tensortrade Documentation

Release 1.0.4-dev1

Adam King

Aug 23, 2022

Contents

TensorTrade is an open source Python framework for building, training, evaluating, and deploying robust trading algorithms using reinforcement learning. The framework focuses on being highly composable and extensible, to allow the system to scale from simple trading strategies on a single CPU, to complex investment strategies run on a distribution of HPC machines.

Under the hood, the framework uses many of the APIs from existing machine learning libraries to maintain high quality data pipelines and learning models. One of the main goals of TensorTrade is to enable fast experimentation with algorithmic trading strategies, by leveraging the existing tools and pipelines provided by `numpy`, `pandas`, `gym`, `keras`, and `tensorflow`.

Every piece of the framework is split up into re-usable components, allowing you to take advantage of the general use components built by the community, while keeping your proprietary features private. The aim is to simplify the process of testing and deploying robust trading agents using deep reinforcement learning, to allow you and I to focus on creating profitable strategies.

The goal of this framework is to enable fast experimentation, while maintaining production-quality data pipelines.

Feel free to also walk through the Medium tutorial.

The most up to date example is Train and Evaluate using Ray. We suggest you to start there!

CHAPTER 1

Guiding principles

Inspired by Keras' guiding principles.

User friendliness. TensorTrade is an API designed for human beings, not machines. It puts user experience front and center. TensorTrade follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

Modularity. A trading environment is a conglomeration of fully configurable modules that can be plugged together with as few restrictions as possible. In particular, exchanges, feature pipelines, action schemes, reward schemes, trading agents, and performance reports are all standalone modules that you can combine to create new trading environments.

Easy extensibility. New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making TensorTrade suitable for advanced research and production use.

1.1 Getting Started

You can get started testing on Google Colab or your local machine, by viewing our many examples

1.2 Installation

TensorTrade requires Python >= 3.7 for all functionality to work as expected.

You can install the package from PyPi via pip or from the Github repo.

```
pip install tensortrade
```

OR

```
pip install git+https://github.com/tensortrade-org/tensortrade.git
```

1.3 Docker

To run the commands below ensure Docker is installed. Visit https://docs.docker.com/install/ for more information

1.3.1 Run Jupyter Notebooks

To run a jupyter notebook execute the following

```
make run-notebook
```

which will generate a link of the form 127.0.0.1:8888/?token=... Paste this link into your browsers and select the notebook you'd like to explore

1.3.2 Build Documentation

To build documentation execute the following

```
make run-docs
```

1.3.3 Run Test Suite

To run the test suite execute the following

```
make run-tests
```

1.4 Code Structure

The TensorTrade library is modular. The `tensortrade` library usually has a common setup when using components. If you wish to make a particular class a component all you need to do is subclass `Component`.

```
class Example(Component):
    """An example component to show how to subclass."""

    def foo(self, arg1, arg2) -> str:
        """A method to return a string."""
        raise NotImplementedError()

    def bar(self, arg1, arg2, **kwargs) -> int:
        """A method to return an integer."""
```

From this abstract base class, more concrete and custom subclasses can be made that provide the implementation of these methods.

Example of Structure A good example of this structure is the `RewardScheme` component. This component controls the reward mechanism of a `TradingEnv`.

The beginning of the code in RewardScheme is seen here.

```
from abc import abstractmethod

from tensortrade.core.component import Component
from tensortrade.core.base import TimeIndexed


class RewardScheme(Component, TimeIndexed):
    """A component to compute the reward at each step of an episode."""

    registered_name = "rewards"

    @abstractmethod
    def reward(self, env: 'TradingEnv') -> float:
        """Computes the reward for the current step of an episode.

        Parameters
        ----------
        env : `TradingEnv`
            The trading environment

        Returns
        -------
        float
            The computed reward.
        """
        raise NotImplementedError()

    def reset(self) -> None:
        """Resets the reward scheme."""
        pass
```

As you can see above, the RewardScheme has a majority of the structural and mechanical details that guide all other representations of that type of class. When creating a new reward scheme, one needs to add further details for how information from then environment gets converted into a reward.

1.5 Setup Environment

```
import ta

import pandas as pd
import tensortrade.env.default as default

from tensortrade.data.cdd import CryptoDataDownload
from tensortrade.feed.core import Stream, DataFeed, NameSpace
from tensortrade.oms.instruments import USD, BTC, ETH, LTC
from tensortrade.oms.wallets import Wallet, Portfolio
from tensortrade.oms.exchanges import Exchange
from tensortrade.oms.services.execution.simulated import execute_order
```

Fetch Historical Data

```
cdd = CryptoDataDownload()

bitfinex_data = pd.concat([
    cdd.fetch("Bitfinex", "USD", "BTC", "1h").add_prefix("BTC:"),
```

(continues on next page)

(continued from previous page)

```
    cdd.fetch("Bitfinex", "USD", "ETH", "1h").add_prefix("ETH:")
], axis=1)

bitstamp_data = pd.concat([
    cdd.fetch("Bitstamp", "USD", "BTC", "1h").add_prefix("BTC:"),
    cdd.fetch("Bitstamp", "USD", "LTC", "1h").add_prefix("LTC:")
], axis=1)
```

Define Exchanges

An exchange needs a name, an execution service, and streams of price data in order to function properly.

The setups supported right now are the simulated execution service using simulated or stochastic data. More execution services will be made available in the future, as well as price streams so that live data and execution can be supported.

```
bitfinex = Exchange("bitfinex", service=execute_order)(
    Stream.source(list(bitfinex_data['BTC:close']), dtype="float").rename("USD-BTC"),
    Stream.source(list(bitfinex_data['ETH:close']), dtype="float").rename("USD-ETH")
)

bitstamp = Exchange("bitstamp", service=execute_order)(
    Stream.source(list(bitstamp_data['BTC:close']), dtype="float").rename("USD-BTC"),
    Stream.source(list(bitstamp_data['LTC:close']), dtype="float").rename("USD-LTC")
)
```

Now that the exchanges have been defined we can define our features that we would like to include, excluding the prices we have provided for the exchanges.

Define External Data Feed

Here we will define the feed to use whatever data you would like. From financial indicators to alternative datasets, they will all have to be defined and incorporated into the `DataFeed` provided to the environment.

```
# Add all features for bitstamp BTC & ETH
bitfinex_btc = bitfinex_data.loc[:, [name.startswith("BTC") for name in bitfinex_data.
↪columns]]
bitfinex_eth = bitfinex_data.loc[:, [name.startswith("ETH") for name in bitfinex_data.
↪columns]]

ta.add_all_ta_features(
    bitfinex_btc,
    colprefix="BTC:",
    **{k: "BTC:" + k for k in ['open', 'high', 'low', 'close', 'volume']}
)


with NameSpace("bitfinex"):
    bitfinex_streams = [
        Stream.source(list(bitfinex_btc[c]), dtype="float").rename(c) for c in␣
↪bitfinex_btc.columns
    ]
    bitfinex_streams += [
        Stream.source(list(bitfinex_eth[c]), dtype="float").rename(c) for c in␣
↪bitfinex_eth.columns
    ]


# Add all features for bitstamp BTC & LTC
```

(continues on next page)

(continued from previous page)

```
bitstamp_btc = bitstamp_data.loc[:, [name.startswith("BTC") for name in bitstamp_data.
↪columns]]
bitstamp_ltc = bitstamp_data.loc[:, [name.startswith("LTC") for name in bitstamp_data.
↪columns]]

ta.add_all_ta_features(
    bitstamp_ltc,
    colprefix="LTC:",
    **{k: "LTC:" + k for k in ['open', 'high', 'low', 'close', 'volume']}
)

with NameSpace("bitstamp"):
    bitstamp_streams = [
        Stream.source(list(bitstamp_btc[c]), dtype="float").rename(c) for c in␣
↪bitstamp_btc.columns
    ]
    bitstamp_streams += [
        Stream.source(list(bitstamp_ltc[c]), dtype="float").rename(c) for c in␣
↪bitstamp_ltc.columns
    ]


feed = DataFeed(bitfinex_streams + bitstamp_streams)
```

```
feed.next()
```

```
{'bitfinex:/BTC:date': Timestamp('2017-07-01 11:00:00'),
 'bitfinex:/BTC:open': 2505.56,
 'bitfinex:/BTC:high': 2513.38,
 'bitfinex:/BTC:low': 2495.12,
 'bitfinex:/BTC:close': 2509.17,
 'bitfinex:/BTC:volume': 287000.32,
 'bitfinex:/BTC:volume_adi': 462887.3781183644,
 'bitfinex:/BTC:volume_obv': nan,
 'bitfinex:/BTC:volume_cmf': 0.5388828039430464,
 'bitfinex:/BTC:volume_fi': nan,
 'bitfinex:/BTC:volume_em': nan,
 'bitfinex:/BTC:volume_vpt': -190920.02711825827,
 'bitfinex:/BTC:volume_nvi': 1000.0,
 'bitfinex:/BTC:volatility_atr': 85.51648155760596,
 'bitfinex:/BTC:volatility_bbh': 2509.17,
 'bitfinex:/BTC:volatility_bbl': 2509.17,
 'bitfinex:/BTC:volatility_bbm': 2509.17,
 'bitfinex:/BTC:volatility_bbhi': 0.0,
 'bitfinex:/BTC:volatility_bbli': 0.0,
 'bitfinex:/BTC:volatility_kcc': 2505.89,
 'bitfinex:/BTC:volatility_kch': 2524.15,
 'bitfinex:/BTC:volatility_kcl': 2487.6299999999997,
 'bitfinex:/BTC:volatility_kchi': 0.0,
 'bitfinex:/BTC:volatility_kcli': 0.0,
 'bitfinex:/BTC:volatility_dch': 2509.17,
 'bitfinex:/BTC:volatility_dcl': 2509.17,
 'bitfinex:/BTC:volatility_dchi': 0.0,
 'bitfinex:/BTC:volatility_dcli': 0.0,
 'bitfinex:/BTC:trend_macd': nan,
 'bitfinex:/BTC:trend_macd_signal': nan,
```

(continues on next page)

(continued from previous page)

```
'bitfinex:/BTC:trend_macd_diff': nan,
'bitfinex:/BTC:trend_ema_fast': nan,
'bitfinex:/BTC:trend_ema_slow': nan,
'bitfinex:/BTC:trend_adx': 0.0,
'bitfinex:/BTC:trend_adx_pos': 0.0,
'bitfinex:/BTC:trend_adx_neg': 0.0,
'bitfinex:/BTC:trend_vortex_ind_pos': nan,
'bitfinex:/BTC:trend_vortex_ind_neg': nan,
'bitfinex:/BTC:trend_vortex_diff': nan,
'bitfinex:/BTC:trend_trix': nan,
'bitfinex:/BTC:trend_mass_index': 0.0,
'bitfinex:/BTC:trend_cci': nan,
'bitfinex:/BTC:trend_dpo': 4963.073762705523,
'bitfinex:/BTC:trend_kst': -664.2012654186367,
'bitfinex:/BTC:trend_kst_sig': -664.2012654186367,
'bitfinex:/BTC:trend_kst_diff': 0.0,
'bitfinex:/BTC:trend_ichimoku_a': 2504.25,
'bitfinex:/BTC:trend_ichimoku_b': 2504.25,
'bitfinex:/BTC:trend_visual_ichimoku_a': 7460.129960014917,
'bitfinex:/BTC:trend_visual_ichimoku_b': 7449.72498449202,
'bitfinex:/BTC:trend_aroon_up': 4.0,
'bitfinex:/BTC:trend_aroon_down': 4.0,
'bitfinex:/BTC:trend_aroon_ind': 0.0,
'bitfinex:/BTC:momentum_rsi': nan,
'bitfinex:/BTC:momentum_mfi': nan,
'bitfinex:/BTC:momentum_tsi': -100.0,
'bitfinex:/BTC:momentum_uo': 0.2822915537138003,
'bitfinex:/BTC:momentum_stoch': 76.94414019715232,
'bitfinex:/BTC:momentum_stoch_signal': 76.94414019715232,
'bitfinex:/BTC:momentum_wr': -23.055859802847678,
'bitfinex:/BTC:momentum_ao': 0.0,
'bitfinex:/BTC:momentum_kama': nan,
'bitfinex:/BTC:others_dr': -66.42012654186367,
'bitfinex:/BTC:others_dlr': nan,
'bitfinex:/BTC:others_cr': 0.0,
'bitfinex:/ETH:date': Timestamp('2017-07-01 11:00:00'),
'bitfinex:/ETH:open': 279.98,
'bitfinex:/ETH:high': 279.99,
'bitfinex:/ETH:low': 272.1,
'bitfinex:/ETH:close': 275.01,
'bitfinex:/ETH:volume': 679358.87,
'bitstamp:/BTC:date': Timestamp('2017-07-01 11:00:00'),
'bitstamp:/BTC:open': 2506.5,
'bitstamp:/BTC:high': 2510.62,
'bitstamp:/BTC:low': 2495.5,
'bitstamp:/BTC:close': 2500.0,
'bitstamp:/BTC:volume': 521903.7,
'bitstamp:/LTC:date': Timestamp('2017-07-01 11:00:00'),
'bitstamp:/LTC:open': 39.67,
'bitstamp:/LTC:high': 39.67,
'bitstamp:/LTC:low': 39.32,
'bitstamp:/LTC:close': 39.45,
'bitstamp:/LTC:volume': 1957.48,
'bitstamp:/LTC:volume_adi': 5133.608444728016,
'bitstamp:/LTC:volume_obv': nan,
'bitstamp:/LTC:volume_cmf': -0.2571428571428455,
'bitstamp:/LTC:volume_fi': nan,
```

(continues on next page)

(continued from previous page)

```
 'bitstamp:/LTC:volume_em': nan,
 'bitstamp:/LTC:volume_vpt': -895.4321955984681,
 'bitstamp:/LTC:volume_nvi': 1000.0,
 'bitstamp:/LTC:volatility_atr': 1.4838450097847482,
 'bitstamp:/LTC:volatility_bbh': 39.45,
 'bitstamp:/LTC:volatility_bbl': 39.45,
 'bitstamp:/LTC:volatility_bbm': 39.45,
 'bitstamp:/LTC:volatility_bbhi': 0.0,
 'bitstamp:/LTC:volatility_bbli': 0.0,
 'bitstamp:/LTC:volatility_kcc': 39.480000000000004,
 'bitstamp:/LTC:volatility_kch': 39.830000000000005,
 'bitstamp:/LTC:volatility_kcl': 39.13,
 'bitstamp:/LTC:volatility_kchi': 0.0,
 'bitstamp:/LTC:volatility_kcli': 0.0,
 'bitstamp:/LTC:volatility_dch': 39.45,
 'bitstamp:/LTC:volatility_dcl': 39.45,
 'bitstamp:/LTC:volatility_dchi': 0.0,
 'bitstamp:/LTC:volatility_dcli': 0.0,
 'bitstamp:/LTC:trend_macd': nan,
 'bitstamp:/LTC:trend_macd_signal': nan,
 'bitstamp:/LTC:trend_macd_diff': nan,
 'bitstamp:/LTC:trend_ema_fast': nan,
 'bitstamp:/LTC:trend_ema_slow': nan,
 'bitstamp:/LTC:trend_adx': 0.0,
 'bitstamp:/LTC:trend_adx_pos': 0.0,
 'bitstamp:/LTC:trend_adx_neg': 0.0,
 'bitstamp:/LTC:trend_vortex_ind_pos': nan,
 'bitstamp:/LTC:trend_vortex_ind_neg': nan,
 'bitstamp:/LTC:trend_vortex_diff': nan,
 'bitstamp:/LTC:trend_trix': nan,
 'bitstamp:/LTC:trend_mass_index': 0.0,
 'bitstamp:/LTC:trend_cci': nan,
 'bitstamp:/LTC:trend_dpo': 41.511479785526944,
 'bitstamp:/LTC:trend_kst': -512.7312383060929,
 'bitstamp:/LTC:trend_kst_sig': -512.7312383060929,
 'bitstamp:/LTC:trend_kst_diff': 0.0,
 'bitstamp:/LTC:trend_ichimoku_a': 39.495000000000005,
 'bitstamp:/LTC:trend_ichimoku_b': 39.495000000000005,
 'bitstamp:/LTC:trend_visual_ichimoku_a': 80.84515204884308,
 'bitstamp:/LTC:trend_visual_ichimoku_b': 80.77039939728148,
 'bitstamp:/LTC:trend_aroon_up': 4.0,
 'bitstamp:/LTC:trend_aroon_down': 4.0,
 'bitstamp:/LTC:trend_aroon_ind': 0.0,
 'bitstamp:/LTC:momentum_rsi': nan,
 'bitstamp:/LTC:momentum_mfi': nan,
 'bitstamp:/LTC:momentum_tsi': -100.0,
 'bitstamp:/LTC:momentum_uo': 0.31218871344045224,
 'bitstamp:/LTC:momentum_stoch': 37.14285714285772,
 'bitstamp:/LTC:momentum_stoch_signal': 37.14285714285772,
 'bitstamp:/LTC:momentum_wr': -62.85714285714228,
 'bitstamp:/LTC:momentum_ao': 0.0,
 'bitstamp:/LTC:momentum_kama': nan,
 'bitstamp:/LTC:others_dr': -51.27312383060929,
 'bitstamp:/LTC:others_dlr': nan,
 'bitstamp:/LTC:others_cr': 0.0}
```

Portfolio

Make the portfolio using the any combinations of exchanges and intruments that the exchange supports

```
portfolio = Portfolio(USD, [
    Wallet(bitfinex, 10000 * USD),
    Wallet(bitfinex, 10 * BTC),
    Wallet(bitfinex, 5 * ETH),
    Wallet(bitstamp, 1000 * USD),
    Wallet(bitstamp, 5 * BTC),
    Wallet(bitstamp, 3 * LTC),
])
```

Environment

```
env = default.create(
    portfolio=portfolio,
    action_scheme="managed-risk",
    reward_scheme="simple",
    feed=feed,
    window_size=15,
    enable_logger=False
)
```

```
env.observer.feed.next()
```

```
{'internal': {'bitfinex:/USD-BTC': 2509.17,
  'bitfinex:/USD-ETH': 275.01,
  'bitfinex:/USD:/free': 10000.0,
  'bitfinex:/USD:/locked': 0.0,
  'bitfinex:/USD:/total': 10000.0,
  'bitfinex:/BTC:/free': 10.0,
  'bitfinex:/BTC:/locked': 0.0,
  'bitfinex:/BTC:/total': 10.0,
  'bitfinex:/BTC:/worth': 25091.7,
  'bitfinex:/ETH:/free': 5.0,
  'bitfinex:/ETH:/locked': 0.0,
  'bitfinex:/ETH:/total': 5.0,
  'bitfinex:/ETH:/worth': 1375.05,
  'bitstamp:/USD-BTC': 2500.0,
  'bitstamp:/USD-LTC': 39.45,
  'bitstamp:/USD:/free': 1000.0,
  'bitstamp:/USD:/locked': 0.0,
  'bitstamp:/USD:/total': 1000.0,
  'bitstamp:/BTC:/free': 5.0,
  'bitstamp:/BTC:/locked': 0.0,
  'bitstamp:/BTC:/total': 5.0,
  'bitstamp:/BTC:/worth': 12500.0,
  'bitstamp:/LTC:/free': 3.0,
  'bitstamp:/LTC:/locked': 0.0,
  'bitstamp:/LTC:/total': 3.0,
  'bitstamp:/LTC:/worth': 118.35000000000001,
  'net_worth': 50085.1},
 'external': {'bitfinex:/BTC:date': Timestamp('2017-07-01 11:00:00'),
  'bitfinex:/BTC:open': 2505.56,
  'bitfinex:/BTC:high': 2513.38,
  'bitfinex:/BTC:low': 2495.12,
  'bitfinex:/BTC:close': 2509.17,
  'bitfinex:/BTC:volume': 287000.32,
```

(continues on next page)

