup to use cosine (an example project has been provided here), you can simply configure the module path in the configuration file to allow the native import mechanics to load your custom module locally at run-time.

For example, let's assume we have a custom feed called `MyCustomFeed` in the local project (at the path `/myproject/pricing/mycustomfeed.py`). We can setup the following in the configuration file (at `/myproject/config.yaml`):

```
...

feeds:
    pricing.mycustomfeed:
        type: stream
        endpoint: wss://mycustomfeed.io
        port: 443
        framework: websockets
        triangulator: https://api.mycustomfeed.io/priceinfo
        triangulator_throttle: 0.5
        instruments:
            "XTN/EUR":
                Ticker: "BTC"
            "RCC/EUR":
                BaseCCY: "ETH"
            "ETH4/EUR":
                Ticker: "ETH"
...
```

Which should inform `CosineAlgo` to load the module, instantiate the feed and initialise it for use.

1.5 Venues

1.5.1 Structure

Cosine's venues all inherit from the `CosineBaseVenue` class and are responsible for providing all market exchange connectivity, on-venue portfolio management and trade execution functionality for the algo framework. Venues are designed to be modular, where an algo can be configured to connect to multiple venues simultaneously, and can work different sets of instruments across different venues as part of a comprehensive, complex trading strategy. Venues are setup via the configuration file (See the *Configuration Management* section for more details) with all of their custom venue-specific attributes defined. These config attributes will be automatically set as member attributes on the class instance on construction via the `CosineAlgo`. Once a venue instance has been created, the `CosineAlgo` calls the venue's `setup()` method, which is expected to initialise any connectivity to the remote exchange as well as retrieve and cache any supported (or at least required) tradable assets/instruments at the destination exchange. The `CosineBaseVenue` base class provides access to the `CosineProcWorkers` worker pool instance, that can be using during setup to instantiate a worker for any asynchronous exchange connectivity required (if at all).

The BlockExMarketsVenue has been provided as an initial exchange venue integration, which supports trade execution via REST (with asynchronous response updates), on-venue portfolio management & symbology retrieval. Feel free to check out the code for this venue implementation to familiarise yourself with the structure.

1.5.2 Developing A Custom Venue

Custom venue implementations can be integrated easily, by inheriting from CosineBaseVenue and overriding the following methods:

- setup()
- teardown()
- update() (optional)
- on() (optional)
- get_instrument_defs
- get_open_orders
- get_inventory
- new_order
- cancel_order
- cancel_all_orders

Custom venues can implement support for these methods either synchronously or asynchronously depending on the anatomy of the remote exchange connectivity API the venue implementation aims to abstract. The venue must inform the cosine framework of which configuration it implements via overriding the is_async() property of the CosineBaseVenue class.

To implement asynchronous connectivity, the venue should define a process worker, which it initialises internally and runs as part of setup() workflow. The CosineProcEventWorker class provides a convenient worker implementation to inherit from, which handles IPC via a shared event queue. The main process initiates requests directly on the main process and CosineProcEventWorker.EventSlot listeners are used to capture and handle responses asynchronously. The worker should connect to the exchange via some asynchronous connectivity protocol and subscribe for the request responses. When the responses arrive they are pushed onto shared queue. The main process will then leverage the update() method to drive event processing, which pulls queued events and handles them via the handlers registered with each relevant CosineProcEventWorker.EventSlot.

If an exchange connectivity implementation requires full asynchronous processing, i.e. requests need to be sent asynchronously, then it maybe best to implement your own CosineProcWorker derivative, with your own CosineProcEventMgr equivalent implementation to better facilitate the specific workflow required.

The BlockExMarketsVenue provides an example implementation of how to implement a CosineBaseVenue, complete with asynchronous workflow, via the BlockExMarketsSignalRWorker implementation.

Once your custom venue has been written and tested, you can leverage it in one of two ways:

- Submit your venue implementation to the main open source repository as a pull request (recommended if applicable)
- Or leverage it locally as part of your algo implementation.

To achieve the latter is fairly simple. Assuming you have an algo project setup to use cosine (an example project has been provided here), you can simply configure the module path in the configuration file to allow the native import mechanics to load your custom module locally at run-time.

For example, let's assume we have a custom venue called `MyCustomVenue` in the local project (at the path `/myproject/venues/mycustomvenue.py`). We can setup the following in the configuration file (at `/myproject/config.yaml`):

```
...

venues:
    venues.mycustomvenue:
        Username: trader101324
        Password: !example123456!
        APIDomain: https://api.exchange-venue.io
        ... <venue specific configs> ...
...
```

Which should inform `CosineAlgo` to load the module, instantiate the venue and initialise it for use.