(continued from previous page)

```
 'bitfinex:/BTC:volume_adi': 462887.3781183644,
 'bitfinex:/BTC:volume_obv': nan,
 'bitfinex:/BTC:volume_cmf': 0.5388828039430464,
 'bitfinex:/BTC:volume_fi': nan,
 'bitfinex:/BTC:volume_em': nan,
 'bitfinex:/BTC:volume_vpt': -190920.02711825827,
 'bitfinex:/BTC:volume_nvi': 1000.0,
 'bitfinex:/BTC:volatility_atr': 85.51648155760596,
 'bitfinex:/BTC:volatility_bbh': 2509.17,
 'bitfinex:/BTC:volatility_bbl': 2509.17,
 'bitfinex:/BTC:volatility_bbm': 2509.17,
 'bitfinex:/BTC:volatility_bbhi': 0.0,
 'bitfinex:/BTC:volatility_bbli': 0.0,
 'bitfinex:/BTC:volatility_kcc': 2505.89,
 'bitfinex:/BTC:volatility_kch': 2524.15,
 'bitfinex:/BTC:volatility_kcl': 2487.6299999999997,
 'bitfinex:/BTC:volatility_kchi': 0.0,
 'bitfinex:/BTC:volatility_kcli': 0.0,
 'bitfinex:/BTC:volatility_dch': 2509.17,
 'bitfinex:/BTC:volatility_dcl': 2509.17,
 'bitfinex:/BTC:volatility_dchi': 0.0,
 'bitfinex:/BTC:volatility_dcli': 0.0,
 'bitfinex:/BTC:trend_macd': nan,
 'bitfinex:/BTC:trend_macd_signal': nan,
 'bitfinex:/BTC:trend_macd_diff': nan,
 'bitfinex:/BTC:trend_ema_fast': nan,
 'bitfinex:/BTC:trend_ema_slow': nan,
 'bitfinex:/BTC:trend_adx': 0.0,
 'bitfinex:/BTC:trend_adx_pos': 0.0,
 'bitfinex:/BTC:trend_adx_neg': 0.0,
 'bitfinex:/BTC:trend_vortex_ind_pos': nan,
 'bitfinex:/BTC:trend_vortex_ind_neg': nan,
 'bitfinex:/BTC:trend_vortex_diff': nan,
 'bitfinex:/BTC:trend_trix': nan,
 'bitfinex:/BTC:trend_mass_index': 0.0,
 'bitfinex:/BTC:trend_cci': nan,
 'bitfinex:/BTC:trend_dpo': 4963.073762705523,
 'bitfinex:/BTC:trend_kst': -664.2012654186367,
 'bitfinex:/BTC:trend_kst_sig': -664.2012654186367,
 'bitfinex:/BTC:trend_kst_diff': 0.0,
 'bitfinex:/BTC:trend_ichimoku_a': 2504.25,
 'bitfinex:/BTC:trend_ichimoku_b': 2504.25,
 'bitfinex:/BTC:trend_visual_ichimoku_a': 7460.129960014917,
 'bitfinex:/BTC:trend_visual_ichimoku_b': 7449.72498449202,
 'bitfinex:/BTC:trend_aroon_up': 4.0,
 'bitfinex:/BTC:trend_aroon_down': 4.0,
 'bitfinex:/BTC:trend_aroon_ind': 0.0,
 'bitfinex:/BTC:momentum_rsi': nan,
 'bitfinex:/BTC:momentum_mfi': nan,
 'bitfinex:/BTC:momentum_tsi': -100.0,
 'bitfinex:/BTC:momentum_uo': 0.2822915537138003,
 'bitfinex:/BTC:momentum_stoch': 76.94414019715232,
 'bitfinex:/BTC:momentum_stoch_signal': 76.94414019715232,
 'bitfinex:/BTC:momentum_wr': -23.055859802847678,
 'bitfinex:/BTC:momentum_ao': 0.0,
 'bitfinex:/BTC:momentum_kama': nan,
 'bitfinex:/BTC:others_dr': -66.42012654186367,
```

(continues on next page)

(continued from previous page)

```
 'bitfinex:/BTC:others_dlr': nan,
 'bitfinex:/BTC:others_cr': 0.0,
 'bitfinex:/ETH:date': Timestamp('2017-07-01 11:00:00'),
 'bitfinex:/ETH:open': 279.98,
 'bitfinex:/ETH:high': 279.99,
 'bitfinex:/ETH:low': 272.1,
 'bitfinex:/ETH:close': 275.01,
 'bitfinex:/ETH:volume': 679358.87,
 'bitstamp:/BTC:date': Timestamp('2017-07-01 11:00:00'),
 'bitstamp:/BTC:open': 2506.5,
 'bitstamp:/BTC:high': 2510.62,
 'bitstamp:/BTC:low': 2495.5,
 'bitstamp:/BTC:close': 2500.0,
 'bitstamp:/BTC:volume': 521903.7,
 'bitstamp:/LTC:date': Timestamp('2017-07-01 11:00:00'),
 'bitstamp:/LTC:open': 39.67,
 'bitstamp:/LTC:high': 39.67,
 'bitstamp:/LTC:low': 39.32,
 'bitstamp:/LTC:close': 39.45,
 'bitstamp:/LTC:volume': 1957.48,
 'bitstamp:/LTC:volume_adi': 5133.608444728016,
 'bitstamp:/LTC:volume_obv': nan,
 'bitstamp:/LTC:volume_cmf': -0.2571428571428455,
 'bitstamp:/LTC:volume_fi': nan,
 'bitstamp:/LTC:volume_em': nan,
 'bitstamp:/LTC:volume_vpt': -895.4321955984681,
 'bitstamp:/LTC:volume_nvi': 1000.0,
 'bitstamp:/LTC:volatility_atr': 1.4838450097847482,
 'bitstamp:/LTC:volatility_bbh': 39.45,
 'bitstamp:/LTC:volatility_bbl': 39.45,
 'bitstamp:/LTC:volatility_bbm': 39.45,
 'bitstamp:/LTC:volatility_bbhi': 0.0,
 'bitstamp:/LTC:volatility_bbli': 0.0,
 'bitstamp:/LTC:volatility_kcc': 39.480000000000004,
 'bitstamp:/LTC:volatility_kch': 39.830000000000005,
 'bitstamp:/LTC:volatility_kcl': 39.13,
 'bitstamp:/LTC:volatility_kchi': 0.0,
 'bitstamp:/LTC:volatility_kcli': 0.0,
 'bitstamp:/LTC:volatility_dch': 39.45,
 'bitstamp:/LTC:volatility_dcl': 39.45,
 'bitstamp:/LTC:volatility_dchi': 0.0,
 'bitstamp:/LTC:volatility_dcli': 0.0,
 'bitstamp:/LTC:trend_macd': nan,
 'bitstamp:/LTC:trend_macd_signal': nan,
 'bitstamp:/LTC:trend_macd_diff': nan,
 'bitstamp:/LTC:trend_ema_fast': nan,
 'bitstamp:/LTC:trend_ema_slow': nan,
 'bitstamp:/LTC:trend_adx': 0.0,
 'bitstamp:/LTC:trend_adx_pos': 0.0,
 'bitstamp:/LTC:trend_adx_neg': 0.0,
 'bitstamp:/LTC:trend_vortex_ind_pos': nan,
 'bitstamp:/LTC:trend_vortex_ind_neg': nan,
 'bitstamp:/LTC:trend_vortex_diff': nan,
 'bitstamp:/LTC:trend_trix': nan,
 'bitstamp:/LTC:trend_mass_index': 0.0,
 'bitstamp:/LTC:trend_cci': nan,
 'bitstamp:/LTC:trend_dpo': 41.511479785526944,
```

(continues on next page)

(continued from previous page)

```
  'bitstamp:/LTC:trend_kst': -512.7312383060929,
  'bitstamp:/LTC:trend_kst_sig': -512.7312383060929,
  'bitstamp:/LTC:trend_kst_diff': 0.0,
  'bitstamp:/LTC:trend_ichimoku_a': 39.495000000000005,
  'bitstamp:/LTC:trend_ichimoku_b': 39.495000000000005,
  'bitstamp:/LTC:trend_visual_ichimoku_a': 80.84515204884308,
  'bitstamp:/LTC:trend_visual_ichimoku_b': 80.77039939728148,
  'bitstamp:/LTC:trend_aroon_up': 4.0,
  'bitstamp:/LTC:trend_aroon_down': 4.0,
  'bitstamp:/LTC:trend_aroon_ind': 0.0,
  'bitstamp:/LTC:momentum_rsi': nan,
  'bitstamp:/LTC:momentum_mfi': nan,
  'bitstamp:/LTC:momentum_tsi': -100.0,
  'bitstamp:/LTC:momentum_uo': 0.31218871344045224,
  'bitstamp:/LTC:momentum_stoch': 37.14285714285772,
  'bitstamp:/LTC:momentum_stoch_signal': 37.14285714285772,
  'bitstamp:/LTC:momentum_wr': -62.85714285714228,
  'bitstamp:/LTC:momentum_ao': 0.0,
  'bitstamp:/LTC:momentum_kama': nan,
  'bitstamp:/LTC:others_dr': -51.27312383060929,
  'bitstamp:/LTC:others_dlr': nan,
  'bitstamp:/LTC:others_cr': 0.0}}
```

1.6 Train and Evaluate using Ray

This doc has been updated to leverage Ray as a training backend. Builtin agents have been deprecated, since Ray offers a much more stable and faster execution, on top of the parallelized training\evaluation in a distributed cluster.

> We will be using the fictional TensorTrade Corp stock (**TTRD**) traded in the TensorTrade Stock Exchange (**TTSE**) in all the examples, hoping noone goes public under that ticker and generates some confusion :)
> The base currency will be US Dollars (**USD**) with a commission of 0.35% on each trade done.

1.6.1 Installing Requirements

TensorTrade

You can follow the "easy" way and install using the provided PIP package: (latest version available at the time of this example is 1.0.3)

```
pip install tensortrade
```

... or you can install the latest code available, pulling directly from the development codebase (this is the suggested way, since many fixes take time to get packaged in a new release)

```
pip install git+https://github.com/tensortrade-org/tensortrade.git
```

Ray

This example focuses on having Ray as a training backend, so we need to install `ray` with the `default` base features and `rrlib` and `tune` features (Latest version available at the time of this example is 1.8.0)

```
pip install ray[default,rllib,tune]==1.8.0
```

Ray has many builtins RLLib algorithms to train agents with, and together with Tune it allows us to run multiple parallel (or sequential) trainings with different parameters in a grid search for the best one to use within TensorTrade.

> Pay attention to where you are installing Ray. Until recently (1.8.0 does *NOT* have this issue, but previous versions do), if Ray was installed in a VENV, it would hang at `ray.init()` during execution. If you need to be using Ray versions prior to 1.8.0 please make sure you are installing Ray in the default system environment.

Other Requirements

On top of TensorTrade and Ray, you need to install whichever library you want to use to fetch the data you want to train on, and any library you plan to use to augment your source data with calculated technical analysis fields such as `log return`, `rsi`, `macd` and so on... In this example I am using `yfinance` to fetch stock ticker data and `pandas-ta` to add technical indicators. I know pandas-ta is not actively mantained, but it's well enough for what I need, and it's really simple to use (you'll see later). Also, for pandas-ta, you need to specify the `--pre` parameter when installing the package since pandas-ta has not had a stable release yet, and pip only looks for stable releases by default.

```
pip install yfinance==0.1.64
pip install pandas-ta==0.3.14b --pre
```

Of course you can choose whichever providers\augmenters you want, and this is just an example to show one possible implementation.

1.6.2 Build the training\evaluation dataset

We now arbitrarily define the timeframes we want to use in training our agent. Please note **you have to substitute the ticker name**, as it appears on Yahoo! Finance (ie: Gold Futures is `GC=F`, CBOE Volatility Index is `^VIX`, GameStop Corp is `GME`, and so on...).

Once retrieved we need to add the relevant TA values, and this is enabled just by importing `pandas_ta` and then accessing the `ta` child in a dataframe. The `#noqa` tag is to silence PyCharm from reporting that `pandas_ta` has been imported but it has not been used (which is not true, but it can't know that)

In the end, to avoid continuous Yahoo! Finance API requests, we simply save those dataframes to CSV, so that we can then simply read those preprocessed files during training and evaluation.

```
import yfinance
import pandas_ta  #noqa
TICKER = 'TTRD'  # TODO: replace this with your own ticker
TRAIN_START_DATE = '2021-02-09'  # TODO: replace this with your own start date
TRAIN_END_DATE = '2021-09-30'  # TODO: replace this with your own end date
EVAL_START_DATE = '2021-10-01'  # TODO: replace this with your own end date
EVAL_END_DATE = '2021-11-12'  # TODO: replace this with your own end date

yf_ticker = yfinance.Ticker(ticker=TICKER)

df_training = yf_ticker.history(start=TRAIN_START_DATE, end=TRAIN_END_DATE, interval=
↪'60m')
df_training.drop(['Dividends', 'Stock Splits'], axis=1, inplace=True)
df_training["Volume"] = df_training["Volume"].astype(int)
df_training.ta.log_return(append=True, length=16)
```

(continues on next page)

(continued from previous page)

```
df_training.ta.rsi(append=True, length=14)
df_training.ta.macd(append=True, fast=12, slow=26)
df_training.to_csv('training.csv', index=False)

df_evaluation = yf_ticker.history(start=EVAL_START_DATE, end=EVAL_END_DATE, interval=
↪'60m')
df_evaluation.drop(['Dividends', 'Stock Splits'], axis=1, inplace=True)
df_evaluation["Volume"] = df_evaluation["Volume"].astype(int)
df_evaluation.ta.log_return(append=True, length=16)
df_evaluation.ta.rsi(append=True, length=14)
df_evaluation.ta.macd(append=True, fast=12, slow=26)
df_evaluation.to_csv('evaluation.csv', index=False)
```

We should now have the two preprocessed files ready (`training.csv` and `evaluation.csv`)

> Please note that there are many better ways to do this. For example there is an obvious issue in doing things this way: all TA values that we have added require a minimum of 12 samples to be processed, since they perform calculations on longer timeframes. This means that the first 12 rows will not have any meaningful TA values to be trained (or evaluated) with.
>
> This additional care in retrieving and preprocessing data is left to the user to implement, since many different approaches can be taken, each one with its pros and cons

1.6.3 Training and Evaluation

Create the environment build function

Here we are using the `config` dictionary to store the CSV filename that we need to read. During the training phase, we will pass `training.csv` as the value, while during the evaluation phase we will pass `evaluation.csv`

```
import pandas as pd
from tensortrade.feed.core import DataFeed, Stream
from tensortrade.oms.instruments import Instrument
from tensortrade.oms.exchanges import Exchange, ExchangeOptions
from tensortrade.oms.services.execution.simulated import execute_order
from tensortrade.oms.wallets import Wallet, Portfolio
import tensortrade.env.default as default

def create_env(config):
    dataset = pd.read_csv(filepath_or_buffer=config["csv_filename"], parse_dates=[
↪'Datetime']).fillna(method='backfill').fillna(method='ffill')
    ttse_commission = 0.0035  # TODO: adjust according to your commission percentage,␣
↪if present
    price = Stream.source(list(dataset["Close"]), dtype="float").rename("USD-TTRD")
    ttse_options = ExchangeOptions(commission=ttse_commission)
    ttse_exchange = Exchange("TTSE", service=execute_order, options=ttse_
↪options)(price)

 # Instruments, Wallets and Portfolio
    USD = Instrument("USD", 2, "US Dollar")
    TTRD = Instrument("TTRD", 2, "TensorTrade Corp")
    cash = Wallet(ttse_exchange, 1000 * USD)  # This is the starting cash we are␣
↪going to use
    asset = Wallet(ttse_exchange, 0 * TTRD)  # And we will start owning 0 stocks of␣
↪TTRD
```

(continues on next page)

(continued from previous page)

```
    portfolio = Portfolio(USD, [cash, asset])

    # Renderer feed
    renderer_feed = DataFeed([
        Stream.source(list(dataset["Datetime"])).rename("date"),
        Stream.source(list(dataset["Open"]), dtype="float").rename("open"),
        Stream.source(list(dataset["High"]), dtype="float").rename("high"),
        Stream.source(list(dataset["Low"]), dtype="float").rename("low"),
        Stream.source(list(dataset["Close"]), dtype="float").rename("close"),
        Stream.source(list(dataset["Volume"]), dtype="float").rename("volume")
    ])

    features = []
    for c in dataset.columns[1:]:
        s = Stream.source(list(dataset[c]), dtype="float").rename(dataset[c].name)
        features += [s]
    feed = DataFeed(features)
    feed.compile()

    reward_scheme = default.rewards.SimpleProfit(window_size=config["reward_window_
↪size"])
    action_scheme = default.actions.BSH(cash=cash, asset=asset)

    env = default.create(
            feed=feed,
            portfolio=portfolio,
            action_scheme=action_scheme,
            reward_scheme=reward_scheme,
            renderer_feed=renderer_feed,
            renderer=[],
            window_size=config["window_size"],
            max_allowed_loss=config["max_allowed_loss"]
        )

    return env
```

Initialize and run Ray

Now it's time to actually initialize and run Ray, passing all the parameters necessary, including the name of the environment creator function (`create_env` defined above).

Please note that many of these parameters need to be tuned for your specific use case (ie: `"training_iteration": 5` is ***way*** too few to get anything remotely useful, but it allows the example to quickly reach the end)

```
import ray
import os
from ray import tune
from ray.tune.registry import register_env

# Let's define some tuning parameters
FC_SIZE = tune.grid_search([[256, 256], [1024], [128, 64, 32]])  # Those are the␣
↪alternatives that ray.tune will try...
LEARNING_RATE = tune.grid_search([0.001, 0.0005, 0.00001])  # ... and they will be␣
↪combined with these ones ...
```

(continues on next page)

(continued from previous page)

```
MINIBATCH_SIZE = tune.grid_search([5, 10, 20])  # ... and these ones, in a cartesian␣
↪product.

# Get the current working directory
cwd = os.getcwd()

# Initialize Ray
ray.init()  # There are *LOTS* of initialization parameters, like specifying the␣
↪maximum number of CPUs\GPUs to allocate. For now just leave it alone.

# Register our environment, specifying which is the environment creation function
register_env("MyTrainingEnv", create_env)

# Specific configuration keys that will be used during training
env_config_training = {
    "window_size": 14,  # We want to look at the last 14 samples (hours)
    "reward_window_size": 7,  # And calculate reward based on the actions taken in␣
↪the next 7 hours
    "max_allowed_loss": 0.10,  # If it goes past 10% loss during the iteration, we don
↪'t want to waste time on a "loser".
    "csv_filename": os.path.join(cwd, 'training.csv'),  # The variable that will be␣
↪used to differentiate training and validation datasets
}
# Specific configuration keys that will be used during evaluation (only the␣
↪overridden ones)
env_config_evaluation = {
    "max_allowed_loss": 1.00,  # During validation runs we want to see how bad it␣
↪would go. Even up to 100% loss.
    "csv_filename": os.path.join(cwd, 'evaluation.csv'),  # The variable that will be␣
↪used to differentiate training and validation datasets
}

analysis = tune.run(
    run_or_experiment="PPO",  # We'll be using the builtin PPO agent in RLLib
    name="MyExperiment1",
    metric='episode_reward_mean',
    mode='max',
    stop={
        "training_iteration": 5  # Let's do 5 steps for each hyperparameter␣
↪combination
    },
    config={
        "env": "MyTrainingEnv",
        "env_config": env_config_training,  # The dictionary we built before
        "log_level": "WARNING",
        "framework": "torch",
        "ignore_worker_failures": True,
        "num_workers": 1,  # One worker per agent. You can increase this but it will␣
↪run fewer parallel trainings.
        "num_envs_per_worker": 1,
        "num_gpus": 0,  # I yet have to understand if using a GPU is worth it, for␣
↪our purposes, but I think it's not. This way you can train on a non-gpu enabled␣
↪system.
        "clip_rewards": True,
        "lr": LEARNING_RATE,  # Hyperparameter grid search defined above
        "gamma": 0.50,  # This can have a big impact on the result and needs to be␣
↪properly tuned (range is 0 to 1)
```

(continues on next page)

(continued from previous page)

```
        "observation_filter": "MeanStdFilter",
        "model": {
            "fcnet_hiddens": FC_SIZE,  # Hyperparameter grid search defined above
        },
        "sgd_minibatch_size": MINIBATCH_SIZE,  # Hyperparameter grid search defined␣
↪above
        "evaluation_interval": 1,  # Run evaluation on every iteration
        "evaluation_config": {
            "env_config": env_config_evaluation,  # The dictionary we built before␣
↪(only the overriding keys to use in evaluation)
            "explore": False,  # We don't want to explore during evaluation. All␣
↪actions have to be repeatable.
        },
    },
    num_samples=1,  # Have one sample for each hyperparameter combination. You can␣
↪have more to average out randomness.
    keep_checkpoints_num=10,  # Keep the last 2 checkpoints
    checkpoint_freq=1,  # Do a checkpoint on each iteration (slower but you can pick␣
↪more finely the checkpoint to use later)
)
```

Once you launch this, it will block (meaning it will stay running until the stop condition happens for all samples). You will receive a console output that will show the training progress and all the hyperparameters that are being used in each trial.

1.6.4 Monitoring

You can basically monitor two things: how Ray is behaving on your cluster (local or distributed, in this example it will be a local cluster), and how is the training proceeding within the TensorTrade environment.

Ray Dashboard

The Ray Dashboard can be accessed by default at http://127.0.0.1:8265. If you want to access it remotely, you just need to specify `dashboard_host="0.0.0.0"` as a ray.init() parameter. This will allow external\remote connections to the Dashboard, provided the newtork routing\accessibility and eventual firewall is correctly configured.

The Dashboard will show resource usage statistics on the nodes working on the cluster, most importantly CPU and RAM usage. Please refer to the official dashboard documentation for further info on that.

TensorBoard

In order to browse the TensorBoard you first need to launch it, running this console command:

```
tensorboard --logdir path\to\Ray\results\folder
```

You can then access it by default at http://127.0.0.1:6006. As with the Ray Dashboard, if you want to access it remotely, you need to specify `--host 0.0.0.0` in the commandline parameter, like:

```
tensorboard --logdir path\to\Ray\results\folder --host 0.0.0.0
```

The most important values you need to watch out for during a training are `tune/episode_reward_min`, `tune/episode_reward_mean` and `tune/episode_reward_max` that represent the minimum, average and maximum reward obtained by the agent during training on that specific iteration, using the training dataset.

Alongside with those three, there are the evaluation counterparts, so `tune/evaluation/episode_reward_min`, `tune/evaluation/episode_reward_mean` and `tune/evaluation/episode_reward_max` which represent the same metric, calculated on the evaluation dataset.

The best model will be the one that will score the highest evaluation values, given that the training values will be always higher\better than the evaluation ones.

1.7 Renderers and Plotly Chart

Data Loading Function

```
# ipywidgets is required to run Plotly in Jupyter Notebook.
# Uncomment and run the following line to install it if required.

!pip install ipywidgets
```

```
import ta

import pandas as pd

from tensortrade.feed.core import Stream, DataFeed, NameSpace
from tensortrade.oms.exchanges import Exchange
from tensortrade.oms.services.execution.simulated import execute_order
from tensortrade.oms.instruments import USD, BTC
from tensortrade.oms.wallets import Wallet, Portfolio

%matplotlib inline
```

```
def load_csv(filename):
    df = pd.read_csv('data/' + filename, skiprows=1)
    df.drop(columns=['symbol', 'volume_btc'], inplace=True)

    # Fix timestamp form "2019-10-17 09-AM" to "2019-10-17 09-00-00 AM"
    df['date'] = df['date'].str[:14] + '00-00 ' + df['date'].str[-2:]

    # Convert the date column type from string to datetime for proper sorting.
    df['date'] = pd.to_datetime(df['date'])

    # Make sure historical prices are sorted chronologically, oldest first.
    df.sort_values(by='date', ascending=True, inplace=True)

    df.reset_index(drop=True, inplace=True)

    # Format timestamps as you want them to appear on the chart buy/sell marks.
    df['date'] = df['date'].dt.strftime('%Y-%m-%d %I:%M %p')

    return df
```

```
df = load_csv('Bitfinex_BTCUSD_1h.csv')
df.head()
```

Data Preparation

```
dataset = ta.add_all_ta_features(df, 'open', 'high', 'low', 'close', 'volume',␣
↪fillna=True)
dataset.head(3)
```

Note: It is recommended to create the chart data *after* creating and cleaning the dataset to ensure one-to-one mapping between the historical prices data and the dataset.

```
price_history = dataset[['date', 'open', 'high', 'low', 'close', 'volume']]  # chart␣
↪data
display(price_history.head(3))

dataset.drop(columns=['date', 'open', 'high', 'low', 'close', 'volume'], inplace=True)
```

Setup Trading Environment

```
bitfinex = Exchange("bitfinex", service=execute_order)(
    Stream.source(price_history['close'].tolist(), dtype="float").rename("USD-BTC")
)

portfolio = Portfolio(USD, [
    Wallet(bitfinex, 10000 * USD),
    Wallet(bitfinex, 10 * BTC),
])

with NameSpace("bitfinex"):
    streams = [Stream.source(dataset[c].tolist(), dtype="float").rename(c) for c in␣
↪dataset.columns]

feed = DataFeed(streams)
feed.next()
```

```
{'bitfinex:/volume_adi': 503274.35945218964,
 'bitfinex:/volume_obv': 0.0,
 'bitfinex:/volume_cmf': 0.5388828039430464,
 'bitfinex:/volume_fi': 0.0,
 'bitfinex:/volume_em': 0.0,
 'bitfinex:/volume_vpt': -187039.68188942864,
 'bitfinex:/volume_nvi': 1000.0,
 'bitfinex:/volatility_atr': 88.87448632521046,
 'bitfinex:/volatility_bbh': 2509.17,
 'bitfinex:/volatility_bbl': 2509.17,
 'bitfinex:/volatility_bbm': 2509.17,
 'bitfinex:/volatility_bbhi': 0.0,
 'bitfinex:/volatility_bbli': 0.0,
 'bitfinex:/volatility_kcc': 2505.89,
 'bitfinex:/volatility_kch': 2524.15,
 'bitfinex:/volatility_kcl': 2487.6299999999997,
 'bitfinex:/volatility_kchi': 0.0,
 'bitfinex:/volatility_kcli': 0.0,
 'bitfinex:/volatility_dch': 2509.17,
 'bitfinex:/volatility_dcl': 2509.17,
 'bitfinex:/volatility_dchi': 0.0,
 'bitfinex:/volatility_dcli': 0.0,
 'bitfinex:/trend_macd': 0.0,
 'bitfinex:/trend_macd_signal': 0.0,
 'bitfinex:/trend_macd_diff': 0.0,
 'bitfinex:/trend_ema_fast': 2509.17,
```

(continues on next page)

(continued from previous page)

```
'bitfinex:/trend_ema_slow': 2509.17,
'bitfinex:/trend_adx': 0.0,
'bitfinex:/trend_adx_pos': 0.0,
'bitfinex:/trend_adx_neg': 0.0,
'bitfinex:/trend_vortex_ind_pos': 1.0,
'bitfinex:/trend_vortex_ind_neg': 1.0,
'bitfinex:/trend_vortex_diff': 0.0,
'bitfinex:/trend_trix': -65.01942947444225,
'bitfinex:/trend_mass_index': 1.0,
'bitfinex:/trend_cci': 0.0,
'bitfinex:/trend_dpo': 4669.658895132072,
'bitfinex:/trend_kst': -650.476416605854,
'bitfinex:/trend_kst_sig': -650.476416605854,
'bitfinex:/trend_kst_diff': 0.0,
'bitfinex:/trend_ichimoku_a': 2504.25,
'bitfinex:/trend_ichimoku_b': 2504.25,
'bitfinex:/trend_visual_ichimoku_a': 7164.427851548871,
'bitfinex:/trend_visual_ichimoku_b': 7151.343258415852,
'bitfinex:/trend_aroon_up': 4.0,
'bitfinex:/trend_aroon_down': 4.0,
'bitfinex:/trend_aroon_ind': 0.0,
'bitfinex:/momentum_rsi': 50.0,
'bitfinex:/momentum_mfi': 50.0,
'bitfinex:/momentum_tsi': -100.0,
'bitfinex:/momentum_uo': 0.29997594458961346,
'bitfinex:/momentum_stoch': 76.94414019715232,
'bitfinex:/momentum_stoch_signal': 76.94414019715232,
'bitfinex:/momentum_wr': -23.055859802847678,
'bitfinex:/momentum_ao': 0.0,
'bitfinex:/momentum_kama': 2509.17,
'bitfinex:/others_dr': -65.0476416605854,
'bitfinex:/others_dlr': 0.0,
'bitfinex:/others_cr': 0.0}
```

Trading Environment Renderers A renderer is a channel for the trading environment to output its current state. One or more renderers can be attached to the environment at the same time. For example, you can let the environment draw a chart and log to a file at the same time.

Notice that while all renderers can technically be used together, you need to select the best combination to avoid undesired results. For example, PlotlyTradingChart can work well with FileLogger but may not display well with ScreenLogger.

Renderer can be set by name (string) or class, single or list. Available renderers are:

- `'screenlog'` or `ScreenLogger`: Shows results on the screen.
- `'filelog'` or `FileLogger`: Logs results to a file.
- `'plotly'` or `PlotlyTradingChart`: A trading chart based on Plotly.

Examples:

- renderers = 'screenlog' (default)
- renderers = ['screenlog', 'filelog']
- renderers = ScreenLogger()
- renderers = ['screenlog', `FileLogger()`]
- renderers = [`FileLogger(filename='example.log')`]

Renderers can also be created and configured first then attached to the environment as seen in a following example.

```
import tensortrade.env.default as default

env = default.create(
    portfolio=portfolio,
    action_scheme="managed-risk",
    reward_scheme="risk-adjusted",
    feed=feed,
    renderer="screen-log",  # ScreenLogger used with default settings
    window_size=20
)
```

```
from tensortrade.agents import DQNAgent

agent = DQNAgent(env)
agent.train(n_episodes=2, n_steps=200, render_interval=10)
```

```
====      AGENT ID: b8b6ad1a-c158-4743-8a2d-aab3a26ce82c      ====
[2020-07-29 3:43:06 PM] Episode: 1/2 Step: 131/200
[2020-07-29 3:43:07 PM] Episode: 1/2 Step: 141/200
[2020-07-29 3:43:08 PM] Episode: 1/2 Step: 151/200
[2020-07-29 3:43:09 PM] Episode: 1/2 Step: 161/200
[2020-07-29 3:43:11 PM] Episode: 1/2 Step: 171/200
[2020-07-29 3:43:12 PM] Episode: 1/2 Step: 181/200
[2020-07-29 3:43:13 PM] Episode: 1/2 Step: 191/200
[2020-07-29 3:43:14 PM] Episode: 1/2 Step: 201/200
[2020-07-29 3:43:15 PM] Episode: 2/2 Step: 11/200
[2020-07-29 3:43:16 PM] Episode: 2/2 Step: 21/200
[2020-07-29 3:43:17 PM] Episode: 2/2 Step: 31/200
[2020-07-29 3:43:19 PM] Episode: 2/2 Step: 41/200
[2020-07-29 3:43:20 PM] Episode: 2/2 Step: 51/200
[2020-07-29 3:43:21 PM] Episode: 2/2 Step: 61/200
[2020-07-29 3:43:22 PM] Episode: 2/2 Step: 71/200
[2020-07-29 3:43:23 PM] Episode: 2/2 Step: 81/200
[2020-07-29 3:43:24 PM] Episode: 2/2 Step: 91/200
[2020-07-29 3:43:25 PM] Episode: 2/2 Step: 101/200
[2020-07-29 3:43:26 PM] Episode: 2/2 Step: 111/200
[2020-07-29 3:43:27 PM] Episode: 2/2 Step: 121/200
[2020-07-29 3:43:29 PM] Episode: 2/2 Step: 131/200
[2020-07-29 3:43:30 PM] Episode: 2/2 Step: 141/200
[2020-07-29 3:43:31 PM] Episode: 2/2 Step: 151/200
[2020-07-29 3:43:32 PM] Episode: 2/2 Step: 161/200
[2020-07-29 3:43:33 PM] Episode: 2/2 Step: 171/200
[2020-07-29 3:43:34 PM] Episode: 2/2 Step: 181/200
[2020-07-29 3:43:35 PM] Episode: 2/2 Step: 191/200
[2020-07-29 3:43:36 PM] Episode: 2/2 Step: 201/200




-125697.1219732128
```

Environment with Multiple Renderers Create PlotlyTradingChart and FileLogger renderers. Configuring renderers is optional as they can be used with their default settings.

```
from tensortrade.env.default.renderers import PlotlyTradingChart, FileLogger

chart_renderer = PlotlyTradingChart(
    display=True,  # show the chart on screen (default)
    height=800,  # affects both displayed and saved file height. None for 100% height.
    save_format="html",  # save the chart to an HTML file
    auto_open_html=True,  # open the saved HTML chart in a new browser tab
)

file_logger = FileLogger(
    filename="example.log",  # omit or None for automatic file name
    path="training_logs"  # create a new directory if doesn't exist, None for no␣
↪directory
)
```

Environement with Multiple Renderers

With the plotly renderer you must provide an parameter called `renderer_feed`. This is a `DataFeed` instance that provides all the information that is required by a renderer to function.

```
renderer_feed = DataFeed([
    Stream.source(price_history[c].tolist(), dtype="float").rename(c) for c in price_
↪history]
)

env = default.create(
    portfolio=portfolio,
    action_scheme="managed-risk",
    reward_scheme="risk-adjusted",
    feed=feed,
    window_size=20,
    renderer_feed=renderer_feed,
    renderers=[
        chart_renderer,
        file_logger
    ]
)
```

Setup and Train DQN Agent The green and red arrows shown on the chart represent buy and sell trades respectively. The head of each arrow falls at the trade execution price.

```
from tensortrade.agents import DQNAgent

agent = DQNAgent(env)

# Set render_interval to None to render at episode ends only
agent.train(n_episodes=2, n_steps=200, render_interval=10)
```

```
====      AGENT ID: fec5e2c5-eb35-4ff6-8416-b876f0e8be66      ====
```


[2022-01-16 02:50:36 AM] Episode: 2/2 Step: 19
2650
2600
2550
2500
2450
2400
3M
2M
1M
0
30k

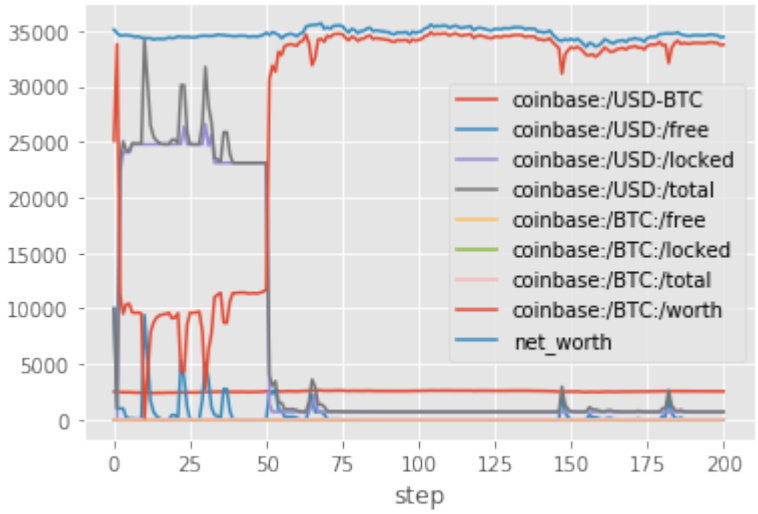
```
-122271.41943956864
```

Direct Performance and Net Worth Plotting Alternatively, the final performance and net worth can be displayed using `Pandas` via `Matplotlib`.

```
%matplotlib inline
import pandas as pd
df = pd.DataFrame(portfolio.performance)
```

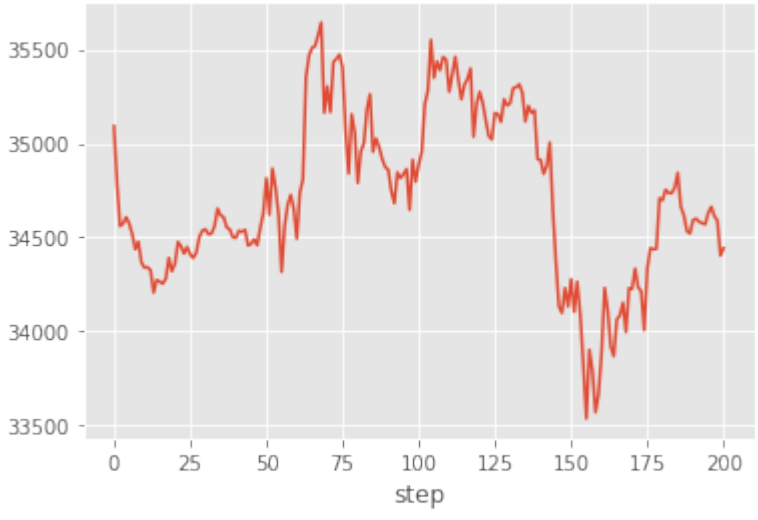
Transpose the dataframe using `T` and call `.plot()`

```
df.T.plot()
```



png

```
df.loc["net_worth"].plot()
```



png

1.8 Stochastic Data

Generating Price Data using Stocastic Processes

- Geometric Brownian Motion (GBM)
- Fractional Brownian Motion (FBM)
- Heston Stochastic Volatility Model
- Cox Ingersoll Ross (CIR)
- Ornstein Uhlebneck stochastic process

Model Parameters

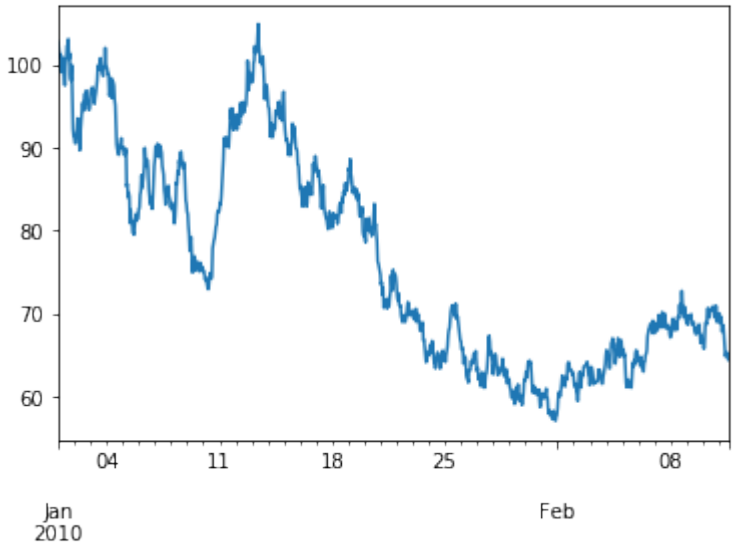
The model parameters class contains all of the parameters used by the following stochastic processes. The parameters have been prefixed with the name of the stochastic process they are used in. Calibration of the stochastic processes would involve looking for the parameter values which best fit some historical data.