1.6 Pricers

1.6.1 Structure

Cosine's pricers all inherit from the `CosinePricer` class and are responsible for providing auxiliary pricing generation which maybe useful or required depending on the strategy implementation at play. Pricers are more of an optional component within the algo framework, as not every strategy will need one. Pricers are designed to be pipelined, such that algos can leverage a chain of them to generate supplementary pricing based on the base pricing retrieved from the feeds. Pricer may also be used to source external pricing information which can be pulled into the algo to generate pricing related information (e.g. generated information which takes the feed pricing + externally sourced data as inputs and outputs information used by the strategy to generate quotes or order working) required for the specific strategy.

1.6.2 An Example Use Case

A typical example use-case could be as follows:

An algo strategy implementation wishes to quote orders in relation to two specific sources of generated pricing information:

- Volume Weighted Average Pricing (VWAP)
- Price Volatility

The strategy implementation aims to construct a dynamic quoting behaviour by performing the following actions on each tick:

- Source aggregate market pricing for each instrument from the relevant feed source
- Update the VWAP, feeding the latest feed pricing snapshot as input
- Source the latest Volatility metrics to generate a dynamic quote width (i.e. widen quotes as the volatility increases, within a specified ratio)
- Construct a set of quotes centered around the VWAP

- Apply the generated leans to dynamically widen/narrow the quotes in-response to current volatility
- Apply the new quotes via the order workers

This strategy can be achieved by leveraging a set of pricers to define the structural building blocks for managing the pricing info generation for use within the strategy.

An implementation here may consist of two pricers:

- `VWAPPricer` where it's `generate_theo_prices()` method override takes the feed snapshot prices as input and calculates the VWAP, setting it into the snapshot and returning it
- `VolPricer` which takes the feed snapshot prices as input, makes a synchronous internal call to a volatility data source (REST/gRPC/etc), calculates the dynamic quote width info and writes that into the snapshot before returning it

With strategy logic to drive their use similar to the following example code:

```
# in the class CustomStrategy(CosineBaseStrategy)
def update(self):
    self.logger.debug("CustomStrategy - ** update **")

    ...

    # pull prices for instruments...
    self.logger.debug("CustomStrategy - source instrument prices from feed cache...")
    feed = self._cxt.feeds['pricing.mycustomfeed']
    prices = feed.capture_latest_prices(instruments=instruments)

    # massage pricing...
    for p in self._cxt.pricer_seq:
        self.logger.debug(f"CustomStrategy - calc pricing: [{p}]")
        prices = self._cxt.pricers[p].generate_theo_prices(instrument_prices=prices)

    ...
```

The pricer configuration in this example would be setup as follows:

```
...

pricers:
    Default: pricing.pricers.vwappricer,pricing.pricers.volpricer # the pipeline␣
↪ordering of pricers...
    settings:
        pricing.pricers.vwappricer:
            ... <pricer-specific config> ...
        pricing.pricers.volpricer:
            ... <pricer-specific config> ...
...
```

Pricers are setup via the configuration file (See the *Configuration Management* section for more details) with all of their custom pricer-specific attributes defined. These config attributes will be automatically set as member attributes on the class instance on construction via the `CosineAlgo`. Once a pricer instance has been created, the `CosineAlgo` calls the pricer's `setup()` method, which is expected to initialise any external data sourcing or internal setup requirements. The `CosinePricer` base class provides access to the `CosineProcWorkers` worker pool instance, that can be using during setup to instantiate a worker for any asynchronous data source connectivity required (if applicable).

The `NullPricer` has been provided as an initial pricer skeleton, mainly to demonstrate how it fits into the wider framework and how it can be used via pipelining within strategy implementations. Feel free to check out the code for this pricer implementation to familiarise yourself with the structure.

1.7 Strategies

1.7.1 Structure

Cosine's strategies represent the core business logic drivers for each algo and leverage the rest of the framework components (venues, orderworkers, feeds, pricers) to achieve their goals. Inheriting from the `CosineBaseStrategy` class, strategies are designed to run the main business logic on an event loop. This can be configured to run on every tick (or throttled) of a specific pricing feed or on a simple timer. Strategies are setup via the configuration file (See the *Configuration Management* section for more details) with all of their custom strategy-specific attributes defined. These config attributes will be automatically set as member attributes on the class instance on construction via the `CosineAlgo`. Once a strategy instance has been created, the `CosineAlgo` calls the strategy's `setup()` method, which is expected to initialise any logical setup required for operation. The `CosineBaseStrategy` base class provides access to all configured feeds, orderworkers and pricers to use as part of the strategy logical flow, mostly via the `CosineCoreContext` passed to it on construction.

The `NoddyFloaterStrategy` has been provided as an initial strategy implementation (limited), which supports simple price consumption, pricer pipelining as well as quote updating based on the generated target prices. See the *Strategy: Noddy Floater* section for more details.