- `all_s0` This is the starting asset value
- `all_time` This is the amount of time to simulate for
- `all_delta` This is the delta, the rate of time e.g. 1/252 = daily, 1/12 = monthly
- `all_sigma` This is the volatility of the stochastic processes
- `gbm_mu` This is the annual drift factor for geometric brownian motion
- `jumps_lamda` This is the probability of a jump happening at each point in time
- `jumps_sigma` This is the volatility of the jump size
- `jumps_mu` This is the average jump size
- `cir_a` This is the rate of mean reversion for Cox Ingersoll Ross
- `cir_mu` This is the long run average interest rate for Cox Ingersoll Ross
- `all_r0` This is the starting interest rate value
- `cir_rho` This is the correlation between the wiener processes of the Heston model
- `ou_a` This is the rate of mean reversion for Ornstein Uhlenbeck
- `ou_mu` This is the long run average interest rate for Ornstein Uhlenbeck
- `sheston_a` This is the rate of mean reversion for volatility in the Heston model
- `heston_mu` This is the long run average volatility for the Heston model
- `heston_vol0` This is the starting volatility value for the Heston model

```
import random

import tensortrade.stochastic as sp

%matplotlib inline
```

Geometric Brownian Motion

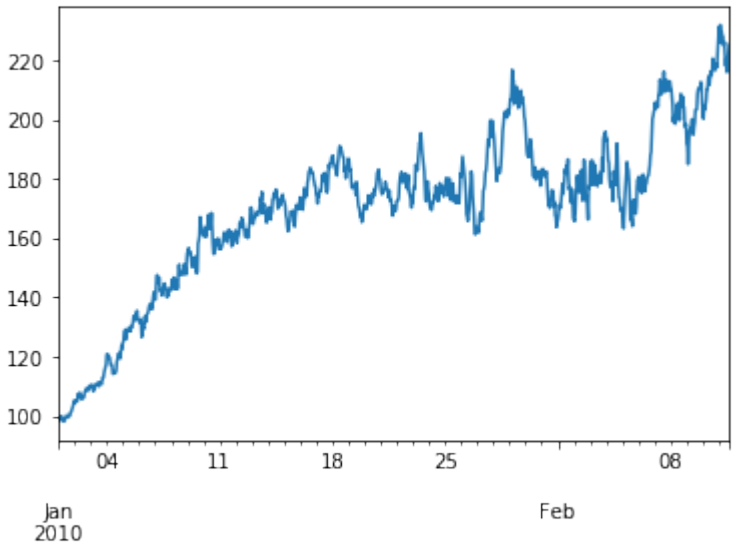
```
data = sp.gbm(
    base_price=100,
    base_volume=5,
    start_date="2010-01-01",
    times_to_generate=1000,
    time_frame='1H'
)

data.close.plot()
```



png

Heston Stochastic Volatility Model

```
data = sp.heston(
    base_price=100,
    base_volume=5,
    start_date="2010-01-01",
    times_to_generate=1000,
    time_frame='1H'
)

data.close.plot()
```



png

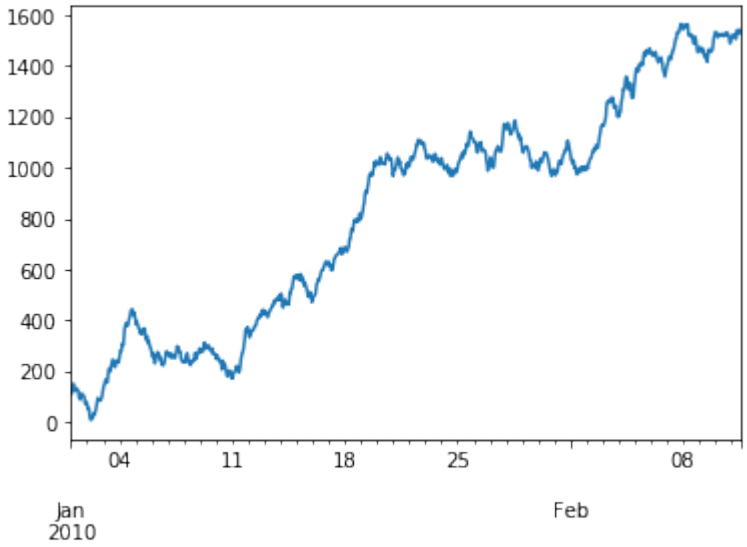
Fractional Brownian Motion

```
data = sp.fbm(
    base_price=100,
    base_volume=5,
    start_date="2010-01-01",
```

(continues on next page)

(continued from previous page)

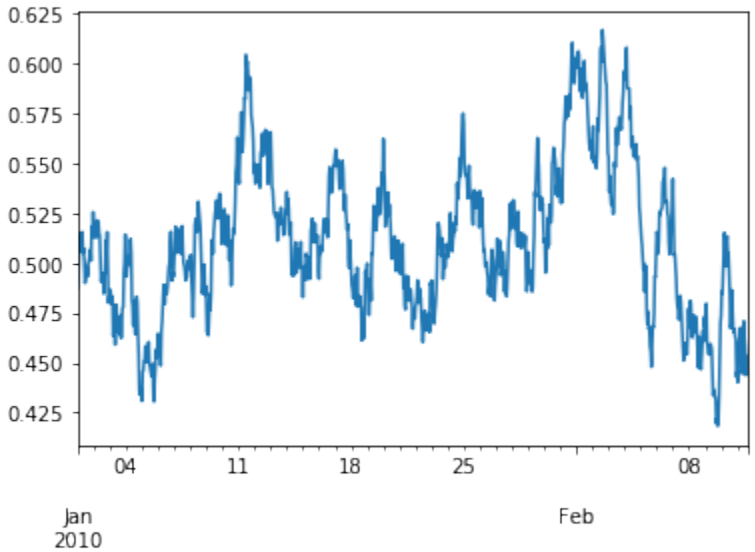
```
    times_to_generate=1000,
    time_frame='1H'
)

data.close.plot()
```



png

Cox Ingersoll Ross (CIR)

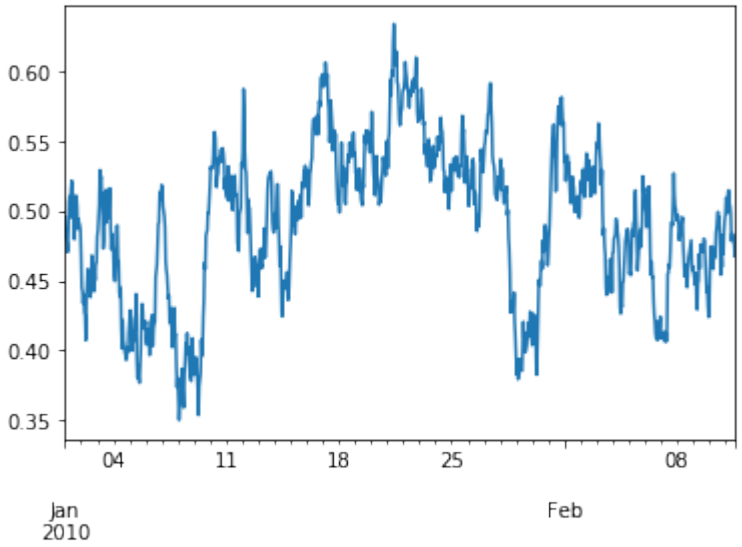
```
data = sp.cox(
    base_price=100,
    base_volume=5,
    start_date="2010-01-01",
    times_to_generate=1000,
    time_frame='1H'
)

data.close.plot()
```



png

Ornstein Uhlenbeck Process

```
data = sp.ornstein(
    base_price=100,
    base_volume=5,
    start_date="2010-01-01",
    times_to_generate=1000,
    time_frame='1H'
)

data.close.plot()
```



png

1.9 Ledger Example

Install master branch of TensorTrade

```
!pip install git+https://github.com/tensortrade-org/tensortrade.git -U
```

```
import tensortrade.env.default as default

from tensortrade.feed.core import Stream, DataFeed
from tensortrade.data.cdd import CryptoDataDownload
from tensortrade.oms.wallets import Portfolio, Wallet
from tensortrade.oms.exchanges import Exchange
from tensortrade.oms.services.execution.simulated import execute_order
from tensortrade.oms.instruments import USD, BTC, ETH, LTC
```

Load Data for Exchanges

Using the `tensortrade.data.cdd` module you can load data from any `csv` file provided at:

- https://www.cryptodatadownload.com/data/northamerican/

using the `CryptoDataDownload` class.

```
cdd = CryptoDataDownload()
bitfinex_btc = cdd.fetch("Bitfinex", "USD", "BTC", "1h")
bitfinex_eth = cdd.fetch("Bitfinex", "USD", "ETH", "1h")

bitstamp_btc = cdd.fetch("Bitstamp", "USD", "BTC", "1h")
bitstamp_eth = cdd.fetch("Bitstamp", "USD", "ETH", "1h")
bitstamp_ltc = cdd.fetch("Bitstamp", "USD", "LTC", "1h")
```

Inspect Transactions

```
bitfinex = Exchange("bitfinex", service=execute_order)(
    Stream.source(list(bitfinex_btc['close'][-100:]), dtype="float").rename("USD-BTC
↪"),
    Stream.source(list(bitfinex_eth['close'][-100:]), dtype="float").rename("USD-ETH")
)

bitstamp = Exchange("bitstamp", service=execute_order)(
    Stream.source(list(bitstamp_btc['close'][-100:]), dtype="float").rename("USD-BTC
↪"),
    Stream.source(list(bitstamp_eth['close'][-100:]), dtype="float").rename("USD-ETH
↪"),
    Stream.source(list(bitstamp_ltc['close'][-100:]), dtype="float").rename("USD-LTC")
)

portfolio = Portfolio(USD, [
    Wallet(bitfinex, 10000 * USD),
    Wallet(bitfinex, 10 * BTC),
    Wallet(bitfinex, 5 * ETH),
    Wallet(bitstamp, 1000 * USD),
    Wallet(bitstamp, 5 * BTC),
    Wallet(bitstamp, 20 * ETH),
    Wallet(bitstamp, 3 * LTC)
])

feed = DataFeed([
    Stream.source(list(bitstamp_eth['volume'][-100:]), dtype="float").rename("volume:/
↪USD-ETH"),
    Stream.source(list(bitstamp_ltc['volume'][-100:]), dtype="float").rename("volume:/
↪USD-LTC")
])

env = default.create(
    portfolio=portfolio,
    action_scheme=default.actions.SimpleOrders(),
    reward_scheme=default.rewards.SimpleProfit(),
    feed=feed
)

done = False
obs = env.reset()
while not done:
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)

portfolio.ledger.as_frame().head(7)
```

Inspect Transactions of ManagedRiskOrders

```
portfolio = Portfolio(USD, [
    Wallet(bitfinex, 10000 * USD),
    Wallet(bitfinex, 0 * BTC),
    Wallet(bitfinex, 0 * ETH),
])

env = default.create(
    portfolio=portfolio,
    action_scheme=default.actions.ManagedRiskOrders(),
    reward_scheme=default.rewards.SimpleProfit(),
    feed=feed
)

done = False

while not done:
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)

portfolio.ledger.as_frame().head(20)
```

Transactions in Spreadsheets

To take a closer look at the transactions that are happening within the system, copy the transactions to a csv file and load it into any spreadsheet software. If where you are running this allows access to the system clipboard, you can directly copy the frame to your system clipboard by doing the following:

- `portfolio.ledger.as_frame().to_clipboard(index=False)`

Then just paste into any spreadsheet software (e.g. Execel, Google Sheets).

1.10 Using Ray with TensorTrade

In this tutorial, we are going to learn how to use `ray` with TensorTrade in order to create a profitable algorithm on a predictable sine curve. You may be asking yourself, why use something so simple when the real world data is much more difficult to predict? Now this is a very good question and there is a simple answer.

> "The man who moves a mountain begins by carrying away small stones."
>
> - Confucius

Before trying to jump into the world of complex trading environments, a simple sine curve can be used to perform a sanity check on your trading algorithm. The reward and action scheme you use should be able to make money on the predictable pattern of a sine curve. If it does not, then you know there is no possibility success will be found on a more complex environment. There are some answers we would like to know fast before we waste time and resources in developing an algorithm which are and one of them is, does our `RewardScheme` correctly specify the goal we are after?

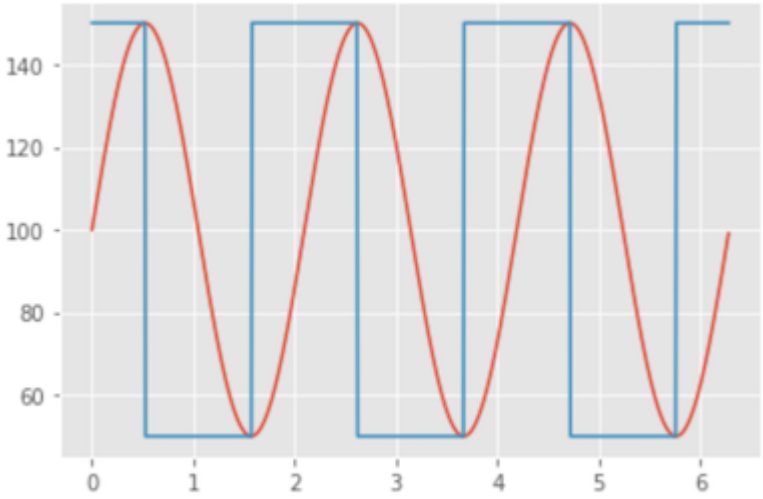
In this tutorial we will propose a new reward scheme and action scheme and show that you can actually have the reward scheme be dependent on the action scheme. This will be done through the use of a `DataFeed`. We first, however, need to install `ray`.

```
!pip install ray==0.8.7
!pip install symfit
```

We will begin by defining the two instruments we want to define our portfolio with. We will be using the U.S. dollar and fake coin called TensorTrade Coin.

```
from tensortrade.oms.instruments import Instrument

USD = Instrument("USD", 2, "U.S. Dollar")
TTC = Instrument("TTC", 8, "TensorTrade Coin")
```

Now let us look at the curve we will be using to define our price.



Ideally, what we will be expecting from our agent is that it will be able to sell at the peaks and buy at the troughs. We will move on to defining our action scheme. The `ActionScheme` we are going to build is going to be extremely simple. There are only 3 states that our agent can be in which are `buy`, `sell`, and `hold`. We will make use of a new function in the library, `proportion_order`. This function enables the user to make an order that can take a percentage of funds at a particular wallet and send it to another. Therefore, we want to structure a way to only have two actions in our scheme and use them as indicators to move our funds to the opposite wallet.

```
from gym.spaces import Discrete

from tensortrade.env.default.actions import TensorTradeActionScheme

from tensortrade.env.generic import ActionScheme, TradingEnv
from tensortrade.core import Clock
from tensortrade.oms.instruments import ExchangePair
from tensortrade.oms.wallets import Portfolio
from tensortrade.oms.orders import (
    Order,
    proportion_order,
    TradeSide,
    TradeType
)


class BSH(TensorTradeActionScheme):

    registered_name = "bsh"

    def __init__(self, cash: 'Wallet', asset: 'Wallet'):
        super().__init__()
        self.cash = cash
        self.asset = asset
```

(continues on next page)

(continued from previous page)

```
        self.listeners = []
        self.action = 0

    @property
    def action_space(self):
        return Discrete(2)

    def attach(self, listener):
        self.listeners += [listener]
        return self

    def get_orders(self, action: int, portfolio: 'Portfolio'):
        order = None

        if abs(action - self.action) > 0:
            src = self.cash if self.action == 0 else self.asset
            tgt = self.asset if self.action == 0 else self.cash
            order = proportion_order(portfolio, src, tgt, 1.0)
            self.action = action

        for listener in self.listeners:
            listener.on_action(action)

        return [order]

    def reset(self):
        super().reset()
        self.action = 0
```

Next, we want to create our reward scheme to reflect how well we are positioned in the environment. Essentially, we want to make a mapping that shows how we would like our rewards to be reflected for each state we are in. The following shows such a mapping:

The signs in the table show what we would like the sign of the rewards to be. The position-based reward scheme (PBR) achieves this mapping.

```
from tensortrade.env.default.rewards import TensorTradeRewardScheme
from tensortrade.feed.core import Stream, DataFeed


class PBR(TensorTradeRewardScheme):

    registered_name = "pbr"

    def __init__(self, price: 'Stream'):
        super().__init__()
        self.position = -1

        r = Stream.sensor(price, lambda p: p.value, dtype="float").diff()
        position = Stream.sensor(self, lambda rs: rs.position, dtype="float")

        reward = (r * position).fillna(0).rename("reward")

        self.feed = DataFeed([reward])
        self.feed.compile()
```

(continues on next page)

(continued from previous page)

```
    def on_action(self, action: int):
        self.position = -1 if action == 0 else 1

    def get_reward(self, portfolio: 'Portfolio'):
        return self.feed.next()["reward"]

    def reset(self):
        self.position = -1
        self.feed.reset()
```

Finally, we would like to make sure we can see if the agent is selling at the peaks and buying at the troughs. We will make a quick `Renderer` that can show this information using `Matplotlib`.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from tensortrade.env.generic import Renderer


class PositionChangeChart(Renderer):

    def __init__(self, color: str = "orange"):
        self.color = "orange"

    def render(self, env, **kwargs):
        history = pd.DataFrame(env.observer.renderer_history)

        actions = list(history.action)
        p = list(history.price)

        buy = {}
        sell = {}

        for i in range(len(actions) - 1):
            a1 = actions[i]
            a2 = actions[i + 1]

            if a1 != a2:
                if a1 == 0 and a2 == 1:
                    buy[i] = p[i]
                else:
                    sell[i] = p[i]

        buy = pd.Series(buy)
        sell = pd.Series(sell)

        fig, axs = plt.subplots(1, 2, figsize=(15, 5))

        fig.suptitle("Performance")

        axs[0].plot(np.arange(len(p)), p, label="price", color=self.color)
        axs[0].scatter(buy.index, buy.values, marker="^", color="green")
        axs[0].scatter(sell.index, sell.values, marker="^", color="red")
        axs[0].set_title("Trading Chart")
```

(continues on next page)

(continued from previous page)

```
        performance_df = pd.DataFrame().from_dict(env.action_scheme.portfolio.
↪performance, orient='index')
        performance_df.plot(ax=axs[1])
        axs[1].set_title("Net Worth")

        plt.show()
```

Train Now in order to use our custom environment in `ray` we must first write a function that creates an instance of the `TradingEnv` from a configuration dictionary.

```
import ray
import numpy as np
import pandas as pd

from ray import tune
from ray.tune.registry import register_env

import tensortrade.env.default as default

from tensortrade.feed.core import DataFeed, Stream
from tensortrade.oms.exchanges import Exchange
from tensortrade.oms.services.execution.simulated import execute_order
from tensortrade.oms.wallets import Wallet, Portfolio



def create_env(config):
    x = np.arange(0, 2*np.pi, 2*np.pi / 1001)
    y = 50*np.sin(3*x) + 100

    x = np.arange(0, 2*np.pi, 2*np.pi / 1000)
    p = Stream.source(y, dtype="float").rename("USD-TTC")

    bitfinex = Exchange("bitfinex", service=execute_order)(
        p
    )

    cash = Wallet(bitfinex, 100000 * USD)
    asset = Wallet(bitfinex, 0 * TTC)

    portfolio = Portfolio(USD, [
        cash,
        asset
    ])

    feed = DataFeed([
        p,
        p.rolling(window=10).mean().rename("fast"),
        p.rolling(window=50).mean().rename("medium"),
        p.rolling(window=100).mean().rename("slow"),
        p.log().diff().fillna(0).rename("lr")
    ])

    reward_scheme = PBR(price=p)

    action_scheme = BSH(
```

(continues on next page)

(continued from previous page)

```
        cash=cash,
        asset=asset
    ).attach(reward_scheme)

    renderer_feed = DataFeed([
        Stream.source(y, dtype="float").rename("price"),
        Stream.sensor(action_scheme, lambda s: s.action, dtype="float").rename("action
↪")
    ])

    environment = default.create(
        feed=feed,
        portfolio=portfolio,
        action_scheme=action_scheme,
        reward_scheme=reward_scheme,
        renderer_feed=renderer_feed,
        renderer=PositionChangeChart(),
        window_size=config["window_size"],
        max_allowed_loss=0.6
    )
    return environment

register_env("TradingEnv", create_env)
```

Now that the environment is registered we can run the training algorithm using the Proximal Policy Optimization (PPO) algorithm implemented in `rllib`.

```
analysis = tune.run(
    "PPO",
    stop={
      "episode_reward_mean": 500
    },
    config={
        "env": "TradingEnv",
        "env_config": {
            "window_size": 25
        },
        "log_level": "DEBUG",
        "framework": "torch",
        "ignore_worker_failures": True,
        "num_workers": 1,
        "num_gpus": 0,
        "clip_rewards": True,
        "lr": 8e-6,
        "lr_schedule": [
            [0, 1e-1],
            [int(1e2), 1e-2],
            [int(1e3), 1e-3],
            [int(1e4), 1e-4],
            [int(1e5), 1e-5],
            [int(1e6), 1e-6],
            [int(1e7), 1e-7]
        ],
        "gamma": 0,
        "observation_filter": "MeanStdFilter",
        "lambda": 0.72,
```

(continues on next page)

(continued from previous page)

```
        "vf_loss_coeff": 0.5,
        "entropy_coeff": 0.01
    },
    checkpoint_at_end=True
)
```

After training is complete, we would now like to get access to the agents policy. We can do that by restoring the agent using the following code.