1.7.2 Running Multiple Strategies

The `CosineAlgo` is designed to run a single strategy instance within the main algo process. It's possible to run multiple strategies within a single algo process, however care must be taken to measure and assess the performance considerations of this, specifically where a lot of heavy lifting is done. Since all strategies would run their `update()` methods serially on each tick of the main event loop, significant latency in update processing may cause undesirable behaviour in the algo. For simple strategies, a `CosineMultiStrategy` class has been provided to allow multiple strategies to be run under it.

To configure the `CosineMultiStrategy` you can setup the `config.yaml` as follows:

```
...
strategy:
    type: cosine.strategies.multi_strategy
    settings:
        cosine.strategies.multi_strategy:
            strategies:
            - strategies.localstrategya
            - strategies.localstrategyb
        strategies.localstrategya:
            ... <strategy specific configs> ...
        strategies.localstrategyb:
            ... <strategy specific configs> ...
...
```

1.7.3 Developing A Custom Strategy

Custom strategy implementations can be integrated easily, by inheriting from `CosineBaseStrategy` and overriding the following methods:

- `setup()` (optional)
- `teardown()` (optional)
- `update()`

A suite of convenience methods have also been provided in the `CosineBaseStrategy` class to streamline strategy logic as needed. Strategies have direct access to all orderworkers, which in-turn are configured and grouped by connected venue. In this way a single strategy can generate quotes ansd work orders across either a single market or across multiple markets simultaneously as required by the business logic implemented within it. In a similar vein, strategies have full access to all configured feeds and can source instrument pricing from one or more feeds as desired.

Once your custom strategy has been written and tested, you can leverage it in one of two ways:

- Submit your strategy implementation to the main open source repository as a pull request (not recommended unless widely useful)
- Or leverage it locally as part of your algo implementation. (recommended)

To achieve the latter is fairly simple. Assuming you have an algo project setup to use cosine (an example project has been provided here), you can simply configure the module path in the configuration file to allow the native import mechanics to load your custom module locally at run-time.

For example, let's assume we have a custom strategy called `MyCustomStrategy` in the local project (at the path `/myproject/strategies/mycustomstrategy.py`). We can setup the following in the configuration file (at `/myproject/config.yaml`):

```
...
strategy:
    type: strategies.mycustomstrategy
    settings:
        strategies.mycustomstrategy:
            ... <strategy specific configs> ...
...
```

Which should inform `CosineAlgo` to load the module, instantiate the strategy and initialise it for use.

1.8 Strategy: Noddy Floater

1.8.1 Structure

Cosine's `NoddyFloaterStrategy` strategy provides a simple limited-functionality example use-case for how to structure and write your own custom strategies in the cosine algo framework. See the *Strategies* section for more details in general around strategies, how they work, how they're shaped and how then can be developed and integrated.

You can also checkout the example algo project here to see how to configure and run the `NoddyFloaterStrategy`.

1.8.2 Core Logic

Noddy floater's core logic is fairly simple and follows the following algorithmic flow on every price tick:

- Retrieve the set of orderworkers for the target venue (BEM)
- Extract the set of instruments associated with each order worker
- Capture the set of cached pricing snapshots for each instrument
- Run the pricer pipeline across all pricing snapshots (to embellish them)
- Generate a set of quotes from the final pricing data
- Push the quotes to the orderworkers to update all order quotes on the markets

Quoting is basically constructed as a butterfly spread around the mid-price for each market, roughly 20% (configurable) away from the mid-price on either side. Quotes are configured to linearly scale up to a maximum spread price with some marginal stochastic variance on each price step. Volume is similarly calculated out based on a linear interpolation between minimum & maximum quote sizes per price level.

This is a very simple implementation that does nothing fancy but proves the basic building blocks for developers to build more sophisticated trading strategies on the cosine framework.

1.9 Troubleshooting

1.9.1 Set up a clean environment

Many things can cause a broken environment. You might be on an unsupported version of Python. Another package might be installed that has a name or version conflict. Often, the best way to guarantee a correct environment is with `virtualenv`, like:

```
# Install pip if it is not available:
$ which pip || curl https://bootstrap.pypa.io/get-pip.py | python

# Install virtualenv if it is not available:
$ which virtualenv || pip install --upgrade virtualenv

# *If* the above command displays an error, you can try installing as root:
$ sudo pip install virtualenv

# Create a virtual environment:
$ virtualenv -p python3 ~/.venv-py3

# Activate your new virtual environment:
$ source ~/.venv-py3/bin/activate

# With virtualenv active, make sure you have the latest packaging tools
$ pip install --upgrade pip setuptools

# Now we can install cosine...
$ pip install --upgrade cosine-crypto
```

Note: Remember that each new terminal session requires you to reactivate your virtualenv, like: `$ source ~/.venv-py3/bin/activate`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search