```
import ray.rllib.agents.ppo as ppo

# Get checkpoint
checkpoints = analysis.get_trial_checkpoints_paths(
    trial=analysis.get_best_trial("episode_reward_mean"),
    metric="episode_reward_mean",
    mode='max'
)
checkpoint_path = checkpoints[0][0]

# Restore agent
agent = ppo.PPOTrainer(
    env="TradingEnv",
    config={
        "env_config": {
            "window_size": 25
        },
        "framework": "torch",
        "log_level": "DEBUG",
        "ignore_worker_failures": True,
        "num_workers": 1,
        "num_gpus": 0,
        "clip_rewards": True,
        "lr": 8e-6,
        "lr_schedule": [
            [0, 1e-1],
            [int(1e2), 1e-2],
            [int(1e3), 1e-3],
            [int(1e4), 1e-4],
            [int(1e5), 1e-5],
            [int(1e6), 1e-6],
            [int(1e7), 1e-7]
        ],
        "gamma": 0,
        "observation_filter": "MeanStdFilter",
        "lambda": 0.72,
        "vf_loss_coeff": 0.5,
        "entropy_coeff": 0.01
    }
)
agent.restore(checkpoint_path)
```

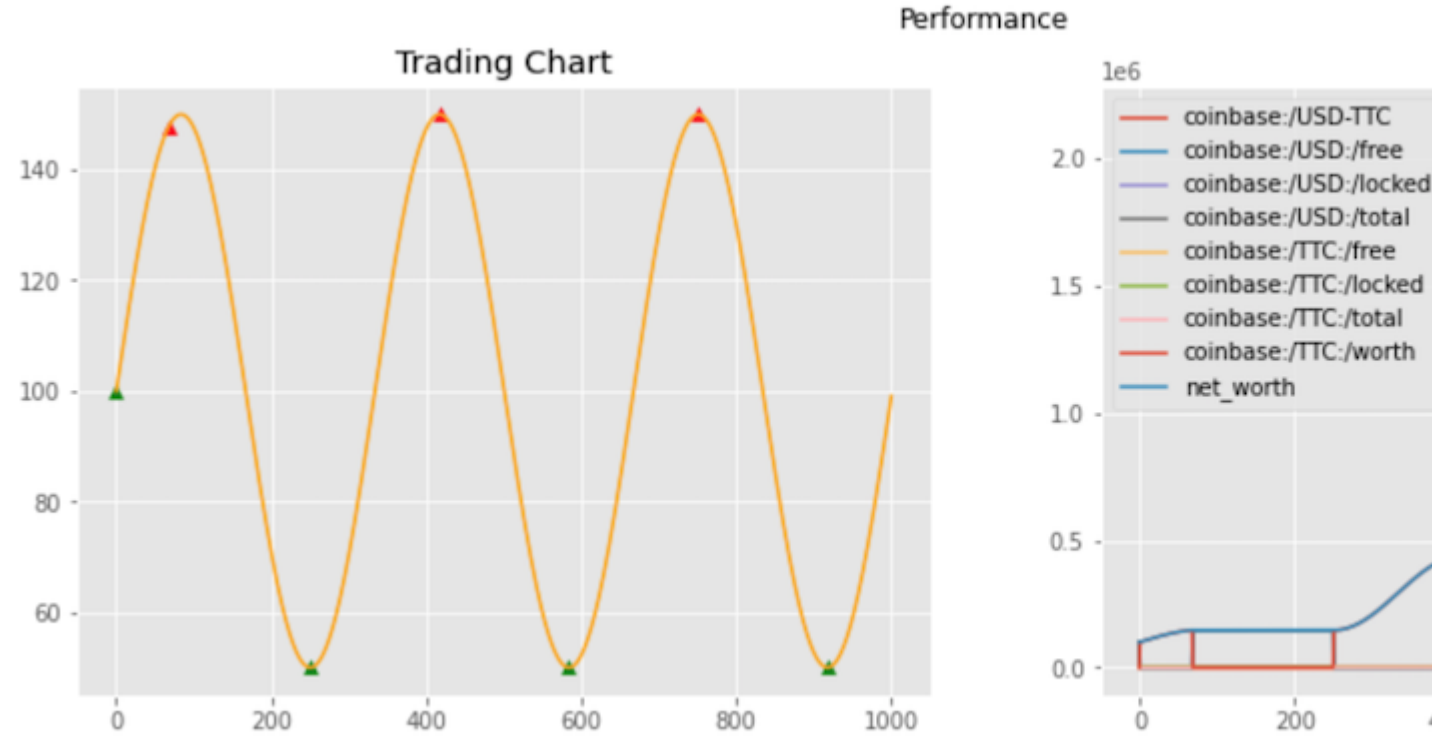
Now let us get a visualization of the agent's decision making on our sine curve example by rendering the environment.

```
# Instantiate the environment
env = create_env({
    "window_size": 25
```

(continues on next page)

(continued from previous page)

```
})

# Run until episode ends
episode_reward = 0
done = False
obs = env.reset()

while not done:
    action = agent.compute_action(obs)
    obs, reward, done, info = env.step(action)
    episode_reward += reward

env.render()
```



From the rendering, you can see that the agent is making near optimal decisions on the environment. Now we want to put the agent in an environment it is not used to and see what kind of decisions it makes. We will use for our price, an `order` 5 Fourier series fitted to a randomly generated Geometric Brownian Motion (GBM). We will use the `symfit` library to do this.

```
from symfit import parameters, variables, sin, cos, Fit


def fourier_series(x, f, n=0):
    """Creates a symbolic fourier series of order `n`.

    Parameters
    ----------
    x : `symfit.Variable`
```

(continues on next page)

(continued from previous page)

```
        The input variable for the function.
    f : `symfit.Parameter`
        Frequency of the fourier series
    n : int
        Order of the fourier series.
    """
    # Make the parameter objects for all the terms
    a0, *cos_a = parameters(','.join(['a{}'.format(i) for i in range(0, n + 1)]))
    sin_b = parameters(','.join(['b{}'.format(i) for i in range(1, n + 1)]))

    # Construct the series
    series = a0 + sum(ai * cos(i * f * x) + bi * sin(i * f * x)
                      for i, (ai, bi) in enumerate(zip(cos_a, sin_b), start=1))
    return series


def gbm(price: float,
        mu: float,
        sigma: float,
        dt: float,
        n: int) -> np.array:
    """Generates a geometric brownian motion path.

    Parameters
    ----------
    price : float
        The initial price of the series.
    mu : float
        The percentage drift.
    sigma : float
        The percentage volatility.
    dt : float
        The time step size.
    n : int
        The number of steps to be generated in the path.

    Returns
    -------
    `np.array`
        The generated path.
    """
    y = np.exp((mu - sigma ** 2 / 2) * dt + sigma * np.random.normal(0, np.sqrt(dt),␣
↪size=n).T)
    y = price * y.cumprod(axis=0)
    return y


def fourier_gbm(price, mu, sigma, dt, n, order):

    x, y = variables('x, y')
    w, = parameters('w')
    model_dict = {y: fourier_series(x, f=w, n=order)}

    # Make step function data
    xdata = np.arange(-np.pi, np.pi, 2*np.pi / n)
    ydata = np.log(gbm(price, mu, sigma, dt, n))
```

(continues on next page)

(continued from previous page)

```
    # Define a Fit object for this model and data
    fit = Fit(model_dict, x=xdata, y=ydata)
    fit_result = fit.execute()

    return np.exp(fit.model(x=xdata, **fit_result.params).y)
```

Now we can make the evaluation environment and see how the agent performs.

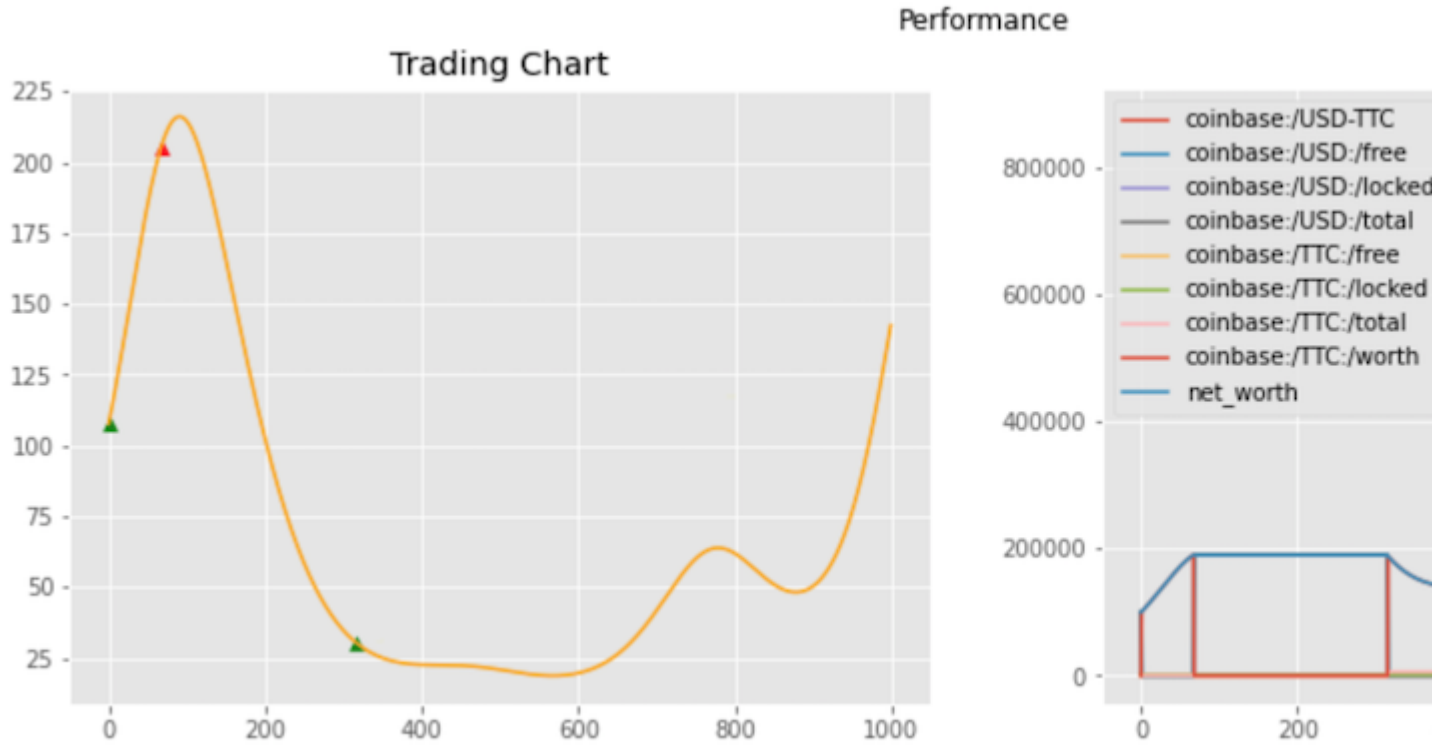
```
def create_eval_env(config):
    y = config["y"]

    x = np.arange(0, 2*np.pi, 2*np.pi / 1000)
    p = Stream.source(y, dtype="float").rename("USD-TTC")

    bitfinex = Exchange("bitfinex", service=execute_order)(
        p
    )

    cash = Wallet(bitfinex, 100000 * USD)
    asset = Wallet(bitfinex, 0 * TTC)

    portfolio = Portfolio(USD, [
        cash,
        asset
    ])

    feed = DataFeed([
        p,
        p.rolling(window=10).mean().rename("fast"),
        p.rolling(window=50).mean().rename("medium"),
        p.rolling(window=100).mean().rename("slow"),
        p.log().diff().fillna(0).rename("lr")
    ])

    reward_scheme = PBR(price=p)

    action_scheme = BSH(
        cash=cash,
        asset=asset
    ).attach(reward_scheme)

    renderer_feed = DataFeed([
        Stream.source(y, dtype="float").rename("price"),
        Stream.sensor(action_scheme, lambda s: s.action, dtype="float").rename("action
↪")
    ])

    environment = default.create(
        feed=feed,
        portfolio=portfolio,
        action_scheme=action_scheme,
        reward_scheme=reward_scheme,
        renderer_feed=renderer_feed,
        renderer=PositionChangeChart(),
        window_size=config["window_size"],
        max_allowed_loss=0.6
    )
```

(continues on next page)

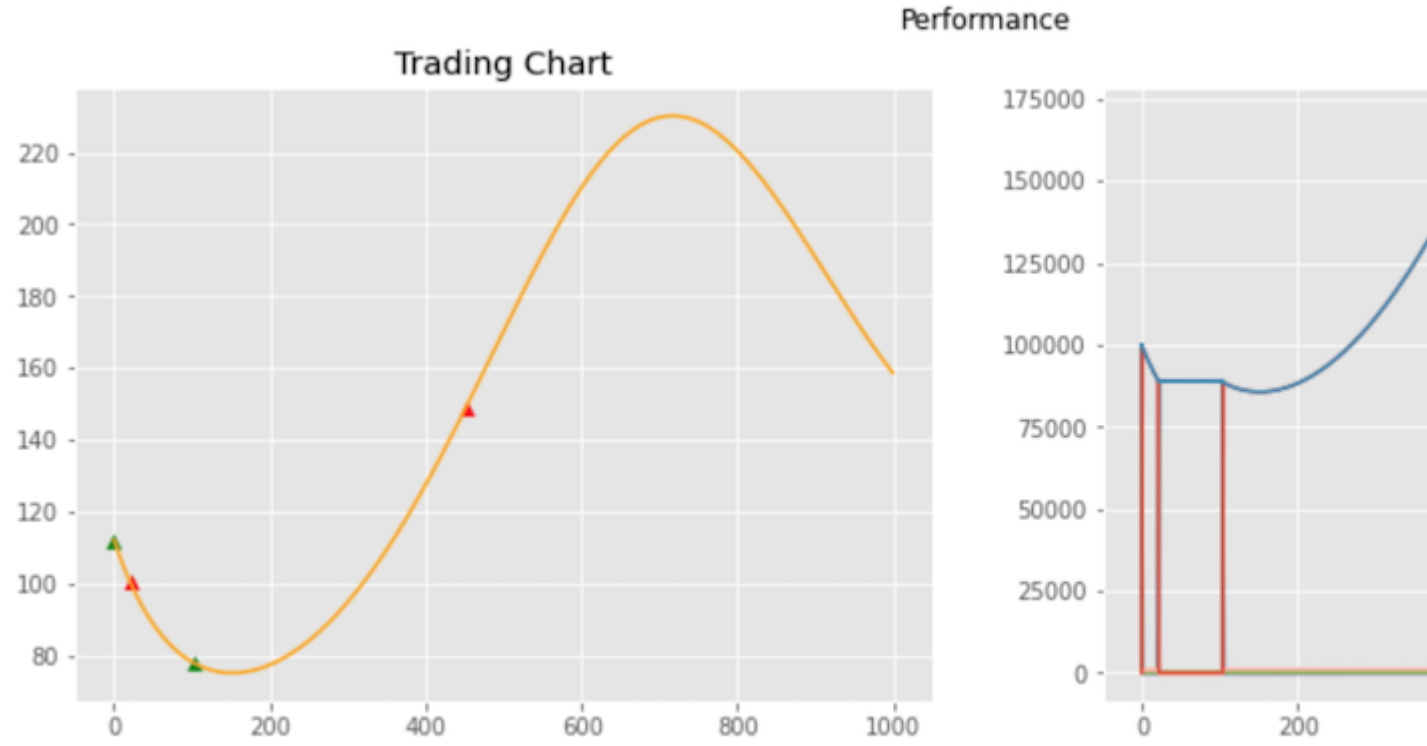
(continued from previous page)

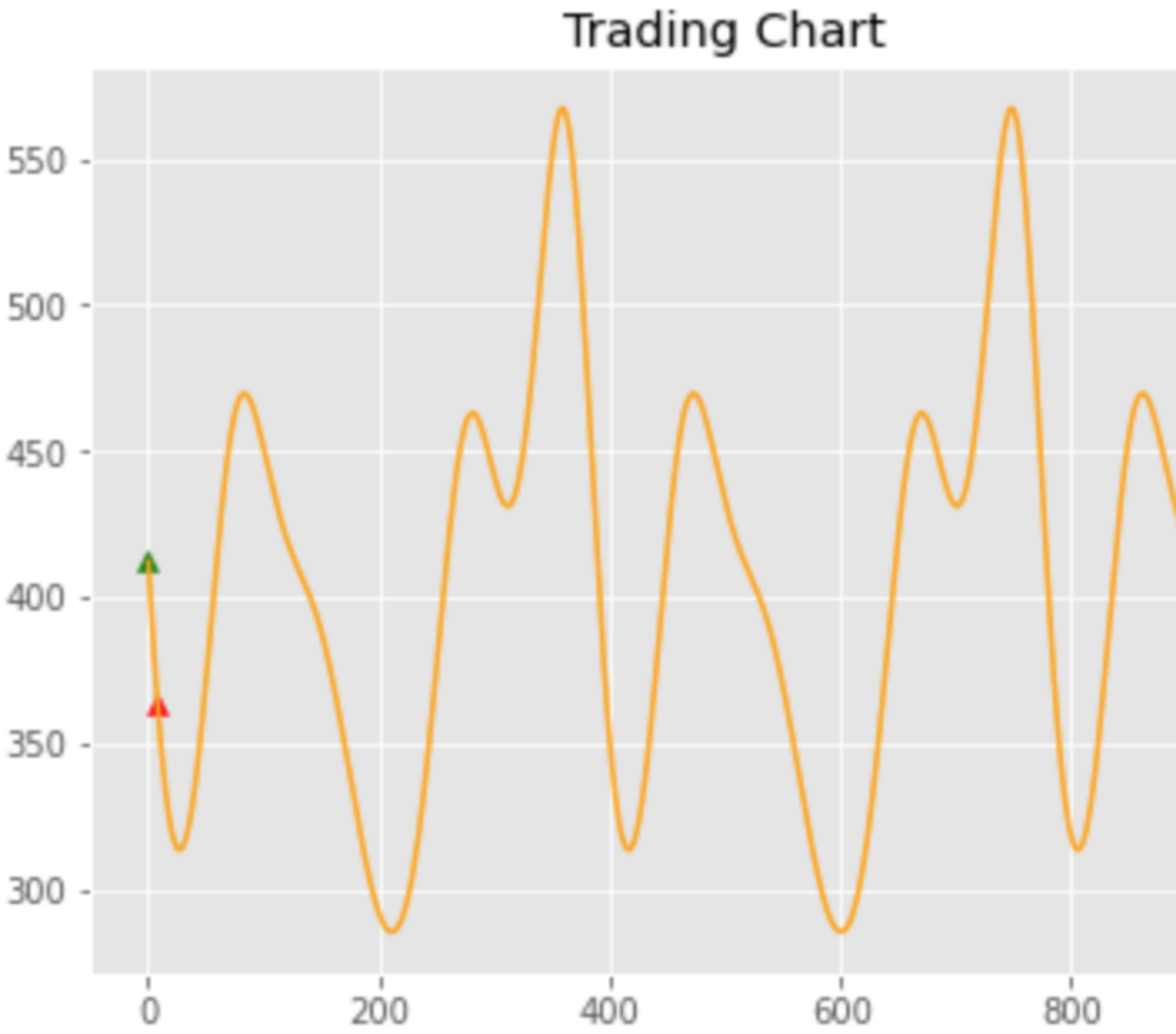
```
    return environment

# Instantiate the environment
env = create_eval_env({
    "window_size": 25,
    "y": fourier_gbm(price=100, mu=0.01, sigma=0.5, dt=0.01, n=1000, order=5)
})


# Run until episode ends
episode_reward = 0
done = False
obs = env.reset()

while not done:
    action = agent.compute_action(obs)
    obs, reward, done, info = env.step(action)
    episode_reward += reward
```

The following are a few examples of the rendering that came from the evaluation environment.




Trading Chart
Performance
coinbase:/USD-TTC
coinbase:/USD:/free
coinbase:/USD:/locked
coinbase:/USD:/total
coinbase:/TTC:/free
coinbase:/TTC:/locked
coinbase:/TTC:/total
coinbase:/TTC:/worth
net_worth


Trading Chart
Performance



As you can see, the agent has been able to make correct decisions on some of the price curves, but not all of them. The last curve shows some of the shortcomings of the agent. The agent seemed to have stop making decisions to move its wealth to capitalize on the changes in price. In the first three, however, it was able to make those decisions. The reason for this is most likely the change in the frequency of the curve. The last chart has a price curve that is volatile, containing many local maxima and minima as opposed to the first three charts.

What have we learned?

- Make an environment in TT.

- Create a custom `ActionScheme`, `RewardScheme`, and `Renderer` in TT.
- Using a simple price curve to understand more about actions and rewards in our use case.
- Using `ray` to train and restore an agent.
- Making evaluation environments for providing insight into the decisions of an agent.

See you in the next tutorial!

1.11 Components

TensorTrade is built around modular components that together make up a trading strategy. Trading strategies combine reinforcement learning agents with composable trading logic in the form of a `gym` environment. A trading environment is made up of a set of modular components that can be mixed and matched to create highly diverse trading and investment strategies.

Just like electrical components, the purpose of TensorTrade components is to be able to mix and match them as necessary.

1.12 Action Scheme

An `ActionScheme` defines the action space of the environment and interprets the action of an agent and how it gets applied to the environment.

For example, if we were using a discrete action space of 3 actions (0 = hold, 1 = buy 100 %, 2 = sell 100%), our learning agent does not need to know that returning an action of 1 is equivalent to buying an instrument. Rather, the agent needs to know the reward for returning an action of 1 in specific circumstances, and can leave the implementation details of converting actions to trades to the `ActionScheme`.

Each action scheme has a `perform` method, which will interpret the agent's specified action into a change in the environmental state. It is often necessary to store additional state within the scheme, for example to keep track of the currently traded position. This state should be reset each time the action scheme's reset method is called, which is done automatically when the parent `TradingEnv` is reset.

1.12.1 What is an Action?

This is a review of what was mentioned inside of the overview section and explains how a RL operates. You'll better understand what an action is in context of an observation space and reward. At the same time, hopefully this will be a proper refresher.

An action is a predefined value of how the machine should move inside of the world. To better summarize, its a *command that a player would give inside of a video game in respose to a stimuli*. The commands usually come in the form of an `action_space`. An `action_space` defines the rules for how a user is allowed to act inside of an environment. While it might not be easily interpretable by humans, it can easily be interpreted by a machine.

Let's look at a good example. Let's say we're trying to balance a cart with a pole on it (cartpole). We can choose to move the cart left and right. This is a `Discrete(2)` action space.

- 0 - Push cart to the left
- 1 - Push cart to the right

When we get the action from the agent, the environment will see that number instead of a name.

Watch Link Run Around In Circles

An `ActionScheme` supports any type of action space that subclasses `Space` from `gym`. For example, here is an implementation of an action space that represents a probability simplex.

```
import numpy as np

from gym.spaces import Space

class Simplex(Space):

    def __init__(self, k: int) -> None:
        assert k >= 2
        super().__init__(shape=(k, ), dtype=np.float32)
        self.k = k

    def sample(self) -> float:
        return np.random.dirichlet(alpha=self.k*[3*np.random.random()])

    def contains(self, x) -> bool:
        if len(x) != self.k:
            return False
        if sum(x) != 1.0:
            return False
        return True
```

1.12.2 Default

The default TensorTrade action scheme is made to be compatible with the built-in order management system (OMS). The OMS is a system that is able to have orders be submitted to it for particular financial instruments.

Simple

Overview A discrete action scheme that determines actions based on a list of trading pairs, order criteria, and trade sizes.

Action Space The action space is a discrete set of N options. The total number of discrete actions is the product of

- criteria (order criteria for order creation/completion)
- trade sizes (e.g. 1/4, 1/2, 1/3)
- trade durations (e.g. order open for 30 seconds or 60 seconds)
- trade sides (i.e. Buy or Sell)
- the number of tradeable pairs (i.e. BTC/USDT, ETH/BTC, etc)

Perform Performs as per TensorTradeActionScheme, creates order based on models discrete output and submits it to the broker. The state action mapping varies with the parameters above.

ManagedRisk

Overview A discrete action scheme that determines actions based on managing risk, through setting a follow-up stop loss and take profit on every order.

Action Space The action space is a discrete set of N options for the model to take. The total number of discrete actions is determined by taking a product of:

- stop percents (i.e. [0.02, 0.04, 0.06] percent changes to trigger a stop loss)

- take percents (i.e. [0.02, 0.03] value percent changes to take profit at)
- trade sizes (e.g. 1/4, 1/2, 1/3)
- trade durations (e.g. order open for 30 seconds or 60 seconds)
- trade sides (i.e. Buy or Sell)
- the number of tradable pairs (i.e. BTC/USDT, ETH/BTC, etc)

Perform Performs as per TensorTradeActionScheme, creates order based on models discrete output and submits it to the broker. The state action mapping varies with the parameters above.

BSH

Overview The buy/sell/hold (BSH) action scheme was made to capture the simplest type of action space that can be made. If the agent is in state 0, then all of its net worth is located in our `cash` wallet (e.g. USD). If the agent is in state 1, then all of its net worth is located in our `asset` wallet (e.g. BTC).

Action Space

- `Discrete(2)` options, buy or sell

Perform Below is a table that shows the mapping `(state, action) -> (state)`.

1.13 Reward Scheme

Reward schemes receive the `TradingEnv` at each time step and return a `float`, corresponding to the benefit of that specific action. For example, if the action taken this step was a sell that resulted in positive profits, our `RewardScheme` could return a positive number to encourage more trades like this. On the other hand, if the action was a sell that resulted in a loss, the scheme could return a negative reward to teach the agent not to make similar actions in the future.

TensorTrade currently implements `SimpleProfit`, `RiskAdjustedReturns` and `PBR` (position-based returns), however more complex schemes can obviously be used instead.

Each reward scheme has a `reward` method, which takes in the `TradingEnv` at each time step and returns a `float` corresponding to the value of that action. As with action schemes, it is often necessary to store additional state within a reward scheme for various reasons. This state should be reset each time the reward scheme's reset method is called, which is done automatically when the environment is reset.

Ultimately the agent creates a sequence of actions to maximize its total reward over a given time. The `RewardScheme` is an abstract class that encapsulates how to tell the trading bot in `tensortrade` if it's trading positively or negatively over time. The same methods will be called each time for each step, and we can directly swap out compatible schemes.

```
from tensortrade.env.default.rewards import SimpleProfit

reward_scheme = SimpleProfit()
```

1.13.1 Simple Profit

A reward scheme that simply rewards the agent for profitable trades, no matter how it got there.

Overview

The simple profit scheme needs to keep a history of profit over time. The way it does this is through looking at the portfolio as a means of keeping track of how the portfolio moves. This is seen inside of the `get_reward` function.

Computing Reward

Reward is calculated as the cumulative percentage change in net worth over the previous `window_size` time steps

1.13.2 Risk Adjusted Returns

A reward scheme that rewards the agent for increasing its net worth, while penalizing more volatile strategies.

Overview

When trading you often are not just looking at the overall returns of your model. You're also looking at the overall volatility of your trading strategy over time compared to other metrics. The two major strategies here are the sharpe and sortino ratio.

The **sharpe ratio** looks at the overall movements of the portfolio and generates a penalty for massive movements through a lower score. This includes major movements towards the upside and downside.

$$S = \left(\frac{R_p - R_f}{\sigma_p} \right)$$

Sharpe Ratio

The **sortino ratio** takes the same idea, though it focuses more on penalizing only the upside. That means it'll give a huge score for moments when the price moves upward, and will only give a negative score when the price drops heavily. This is a great direction for the RL algorithm. Seeing that we don't want to incur heavy downsides, yet want to take on large upsides, using this metric alone gives us lots of progress to mititgate downsides and increase upsides.

$$S = \frac{(R - MAR)}{\text{downside deviation}}$$

Sortino Ratio

Computing Reward

Given the choice of `return_algorithm` the reward is computed using the `risk_free_rate` and the `target_returns` parameters.

1.13.3 Position-based returns (PBR)

Documentation is missing. Please submit a pull request and help us expand it!

1.14 Observer

The `Observer` component is what the environment uses to represent the way the agent sees the environment. The way it is built in the `default` environment is with the help of the `DataFeed`. There were many reasons for this decision, but the most important ones are that we would like to be able to compute path-dependent observations in a reliable way. We would also like to minimize lookahead biases and we would also like it to translate well into the live setting as well. To leave the opportunities broad, however, all you need to do is fill out the `observe` method to determine what your agent sees next.

1.15 Default

The `default` observer at the moment uses a structure of a look-back window in order to create an observation. After the `feed` is made the environment will use a window of past observations and will have the shape `(window_size, n_features)`.

1.16 IntradayObserver

The `IntradayObserver` divides the DataFeed into episodes of single trading days. It takes the same parameters as the `default` observer and additional parameters of `stop_time` of type `datetime.time` and `randomize` of type `bool`. `stop_time` is the `datetime.time` of the `timestamp` in the `DataFrame` that marks the end of the episode. It is imperative to ensure that each trading day includes the respective `timestamp`. `randomize` determines if the episode, or trading day, should be selected randomly or in the sequence of the `DataFeed`. The `IntradayObserver` requires that the `DataFeed` include a `Stream` of timestamps named `'timestamp'`.

1.17 Stopper

The `Stopper` component just evaluates whether or not the environment is done running after each `step` in the environment. For example, right now the `default` environment just evaluate if the environment is done based on the profit loss of the agent. If the profit loss drops below a certain level the environment will be done. In addition, if the `feed` runs out of data to give the environment will also be done for that episode.

1.18 Informer

The `Informer` component just delivers contextual environment information after each `step` in the environment. For example, right now the `default` environment just delivers the `portfolio`, `broker`, and `net_worth` in the step functions.

1.19 Renderer

The `Renderer` is the component of the `TradingEnv` that lets us peek into the thoughts and actions of an agent. There are many different ways to render this type of environment, but all you need to do in order to get a different renderering is to subclass `Renderer` and fill in the `render` method. If you want to see more on how to make a `Renderer` from scratch check out the tutorial for using `ray` with TensorTrade.

1.19.1 Default

The `default` renderers are made to work with any instance of the `BaseRenderer` class.

Screen Logger

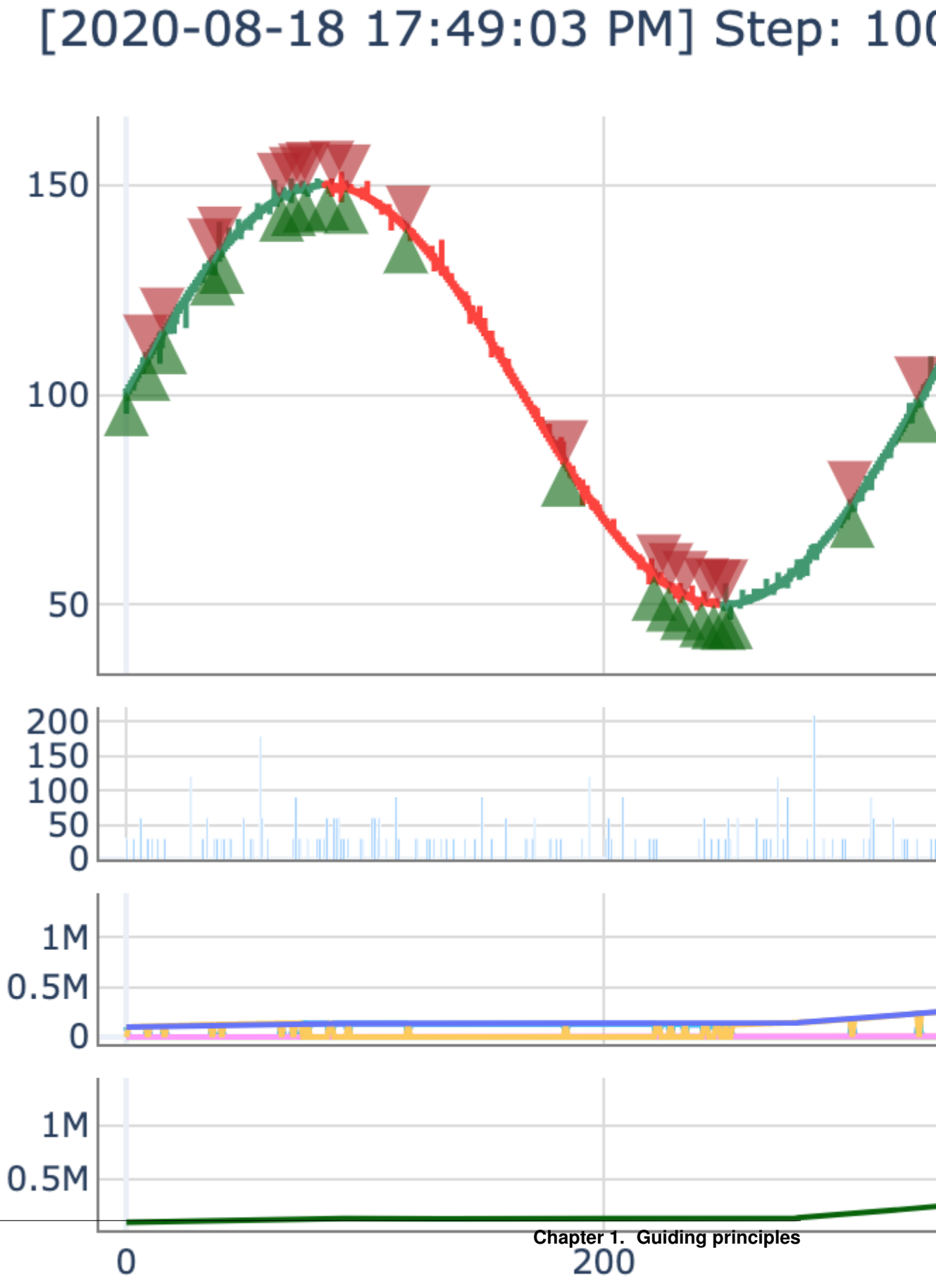
Logs the records of the environment to the screen for the user to see.

File Logger

Logs the records of the environment to a specified file.

Plotly Trading Chart

Uses plotly to make a rendering of the environment. This renderer only works for when the action scheme involves trading one asset. The data required for the `render_feed` must include the date, open, high, low, close, and volume of the instrument being traded.


[2020-08-18 17:49:03 PM] Step: 100
150
100
50
200
150
100
50
0
1M
0.5M
0
1M
0.5M
0
200

References

- https://plot.ly/python-api-reference/generated/plotly.graph_objects.Figure.html
- https://plot.ly/python/figurewidget/
- https://plot.ly/python/subplots/
- https://plot.ly/python/reference/#candlestick
- https://plot.ly/python/#chart-events

1.20 Overview

A trading environment is a reinforcement learning environment that follows OpenAI's `gym.Env` specification. This allows us to leverage many of the existing reinforcement learning models in our trading agent, if we'd like.

`TradingEnv` steps through the various interfaces from the `tensortrade` library in a consistent way, and will likely not change too often as all other parts of `tensortrade` changes. We're going to go through an overview of the Trading environment below.

Trading environments are fully configurable gym environments with highly composable components:

- The `ActionScheme` interprets and applies the agent's actions to the environment.
- The `RewardScheme` computes the reward for each time step based on the agent's performance.
- The `Observer` generates the next observation for the agent.
- The `Stopper` determines whether or not the episode is over.
- The `Informer` generates useful monitoring information at each time step.
- The `Renderer` renders a view of the environment and interactions.

That's all there is to it, now it's just a matter of composing each of these components into a complete environment.

When the reset method of a `TradingEnv` is called, all of the child components will also be reset. The internal state of each action scheme, reward scheme, observer, stopper, and informer will be set back to their default values, ready for the next episode.

1.21 What if I can't make a particular environment?

If none of the environments available in codebase serve your needs let us know! We would love to hear about so we can keep improving the quality of our framework as well as keeping up with the needs of the people using it.

1.22 Overview

The `feed` package provides useful tools when building trading environments. The primary reason for using this package is to help build the mechanisms that generate observations from an environment. Therefore, it is fitting that their primary location of use is in the `Observer` component. The `Stream` API provides the granularity needed to connect specific data sources to the `Observer`.

1.23 What is a `Stream`?

A `Stream` is the basic building block for the `DataFeed`, which is also a stream itself. Each stream has a name and a data type and they can be set after the stream is created. Streams can be created through the following mechanisms:

- generators
- iterables
- sensors
- direct implementation of `Stream`

For example, if you wanted to make a stream for a simple counter. We will make it such that it will start at 0 and increment by 1 each time it is called and on `reset` will set the count back to 0. The following code accomplishes this functionality through creating a generator function.

```
from tensortrade.feed import Stream

def counter():
    i = 0
    while True:
        yield i
        i += 1

s = Stream.source(counter)
```

In addition, you can also use the built-in `count` generator from the `itertools` package.

```
from itertools import count

s = Stream.source(count(start=0, step=1))
```

These will all create infinite streams that will keep incrementing by 1 to infinity. If you wanted to make something that counted until some finite number you can use the built in `range` function.

```
s = Stream.source(range(5))
```

This can also be done by giving in a `list` directly.

```
s = Stream.source([1, 2, 3, 4, 5])
```

The direct approach to stream creation is by subclassing `Stream` and implementing the `forward`, `has_next`, and `reset` methods. If the stream does not hold stateful information, then `reset` is not required to be implemented and can be ignored.

```
class Counter(Stream):

    def __init__(self):
        super().__init__()
        self.count = None

    def forward(self):
        if self.count is None:
            self.count = 0
        else:
            self.count += 1
        return self.count
```

(continues on next page)

(continued from previous page)

```
    def has_next(self):
        return True

    def reset(self):
        self.count = None

s = Counter()
```

There is also a way of creating streams which serves the purpose of watching a particular object and how it changes over time. This can be done through the `sensor` function. For example, we can use this to directly track performance statistics on our `portfolio`. Here is a specific example of how we can use it to track the number of orders the are currently active inside the order management system.

```
from tensortrade.env.default.actions import SimpleOrders

action_scheme = SimpleOrders()

s = Stream.sensor(action_scheme.broker, lambda b: len(b.unexecuted))
```

As the agent and the environment are interacting with one another, this stream will be able to monitor the number of active orders being handled by the broker. This stream can then be used by either computing performance statistics and supplying them to a `Renderer` or simply by including it within the observation space.

Now that we have seen the different ways we can create streams, we need to understand the ways in which we can aggregate new streams from old. This is where the data type of a stream becomes important.

1.24 Using Data Types

The purpose of the data type of a stream, `dtype`, is to add additional functionality and behavior to a stream such that it can be aggregated with other streams of the same type in an easy and intuitive way. For example, what if the number of executed orders from the `broker` is not important by itself, but is important with respect to the current time of the process. This can be taken into account if we create a stream for keeping count of the active orders and another one for keeping track of the step in the process. Here is what that would look like.

```
from itertools import count

from tensortrade.feed import Stream
from tensortrade.env.default.actions import SimpleOrders

n = Stream.source(count(0, step=1), dtype="float")
n_active = Stream.sensor(action_scheme.broker, lambda b: len(b.unexecuted), dtype=
↪"float")

s = (n_active / (n + 1)).rename("avg_n_active")
```

Suppose we find that this is not a useful statistic and instead would like to know how many of the active order have been filled since the last time step. This can be done by using the `lag` operator on our stream and finding the difference between the current count and the count from the last time step.

```
n_active = Stream.sensor(action_scheme.broker, lambda b: len(b.unexecuted), dtype=
↪"float")

s = (n_active - n_active.lag()).rename("n_filled")
```

As you can see from the code above, we were able to make more complex streams by using simple ones. Take note, however, in the way we use the `rename` function. We only really want to rename a stream if we will be using it somewhere else where its name will be useful (e.g. in our `feed`). We do not want to name all the intermediate streams that are used to build our final statistic because the code will become too cumbersome and annoying. To avoid these complications, streams are created to automatically generate a unique name on instantiation. We leave the naming for the user to decide which streams are useful to name.

Since the most common data type is `float` in these tasks, the following is a list of supported special operations for it:

- Let `s`, `s1`, `s2` be streams.
- Let `c` be a constant.
- Let `n` be a number.
- Unary:
  - `-s`, `s.neg()`
  - `abs(s)`, `s.abs()`
  - `s**2`, `pow(s, n)`
- Binary:
  - `s1 + s2`, `s1.add(s2)`, `s + c`, `c + s`
  - `s1 - s2`, `s1.sub(s2)`, `s - c`, `c - s`
  - `s1 * s2`, `s1.mul(s2)`, `s * c`, `c * s`
  - `s1 / s2`, `s1.div(s2)`, `s / c`, `c / s`

There are many more useful functions that can be utilized, too many to list in fact. You can find all of the. however, in the API reference section of the documentation.

1.25 Advanced Usages

The `Stream` API is very robust and can handle complex streaming operations, particularly for the `float` data type. Some of the more advanced usages include performance tracking and developing reward schemes for the `default` trading environment. In the following example, we will show how to track the net worth of a portfolio. This implementation will be coming directly from the wallets that are defined in the `portfolio`.

```
# Suppose we have an already constructed portfolio object, `portfolio`.

worth_streams = []

for wallet in portfolio.wallets:

    total_balance = Stream.sensor(
        wallet,
        lambda w: w.total_balance.as_float(),
        dtype="float"
    )

    symbol = w.instrument.symbol

    if symbol == portfolio.base_instrument.symbol
        worth_streams += [total_balance]
    else:
```

(continues on next page)

(continued from previous page)

```
        price = Stream.select(
            w.exchange.streams(),
            lambda s: s.name.endswith(symbol)
        )
        worth_streams += [(price * total_balance)]

net_worth = Stream.reduce(worth_streams).sum().rename("net_worth")
```

1.26 Overview

An `order management system (OMS)` is a system that controls how an order for a specific financial instrument is filled. In building an OMS, you have to make clear what the lifecycle of an order is. The OMS we use in the default environment for TensorTrade is a first attempt at building such a system. The goal of our system, however, was meant to serve the purpose simulating a real order management system. We created it with the user in mind, hoping to give maximum customization the types of order placed.

1.27 Portfolio

A portfolio is a collection of financial investments like stocks, bonds, commodities, cash, and cash equivalents, including closed-end funds and exchange-traded funds (ETFs). Now a portfolio can include more than just these types of assets. For example, real-estate is an investment that can be considered to be part of a portfolio. For the purposes of algorithmic trading, however, a more formal definition of a portfolio is needed. In order for that to be done we first need to define what a financial instrument is and how it fits in with the idea of a portfolio.

Instruments The core idea surrounding a financial instrument is the idea of tradability. In our universe, an object is said to be tradable if and only if can be traded on an exchange. This definition makes the idea more concrete and solves some ambiguity that can arise when multiple exchanges are involved. For example, Bitcoin (BTC) is an asset that you can hold an amount of, however, its value depends on the exchange that you are trading it on. Therefore, given an `Exchange` we need to be able hold a `Quantity` of an `Instrument`. Now enters the idea for a `Wallet`.

A `Quantity` of an instrument can be created with the following code

```
from tensortrade.oms.instruments import Quantity, USD, BTC

q = 10000 * USD

q = Quantity(BTC, 10000)
```

Wallets A `Wallet` is specified by an `Exchange` and a `Quantity`. The financial instrument that the wallet holds is implicitly defined in the `Quantity` we are holding in the wallet. The wallet also gives us the ability to transfer funds from a given wallet to another.

```
# Suppose we have already defined an exchange that supports the financial instrument
# we are creating.
from tensortrade.oms.instruments import USD, BTC
from tensortrade.oms.wallets import Wallet

w1 = Wallet(exchange, 10000 * USD)
w2 = Wallet(exchange, 0 * BTC)
```

Creating a Portfolio A `Portfolio` in the library is defined to be a set of wallets. This makes building a portfolio rather simple. All we need to do is create the wallets and pass them in to the `Portfolio` for construction.

```
# Suppose we have already defined two exchange `e1` and `e2`.

from tensortrade.oms.instruments import USD, BTC, ETH, LTC
from tensortrade.oms.wallets import Wallet, Portfolio

portfolio = Portfolio(
    base_instrument=USD,
    wallets=[
        Wallet(e1, 10000 * USD),
        Wallet(e1, 0 * BTC),
        Wallet(e1, 0 * ETH),
        Wallet(e2, 10 * USD),
        Wallet(e2, 0 * ETH),
        Wallet(e2, 0 * LTC),
    ]
)
```

In addition, you also have to specify what the base instrument is so the portfolio knows what every instruments value should be in.

1.28 Orders

An `Order` is the way in which you can move funds from one wallet to another. The supported orders that can be made right now are the following:

- Market
- Limit
- Stop Loss
- Take Profit

Currently all the `default` action schemes use these orders when interpreting agent actions. The stop loss and take profit orders are the most complicated of which and require the use of an `OrderSpec` for them to function properly. An `OrderSpec` is required when an order must be connected with and followed by a successive order. In the case of a stop order, the process is to buy the quantity requested at the current price and then wait until the price hits a particular mark and then sell it. In addition, an `Order` has an optional `criteria` parameter that needs to be satisfied before being able to execute on an exchange.

1.29 Overview

This is where the "deep" part of the deep reinforcement learning framework come in. Learning agents are where the math (read: magic) happens.

At each time step, the agent takes the observation from the environment as input, runs it through its underlying model (a neural network most of the time), and outputs the action to take. For example, the observation might be the previous `open`, `high`, `low`, and `close` price from the exchange. The learning model would take these values as input and output a value corresponding to the action to take, such as `buy`, `sell`, or `hold`.

It is important to remember the learning model has no intuition of the prices or trades being represented by these values. Rather, the model is simply learning which values to output for specific input values or sequences of input values, to earn the highest reward.

In this example, we will be using the Stable Baselines library to provide learning agents to our trading scheme, however, the TensorTrade framework is compatible with many reinforcement learning libraries such as Tensorforce, Ray's RLLib, OpenAI's Baselines, Intel's Coach, or anything from the TensorFlow line such as TF Agents.

It is possible that custom TensorTrade learning agents will be added to this framework in the future, though it will always be a goal of the framework to be interoperable with as many existing reinforcement learning libraries as possible, since there is so much concurrent growth in the space. But for now, Stable Baselines is simple and powerful enough for our needs.

1.30 Ray

The following is an example of how to train a strategy on `ray` using the `PPO` algorithm.

```
import ray
import numpy as np

from ray import tune
from ray.tune.registry import register_env

import tensortrade.env.default as default

from tensortrade.feed.core import DataFeed, Stream
from tensortrade.oms.instruments import Instrument
from tensortrade.oms.exchanges import Exchange
from tensortrade.oms.services.execution.simulated import execute_order
from tensortrade.oms.wallets import Wallet, Portfolio


USD = Instrument("USD", 2, "U.S. Dollar")
TTC = Instrument("TTC", 8, "TensorTrade Coin")


def create_env(config):
    x = np.arange(0, 2*np.pi, 2*np.pi / 1000)
    p = Stream.source(50*np.sin(3*x) + 100, dtype="float").rename("USD-TTC")

    bitfinex = Exchange("bitfinex", service=execute_order)(
        p
    )

    cash = Wallet(bitfinex, 100000 * USD)
    asset = Wallet(bitfinex, 0 * TTC)

    portfolio = Portfolio(USD, [
        cash,
        asset
    ])

    feed = DataFeed([
        p,
        p.rolling(window=10).mean().rename("fast"),
        p.rolling(window=50).mean().rename("medium"),
        p.rolling(window=100).mean().rename("slow"),
        p.log().diff().fillna(0).rename("lr")
    ])
```

(continues on next page)

(continued from previous page)

```
    reward_scheme = default.rewards.PBR(price=p)

    action_scheme = default.actions.BSH(
        cash=cash,
        asset=asset
    ).attach(reward_scheme)

    env = default.create(
        feed=feed,
        portfolio=portfolio,
        action_scheme=action_scheme,
        reward_scheme=reward_scheme,
        window_size=config["window_size"],
        max_allowed_loss=0.6
    )
    return env

register_env("TradingEnv", create_env)


analysis = tune.run(
    "PPO",
    stop={
      "episode_reward_mean": 500
    },
    config={
        "env": "TradingEnv",
        "env_config": {
            "window_size": 25
        },
        "log_level": "DEBUG",
        "framework": "torch",
        "ignore_worker_failures": True,
        "num_workers": 1,
        "num_gpus": 0,
        "clip_rewards": True,
        "lr": 8e-6,
        "lr_schedule": [
            [0, 1e-1],
            [int(1e2), 1e-2],
            [int(1e3), 1e-3],
            [int(1e4), 1e-4],
            [int(1e5), 1e-5],
            [int(1e6), 1e-6],
            [int(1e7), 1e-7]
        ],
        "gamma": 0,
        "observation_filter": "MeanStdFilter",
        "lambda": 0.72,
        "vf_loss_coeff": 0.5,
        "entropy_coeff": 0.01
    },
    checkpoint_at_end=True
)
```

And then to restore the agent just use the following code.

```
import ray.rllib.agents.ppo as ppo

# Get checkpoint
checkpoints = analysis.get_trial_checkpoints_paths(
    trial=analysis.get_best_trial("episode_reward_mean"),
    metric="episode_reward_mean"
)
checkpoint_path = checkpoints[0][0]

# Restore agent
agent = ppo.PPOTrainer(
    env="TradingEnv",
    config={
        "env_config": {
            "window_size": 25
        },
        "framework": "torch",
        "log_level": "DEBUG",
        "ignore_worker_failures": True,
        "num_workers": 1,
        "num_gpus": 0,
        "clip_rewards": True,
        "lr": 8e-6,
        "lr_schedule": [
            [0, 1e-1],
            [int(1e2), 1e-2],
            [int(1e3), 1e-3],
            [int(1e4), 1e-4],
            [int(1e5), 1e-5],
            [int(1e6), 1e-6],
            [int(1e7), 1e-7]
        ],
        "gamma": 0,
        "observation_filter": "MeanStdFilter",
        "lambda": 0.72,
        "vf_loss_coeff": 0.5,
        "entropy_coeff": 0.01
    }
)
agent.restore(checkpoint_path)
```

1.31 Stable Baselines

```
from stable_baselines.common.policies import MlpLnLstmPolicy
from stable_baselines import PPO2

model = PPO2
policy = MlpLnLstmPolicy
params = { "learning_rate": 1e-5 }

agent = model(policy, environment, model_kwargs=params)
```

1.32 Tensorforce

I will also quickly cover the Tensorforce library to show how simple it is to switch between reinforcement learning frameworks.

```
from tensorforce.agents import Agent

agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]

agent = Agent.from_spec(spec=agent_spec,
                        kwargs=dict(network=network_spec,
                                    states=environment.states,
                                    actions=environment.actions))
```

If you would like to know more about Tensorforce agents, you can view the Documentation.

1.33 tensortrade

1.33.1 tensortrade package

Subpackages

tensortrade.agents package

Subpackages

tensortrade.agents.parallel package

Submodules

tensortrade.agents.parallel.parallel_dqn_agent module

class tensortrade.agents.parallel.parallel_dqn_agent.**ParallelDQNAgent**(*create_env: Callable[[None], TradingEnvironment], model: tensortrade.agents.parallel.parallel_* = *None*)

Bases: *tensortrade.agents.agent.Agent*

get_action(*state: numpy.ndarray*, ***kwargs*) → int
Get an action for a specific state in the environment.

restore(*path: str*, ***kwargs*)
Restore the agent from the file specified in *path*.

save(*path: str*, ***kwargs*)
Save the agent to the directory specified in *path*.

train(*n_steps: int = None*, *n_episodes: int = None*, *save_every: int = None*, *save_path: str = None*, *callback: callable = None*, ***kwargs*) → float
Train the agent in the environment and return the mean reward.

update_networks(*model: tensortrade.agents.parallel.parallel_dqn_model.ParallelDQNModel*)

update_target_network()

tensortrade.agents.parallel.parallel_dqn_model module

class tensortrade.agents.parallel.parallel_dqn_model.**ParallelDQNModel**(*create_env: Callable[[], TradingEnvironment], policy_network: <sphinx.ext.autodoc.importer.* = *None*)

Bases: object

get_action(*state: numpy.ndarray*, ***kwargs*) → int

restore(*path: str*, ***kwargs*)

save(*path: str*, ***kwargs*)

update_networks(*model: tensortrade.agents.parallel.parallel_dqn_model.ParallelDQNModel*)

update_target_network()

tensortrade.agents.parallel.parallel_dqn_optimizer module

class tensortrade.agents.parallel.parallel_dqn_optimizer.**ParallelDQNOptimizer**(*model: ParallelDQNModel*, *n_envs: int*, *memory_queue: multiprocessing.context.BaseC*, *model_update_que*, *multiprocessing.context.BaseC*, *done_queue: multiprocessing.context.BaseC*, *discount_factor: float = 0.9999*, *batch_size: int = 128*, *learning_rate: float = 0.001*, *memory_capacity: int = 10000*)

Bases: multiprocessing.context.Process

run()

Method to be run in sub-process; can be overridden in sub-class

tensortrade.agents.parallel.parallel_dqn_trainer module

class tensortrade.agents.parallel.parallel_dqn_trainer.**ParallelDQNTrainer**(*agent: ParallelDQNAgent, create_env: Callable[[None], TrainingEnvironment], memory_queue: multiprocessing.context.BaseContext.Queue, model_update_queue: multiprocessing.context.BaseContext.Queue, done_queue: multiprocessing.context.BaseContext.Queue, n_steps: int, n_episodes: int, eps_end: int = 0.05, eps_start: int = 0.99, eps_decay_steps: int = 2000, update_target_every: int = 2*)

Bases: multiprocessing.context.Process

run()
Method to be run in sub-process; can be overridden in sub-class

tensortrade.agents.parallel.parallel_queue module

class tensortrade.agents.parallel.parallel_queue.**ParallelQueue**
Bases: multiprocessing.queues.Queue

A portable implementation of multiprocessing.Queue.

Because of multithreading / multiprocessing semantics, Queue.qsize() may raise the NotImplementedError exception on Unix platforms like Mac OS X where sem_getvalue() is not implemented. This subclass addresses this problem by using a synchronized shared counter (initialized to zero) and increasing / decreasing its value every time the put() and get() methods are called, respectively. This not only prevents NotImplementedError from being raised, but also allows us to implement a reliable version of both qsize() and empty().

empty() → bool
Reliable implementation of multiprocessing.Queue.empty().

get(**args*, ***kwargs*) → object

put(**args*, ***kwargs*) → None

qsize() → int
Reliable implementation of multiprocessing.Queue.qsize().

class tensortrade.agents.parallel.parallel_queue.**SharedCounter**(*n: int = 0*)
Bases: object

A synchronized shared counter.

The locking done by multiprocessing.Value ensures that only a single process or thread may read or write the in-memory ctypes object. However, in order to do n += 1, Python performs a read followed by a write, so a second process may read the old value before the new one is written by the first process. The solution is to use a multiprocessing.Lock to guarantee the atomicity of the modifications to Value.

Parameters **n** (*int*) – The count to start at.

References

increment(*n: int = 1*) → None
Increment the counter by n.

Parameters **n** (*int*) – The amount to increment the counter by.

value
The value of the counter. (int, read-only)

Submodules

tensortrade.agents.a2c_agent module

References

- http://inoryy.com/post/tensorflow2-deep-reinforcement-learning/#agent-interface

class tensortrade.agents.a2c_agent.**A2CAgent**(*env: TradingEnvironment*, *shared_network: <sphinx.ext.autodoc.importer._MockObject object at 0x7f1ed878bf90> = None*, *actor_network: <sphinx.ext.autodoc.importer._MockObject object at 0x7f1ed878bf50> = None*, *critic_network: <sphinx.ext.autodoc.importer._MockObject object at 0x7f1ed878bf10> = None*)

Bases: *tensortrade.agents.agent.Agent*

get_action(*state: numpy.ndarray*, ***kwargs*) → int
Get an action for a specific state in the environment.

restore(*path: str*, ***kwargs*)
Restore the agent from the file specified in *path*.

save(*path: str*, ***kwargs*)
Save the agent to the directory specified in *path*.

train(*n_steps: int = None*, *n_episodes: int = None*, *save_every: int = None*, *save_path: str = None*, *callback: callable = None*, ***kwargs*) → float
Train the agent in the environment and return the mean reward.

class tensortrade.agents.a2c_agent.**A2CTransition**(*state*, *action*, *reward*, *done*, *value*)

Bases: tuple

action
Alias for field number 1

done
Alias for field number 3

reward
Alias for field number 2

state
Alias for field number 0

value
Alias for field number 4

tensortrade.agents.agent module

class tensortrade.agents.agent.**Agent**

Bases: *tensortrade.core.base.Identifiable*

get_action(*state: numpy.ndarray*, ***kwargs*) → int
Get an action for a specific state in the environment.

restore(*path: str*, ***kwargs*)
Restore the agent from the file specified in *path*.

save(*path: str*, ***kwargs*)
Save the agent to the directory specified in *path*.

train(*n_steps: int = None*, *n_episodes: int = 10000*, *save_every: int = None*, *save_path: str = None*, *callback: callable = None*, ***kwargs*) → float
Train the agent in the environment and return the mean reward.

tensortrade.agents.dqn_agent module

class tensortrade.agents.dqn_agent.**DQNAgent**(*env: TradingEnv*, *policy_network: <sphinx.ext.autodoc.importer._MockObject object at 0x7f1ed895ec90> = None*)

Bases: *tensortrade.agents.agent.Agent*

- https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow
- https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#dqn-algorithm
- https://arxiv.org/abs/1802.00308

get_action(*state: numpy.ndarray*, ***kwargs*) → int
Get an action for a specific state in the environment.

restore(*path: str*, ***kwargs*)
Restore the agent from the file specified in *path*.

save(*path: str*, ***kwargs*)
Save the agent to the directory specified in *path*.

train(*n_steps: int = 1000*, *n_episodes: int = 10*, *save_every: int = None*, *save_path: str = 'agent/'*, *callback: callable = None*, ***kwargs*) → float
Train the agent in the environment and return the mean reward.

class tensortrade.agents.dqn_agent.**DQNTransition**(*state*, *action*, *reward*, *next_state*, *done*)

Bases: tuple

action
Alias for field number 1

done
Alias for field number 4

next_state
Alias for field number 3

reward
Alias for field number 2

state
Alias for field number 0

tensortrade.agents.replay_memory module

class tensortrade.agents.replay_memory.**ReplayMemory**(*capacity: int*, *transition_type: collections.namedtuple = <class 'tensortrade.agents.replay_memory.Transition'>*)

Bases: object

head(*batch_size*) → List[collections.namedtuple]

push(**args*)

sample(*batch_size*) → List[collections.namedtuple]

tail(*batch_size*) → List[collections.namedtuple]

class tensortrade.agents.replay_memory.**Transition**(*state*, *action*, *reward*, *done*)
Bases: tuple

action
Alias for field number 1

done
Alias for field number 3

reward
Alias for field number 2

state
Alias for field number 0

tensortrade.contrib package

tensortrade.core package

Submodules

tensortrade.core.base module

Responsible for the basic classes in the project.

tensortrade.core.base.**global_clock**
A clock that provides a global reference for all objects that share a timeline.

Type *Clock*

class tensortrade.core.base.**Identifiable**
Bases: object

Identifiable mixin for adding a unique *id* property to instances of a class.

id
Gets the identifier for the object.

Returns *str* – The identifier for the object.

class tensortrade.core.base.**Observable**
Bases: object

An object with some value that can be observed.

An object to which a *listener* can be attached to and be alerted about on an event happening.

listeners
A list of listeners that the object will alert on events occurring.

Type list of listeners

attach(*listener*)
Adds a listener to receive alerts.

detach(*listener*)
Removes a listener from receiving alerts.

attach(*listener*) → tensortrade.core.base.Observable
Adds a listener to receive alerts.

Parameters **listener** (*a listener object*) –

Returns *Observable* – The observable being called.

detach (*listener*) → tensortrade.core.base.Observable
Removes a listener from receiving alerts.

Parameters **listener** (*a listener object*) –

Returns *Observable* – The observable being called.

class tensortrade.core.base.**TimeIndexed**
Bases: object

A class for objects that are indexed by time.

clock
Gets the clock associated with this object.

Returns *Clock* – The clock associated with this object.

class tensortrade.core.base.**TimedIdentifiable**
Bases: *tensortrade.core.base.Identifiable*, *tensortrade.core.base.TimeIndexed*

A class an identifiable object embedded in a time process.

created_at
The time at which this object was created according to its associated clock.

Type *datetime.datetime*

clock
Gets the clock associated with the object.

Returns *Clock* – The clock associated with the object.

tensortrade.core.clock module

class tensortrade.core.clock.**Clock**
Bases: object

A class to track the time for a process.

start
The time of start for the clock.

Type int

step
The time of the process the clock is at currently.

Type int

now (*format=None*)
Gets the current time in the provided format.

increment ()
Increments the clock by specified time increment.

reset ()
Resets the clock.

increment () → None
Increments the clock by specified time increment.

now (*format: str = None*) → datetime.datetime
Gets the current time in the provided format. :param format: The format to put the current time into. :type format: str or None, optional

Returns *datetime* – The current time.

reset () → None
Resets the clock.

tensortrade.core.component module

class tensortrade.core.component.**Component**
Bases: abc.ABC, *tensortrade.core.component.ContextualizedMixin*, *tensortrade.core.base.Identifiable*

The main class for setting up components to be used in the *TradingEnv*.

This class if responsible for providing a common way in which different components of the library can be created. Specifically, it enables the creation of components from a *TradingContext*. Therefore making the creation of complex environments simpler where there are only a few things that need to be changed from case to case.

registered_name
The name under which constructor arguments are to be given in a dictionary and passed to a *TradingContext*.

Type str

classmethod __init_subclass__ (****kwargs*) → None
Constructs the concrete subclass of *Component*.

In constructing the subclass, the concrete subclass is also registered into the project level registry.

Parameters **kwargs** (*keyword arguments*) – The keyword arguments to be provided to the concrete subclass of *Component* to create an instance.

default (*key: str*, *value: Any*, *kwargs: dict = None*) → Any
Resolves which defaults value to use for construction.

A concrete subclass will use this method to resolve which default value it should use when creating an instance. The default value should go to the value specified for the variable within the *TradingContext*. If that one is not provided it will resolve to *value*.

Parameters

- **key** (*str*) – The name of the attribute to be resolved for the class.
- **value** (*any*) – The *value* the attribute should be set to if not provided in the *TradingContext*.
- **kwargs** (*dict, optional*) – The dictionary to search through for the value associated with *key*.

registered_name = None

class tensortrade.core.component.**ContextualizedMixin**
Bases: object

A mixin that is to be mixed with any class that must function in a contextual setting.

context
Gets the *Context* the object is under.

Returns *Context* – The context the object is under.

class tensortrade.core.component.**InitContextMeta**
Bases: abc.ABCMeta

Metaclass that executes *__init__* of instance in its core.

This class works with the *TradingContext* class to ensure the correct data is being given to the instance created by a concrete class that has subclassed *Component*.

__call__(**args*, ***kwargs*) → tensortrade.core.component.InitContextMeta

Parameters

- **args** – positional arguments to give constructor of subclass of *Component*
- **kwargs** – keyword arguments to give constructor of subclass of *Component*

Returns *Component* – An instance of a concrete class the subclasses *Component*

tensortrade.core.context module

class tensortrade.core.context.**Context**(***kwargs*)
Bases: collections.UserDict

A context that is injected into every instance of a class that is a subclass of *Component*.

class tensortrade.core.context.**TradingContext**(*config: dict*)
Bases: collections.UserDict

A class for objects that put themselves in a *Context* using the *with* statement.

The implementation for this class is heavily borrowed from the pymc3 library and adapted with the design goals of TensorTrade in mind.

Parameters **config** (*dict*) – The configuration holding the information for each *Component*.

from_json(*path*)
Creates a *TradingContext* from a json file.

from_yaml(*path*)
Creates a *TradingContext* from a yaml file.

Warning: If there is a conflict in the contexts of different components because they were initialized under different contexts, can have undesirable effects. Therefore, a warning should be made to the user indicating that using components together that have conflicting contexts can lead to unwanted behavior.

References

[1] https://github.com/pymc-devs/pymc3/blob/master/pymc3/model.py

__enter__() → tensortrade.core.context.TradingContext
Adds a new *TradingContext* to the context stack.

This method is used for a *with* statement and adds a *TradingContext* to the context stack. The new context on the stack is then used by every class that subclasses *Component* the initialization of its instances.

Returns *TradingContext* – The context associated with the given with statement.

__exit__ (*typ*, *value*, *traceback*) → None
Pops the first *TradingContext* of the stack.

Parameters

- **typ** (*type*) – The type of *Exception*
- **value** (*Exception*) – An instance of *typ*.
- **traceback** (*python traceback object*) – The traceback object associated with the exception.

contexts = <_thread._local object>

classmethod from_json (*path: str*) → tensortrade.core.context.TradingContext
Creates a *TradingContext* from a json file.

Parameters **path** (*str*) – The path to locate the json file.

Returns *TradingContext* – A trading context with all the variables provided in the json file.

classmethod from_yaml (*path: str*) → tensortrade.core.context.TradingContext
Creates a *TradingContext* from a yaml file.

Parameters **path** (*str*) – The path to locate the yaml file.

Returns *TradingContext* – A trading context with all the variables provided in the yaml file.

classmethod get_context () → tensortrade.core.context.TradingContext
Gets the first context on the stack.

Returns *TradingContext* – The first context on the stack.

classmethod get_contexts () → List[tensortrade.core.context.TradingContext]
Gets the stack of trading contexts.

Returns *List['TradingContext']* – The stack of trading contexts.

shared
The shared values in common for all components involved with the *TradingContext*.

Returns *dict* – Shared values for components under the *TradingContext*.

tensortrade.core.exceptions module

Holds all the exceptions for the project.

exception tensortrade.core.exceptions.**DoubleLockedQuantity** (*quantity: Quantity*, **args*)

Bases: Exception

Raised when a locked *Quantity* is trying to get locked again.

Parameters

- **quantity** (*Quantity*) – A locked quantity.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**DoubleUnlockedQuantity** (*quantity: Quantity*, **args*)

Bases: Exception

Raised when a free *Quantity* is trying to get unlocked.

Parameters

- **quantity** (*Quantity*) – A unlocked quantity.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**IncompatibleInstrumentOperation**(*left: Quantity*, *right: Quantity*, **args*)

Bases: Exception

Raised when two quantities with different instruments occurs.

Parameters

- **left** (*Quantity*) – The left argument of the operation.
- **right** (*Quantity*) – The right argument of the operation.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**InsufficientFunds**(*balance: Quantity*, *size: Quantity*, **args*)

Bases: Exception

Raised when requested funds are greater than the free balance of a *Wallet*

Parameters

- **balance** (*Quantity*) – The balance of the *Wallet* where funds are being allocated from.
- **size** (*Quantity*) – The amount being requested for allocation.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**InvalidNegativeQuantity**(*size: float*, **args*)

Bases: Exception

Raised when a *Quantity* tries to be instantiated with a negative amount.

Parameters

- **size** (*float*) – The size that was specified for the *Quantity*.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**InvalidNonNumericQuantity**(*size: Union[float, int, numbers.Number]*, **args*)

Bases: Exception

Raised when a *Quantity* tries to be instantiated with a value that is not numeric.

Parameters

- **size** (*Union[float, int, Number]*) – The value that was specified for the *Quantity*.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**InvalidOrderQuantity**(*quantity: Quantity*, **args*)

Bases: Exception

Raised when an *Order* with a non-negative amount is placed

Parameters

- **quantity** (*Quantity*) – An invalid order quantity.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**InvalidTradingPair**(*base: Instrument*, *quote: Instrument*, **args*)

Bases: Exception

Raised when an invalid trading pair is trying to be created.

Parameters

- **base** (*'Instrument'*) – The base instrument of the pair.
- **quote** (*'Instrument'*) – The quote instrument of the pair.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**QuantityNotLocked**(*quantity: Quantity*, **args*)

Bases: Exception

Raised when a locked *Quantity* does not have a path_id in the *Wallet* it is trying to be unlocked in.

Parameters

- **quantity** (*Quantity*) – A locked quantity.
- ***args** (*positional arguments*) – More positional arguments for the exception.

exception tensortrade.core.exceptions.**QuantityOpPathMismatch**(*left_id: str*, *right_id: str*, **args*)

Bases: Exception

Raised when an operation tries to occur between quantities that are not under the same path_id.

Parameters

- **left_id** (*str*) – The path_id for the left argument in the operation.
- **right_id** (*str*) – The path_id for the right argument in the operation.
- ***args** (*positional arguments*) – More positional arguments for the exception.

tensortrade.core.registry module

This module hold the project level registry and provides methods to mutate and change the registry.

tensortrade.core.registry.**MAJOR_COMPONENTS**

The list of the major components that can be injected into.

Type List[str]

tensortrade.core.registry.**register**(*component: Component*, *registered_name: str*) → None

Registers a component into the registry

Parameters

- **component** (*'Component'*) – The component to be registered.
- **registered_name** (*str*) – The name to be associated with the registered component.

tensortrade.core.registry.**registry**() → dict
Gets the project level registry.

Returns *dict* – The project level registry.

tensortrade.data package

Submodules

tensortrade.data.cdd module

Contains methods and classes to collect data from https://www.cryptodatadownload.com.

class tensortrade.data.cdd.**CryptoDataDownload**
Bases: object

Provides methods for retrieving data on different cryptocurrencies from https://www.cryptodatadownload.com/cdd/.

url
The url for collecting data from CryptoDataDownload.

Type str

fetch(*exchange_name*, *base_symbol*, *quote_symbol*, *timeframe*, *include_all_volumes=False*)
Fetches data for different exchanges and cryptocurrency pairs.

fetch(*exchange_name: str*, *base_symbol: str*, *quote_symbol: str*, *timeframe: str*, *include_all_volumes: bool = False*) → pandas.core.frame.DataFrame
Fetches data for different exchanges and cryptocurrency pairs.

Parameters

- **exchange_name** (*str*) – The name of the exchange.
- **base_symbol** (*str*) – The base symbol fo the cryptocurrency pair.
- **quote_symbol** (*str*) – The quote symbol fo the cryptocurrency pair.
- **timeframe** (*{"d", "h", "m"}*) – The timeframe to collect data from.
- **include_all_volumes** (*bool, optional*) – Whether or not to include both base and quote volume.

Returns *pd.DataFrame* – A open, high, low, close and volume for the specified exchange and cryptocurrency pair.

fetch_default(*exchange_name: str*, *base_symbol: str*, *quote_symbol: str*, *timeframe: str*, *include_all_volumes: bool = False*) → pandas.core.frame.DataFrame
Fetches data from all exchanges that match the evaluation structure.

Parameters

- **exchange_name** (*str*) – The name of the exchange.
- **base_symbol** (*str*) – The base symbol fo the cryptocurrency pair.
- **quote_symbol** (*str*) – The quote symbol fo the cryptocurrency pair.
- **timeframe** (*{"d", "h", "m"}*) – The timeframe to collect data from.
- **include_all_volumes** (*bool, optional*) – Whether or not to include both base and quote volume.

Returns *pd.DataFrame* – A open, high, low, close and volume for the specified exchange and cryptocurrency pair.

fetch_gemini(*base_symbol: str, quote_symbol: str, timeframe: str*) → pandas.core.frame.DataFrame

Fetches data from the gemini exchange.

Parameters

- **base_symbol** (*str*) – The base symbol fo the cryptocurrency pair.
- **quote_symbol** (*str*) – The quote symbol fo the cryptocurrency pair.
- **timeframe** (*{"d", "h", "m"}*) – The timeframe to collect data from.

Returns *pd.DataFrame* – A open, high, low, close and volume for the specified cryptocurrency pair.

tensortrade.env package

Subpackages

tensortrade.env.default package

tensortrade.env.default.**create**(*portfolio: tensortrade.oms.wallets.portfolio.Portfolio, action_scheme: Union[tensortrade.env.default.actions.TensorTradeActionScheme, str], reward_scheme: Union[tensortrade.env.default.rewards.TensorTradeRewardScheme, str], feed: tensortrade.feed.core.feed.DataFeed, window_size: int = 1, min_periods: int = None, random_start_pct: float = 0.0, **kwargs*) → tensortrade.env.generic.environment.TradingEnv

Creates the default *TradingEnv* of the project to be used in training RL agents.

Parameters

- **portfolio** (*Portfolio*) – The portfolio to be used by the environment.
- **action_scheme** (*actions.TensorTradeActionScheme* or str) – The action scheme for computing actions at every step of an episode.
- **reward_scheme** (*rewards.TensorTradeRewardScheme* or str) – The reward scheme for computing rewards at every step of an episode.
- **feed** (*DataFeed*) – The feed for generating observations to be used in the look back window.
- **window_size** (*int*) – The size of the look back window to use for the observation space.
- **min_periods** (*int, optional*) – The minimum number of steps to warm up the *feed*.
- **random_start_pct** (*float, optional*) – Whether to randomize the starting point within the environment at each observer reset, starting in the first X percentage of the sample
- ****kwargs** (*keyword arguments*) – Extra keyword arguments needed to build the environment.

Returns *TradingEnv* – The default trading environment.

Submodules

tensortrade.env.default.actions module

class tensortrade.env.default.actions.**BSH**(*cash: Wallet*, *asset: Wallet*)
Bases: *tensortrade.env.default.actions.TensorTradeActionScheme*

A simple discrete action scheme where the only options are to buy, sell, or hold.

Parameters

- **cash** (*Wallet*) – The wallet to hold funds in the base intrument.
- **asset** (*Wallet*) – The wallet to hold funds in the quote instrument.

action_space
The action space of the *TradingEnv*. (*Space*, read-only)

attach(*listener*)

get_orders(*action: int*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*) → tensortrade.oms.orders.order.Order
Gets the list of orders to be submitted for the given action.

Parameters

- **action** (*Any*) – The action to be interpreted.
- **portfolio** (*'Portfolio'*) – The portfolio defined for the environment.

Returns *List[Order]* – A list of orders to be submitted to the broker.

registered_name = 'bsh'

reset()
Resets the action scheme.

class tensortrade.env.default.actions.**ManagedRiskOrders**(*stop: List[float] = [0.02, 0.04, 0.06]*, *take: List[float] = [0.01, 0.02, 0.03]*, *trade_sizes: Union[List[float], int] = 10*, *durations: Union[List[int], int, None] = None*, *trade_type: tensortrade.oms.orders.trade.TradeType = <TradeType.MARKET: 'market'>*, *order_listener: Optional[tensortrade.oms.orders.order_listener.Order = None*, *min_order_pct: float = 0.02*, *min_order_abs: float = 0.0*)
Bases: *tensortrade.env.default.actions.TensorTradeActionScheme*

A discrete action scheme that determines actions based on managing risk, through setting a follow-up stop loss and take profit on every order.

Parameters

- **stop** (*List[float]*) – A list of possible stop loss percentages for each order.
- **take** (*List[float]*) – A list of possible take profit percentages for each order.
- **trade_sizes** (*List[float]*) – A list of trade sizes to select from when submitting an order. (e.g. '[1, 1/3]' = 100% or 33% of balance is tradable. '4' = 25%, 50%, 75%, or 100% of balance is tradable.)
- **durations** (*List[int]*) – A list of durations to select from when submitting an order.
- **trade_type** (*TradeType*) – A type of trade to make.
- **order_listener** (OrderListener) – A callback class to use for listening to steps of the order process.
- **min_order_pct** (*float*) – The minimum value when placing an order, calculated in percent over net_worth.
- **min_order_abs** (*float*) – The minimum value when placing an order, calculated in absolute order value.

action_space
The action space of the *TradingEnv*. (*Space*, read-only)

get_orders(*action: int, portfolio: tensortrade.oms.wallets.portfolio.Portfolio*) → List[tensortrade.oms.orders.order.Order]
Gets the list of orders to be submitted for the given action.

Parameters

- **action** (*Any*) – The action to be interpreted.
- **portfolio** ('*Portfolio*') – The portfolio defined for the environment.

Returns *List[Order]* – A list of orders to be submitted to the broker.

class tensortrade.env.default.actions.**SimpleOrders**(*criteria: Union[List[OrderCriteria], OrderCriteria] = None, trade_sizes: Union[List[float], int] = 10, durations: Union[List[int], int] = None, trade_type: TradeType = <TradeType.MARKET: 'market'>, order_listener: OrderListener = None, min_order_pct: float = 0.02, min_order_abs: float = 0.0*)
Bases: *tensortrade.env.default.actions.TensorTradeActionScheme*

A discrete action scheme that determines actions based on a list of trading pairs, order criteria, and trade sizes.

Parameters

- **criteria** (*List[OrderCriteria]*) – A list of order criteria to select from when submitting an order. (e.g. MarketOrder, LimitOrder w/ price, StopLoss, etc.)
- **trade_sizes** (*List[float]*) – A list of trade sizes to select from when submitting an order. (e.g. '[1, 1/3]' = 100% or 33% of balance is tradable. '4' = 25%, 50%, 75%, or 100% of balance is tradable.)
- **durations** (*List[int]*) – A list of durations to select from when submitting an order.
- **trade_type** (TradeType) – A type of trade to make.

- **order_listener** (OrderListener) – A callback class to use for listening to steps of the order process.
- **min_order_pct** (*float*) – The minimum value when placing an order, calculated in percent over net_worth.
- **min_order_abs** (*float*) – The minimum value when placing an order, calculated in absolute order value.

action_space
The action space of the *TradingEnv*. (*Space*, read-only)

get_orders(*action: int*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*) → List[tensortrade.oms.orders.order.Order]
Gets the list of orders to be submitted for the given action.

Parameters

- **action** (*Any*) – The action to be interpreted.
- **portfolio** ('*Portfolio*') – The portfolio defined for the environment.

Returns *List[Order]* – A list of orders to be submitted to the broker.

class tensortrade.env.default.actions.**TensorTradeActionScheme**
Bases: *tensortrade.env.generic.components.action_scheme.ActionScheme*

An abstract base class for any *ActionScheme* that wants to be compatible with the built in OMS.

The structure of the action scheme is built to make sure that action space can be used with the system, provided that the user defines the methods to interpret that action.

portfolio
The portfolio object to be used in defining actions.

Type 'Portfolio'

broker
The broker object to be used for placing orders in the OMS.

Type 'Broker'

perform(*env*, *portfolio*)
Performs the action on the given environment.

get_orders(*action*, *portfolio*)
Gets the list of orders to be submitted for the given action.

clock
The reference clock from the environment. (*Clock*)

When the clock is set for the we also set the clock for the portfolio as well as the exchanges defined in the portfolio.

Returns *Clock* – The environment clock.

get_orders(*action: Any*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*) → List[tensortrade.oms.orders.order.Order]
Gets the list of orders to be submitted for the given action.

Parameters

- **action** (*Any*) – The action to be interpreted.
- **portfolio** ('*Portfolio*') – The portfolio defined for the environment.

Returns *List[Order]* – A list of orders to be submitted to the broker.

perform(*env: tensortrade.env.generic.environment.TradingEnv*, *action: Any*) → None
Performs the action on the given environment.

Under the TT action scheme, the subclassed action scheme is expected to provide a method for getting a list of orders to be submitted to the broker for execution in the OMS.

Parameters

- **env** (*'TradingEnv'*) – The environment to perform the action on.
- **action** (*Any*) – The specific action selected from the action space.

reset() → None
Resets the action scheme.

tensortrade.env.default.actions.**get**(*identifier: str*) → tensortrade.env.generic.components.action_scheme.ActionScheme
Gets the *ActionScheme* that matches with the identifier.

Parameters **identifier** (*str*) – The identifier for the *ActionScheme*.

Returns *'ActionScheme'* – The action scheme associated with the *identifier*.

Raises KeyError: – Raised if the *identifier* is not associated with any *ActionScheme*.

tensortrade.env.default.informers module

class tensortrade.env.default.informers.**TensorTradeInformer**
Bases: *tensortrade.env.generic.components.informer.Informer*

info(*env: tensortrade.env.generic.environment.TradingEnv*) → dict
Provides information at a given step of an episode.

Parameters **env** (*'TradingEnv'*) – The trading environment.

Returns *dict* – A dictionary of information about the portfolio and net worth.

tensortrade.env.default.observers module

class tensortrade.env.default.observers.**IntradayObserver**(*portfolio: Portfolio*, *feed: DataFeed = None*, *renderer_feed: DataFeed = None*, *stop_time: datetime.time = datetime.time(16, 0)*, *window_size: int = 1*, *min_periods: int = None*, *randomize: bool = False*, ***kwargs*)
Bases: *tensortrade.env.generic.components.observer.Observer*

The IntradayObserver observer that is compatible with the other *default* components. :param portfolio: The portfolio to be used to create the internal data feed mechanism. :type portfolio: *Portfolio* :param feed: The feed to be used to collect observations to the observation window. :type feed: *DataFeed* :param renderer_feed: The feed to be used for giving information to the renderer. :type renderer_feed: *DataFeed* :param stop_time: The time at which the episode will stop. :type stop_time: datetime.time :param window_size: The size of the observation window. :type window_size: int :param min_periods: The amount of steps needed to warmup the *feed*. :type min_periods: int :param randomize: Whether or not to select a random episode when reset.

:type randomize: bool :param **kwargs: Additional keyword arguments for observer creation. :type **kwargs: keyword arguments

feed
The master feed in charge of streaming the internal, external, and renderer data feeds.

Type *DataFeed*

stop_time
The time at which the episode will stop.

Type datetime.time

window_size
The size of the observation window.

Type int

min_periods
The amount of steps needed to warmup the *feed*.

Type int

randomize
Whether or not a random episode is selected when reset.

Type bool

history
The observation history.

Type *ObservationHistory*

renderer_history
The history of the renderer data feed.

Type *List[dict]*

has_next() → bool
Checks if there is another observation to be generated. :returns: *bool* – Whether there is another observation to be generated.

observation_space
The observation space of the *TradingEnv*. (*Space*, read-only)

observe(*env: TradingEnv*) → numpy.array
Observes the environment. As a consequence of observing the *env*, a new observation is generated from the *feed* and stored in the observation history. :returns: *np.array* – The current observation of the environment.

reset() → None
Resets the observer

warmup() → None
Warms up the data feed.

class tensortrade.env.default.observers.**ObservationHistory**(*window_size: int*)
Bases: object

Stores observations from a given episode of the environment.

Parameters **window_size** (*int*) – The amount of observations to keep stored before discarding them.

window_size
The amount of observations to keep stored before discarding them.

Type int

rows
The rows of observations that are used as the environment observation at each step of an episode.

Type pd.DataFrame

observe() → numpy.array
Gets the observation at a given step in an episode

Returns *np.array* – The current observation of the environment.

push(*row: dict*) → None
Stores an observation.

Parameters **row** (*dict*) – The new observation to store.

reset() → None
Resets the observation history

class tensortrade.env.default.observers.**TensorTradeObserver**(*portfolio: Portfolio*, *feed: DataFeed = None*, *renderer_feed: DataFeed = None*, *window_size: int = 1*, *min_periods: int = None*, ***kwargs*)
Bases: *tensortrade.env.generic.components.observer.Observer*

The TensorTrade observer that is compatible with the other *default* components.

Parameters

- **portfolio** (*Portfolio*) – The portfolio to be used to create the internal data feed mechanism.
- **feed** (*DataFeed*) – The feed to be used to collect observations to the observation window.
- **renderer_feed** (*DataFeed*) – The feed to be used for giving information to the renderer.
- **window_size** (*int*) – The size of the observation window.
- **min_periods** (*int*) – The amount of steps needed to warmup the *feed*.
- ****kwargs** (*keyword arguments*) – Additional keyword arguments for observer creation.

feed
The master feed in charge of streaming the internal, external, and renderer data feeds.

Type *DataFeed*

window_size
The size of the observation window.

Type int

min_periods
The amount of steps needed to warmup the *feed*.

Type int

history
The observation history.

Type *ObservationHistory*

renderer_history
The history of the renderer data feed.

Type *List[dict]*

has_next() → bool
Checks if there is another observation to be generated.

Returns *bool* – Whether there is another observation to be generated.

observation_space
The observation space of the *TradingEnv*. (*Space*, read-only)

observe(*env: TradingEnv*) → numpy.array
Observes the environment.

As a consequence of observing the *env*, a new observation is generated from the *feed* and stored in the observation history.

Returns *np.array* – The current observation of the environment.

reset(*random_start=0*) → None
Resets the observer

warmup() → None
Warms up the data feed.

tensortrade.env.default.renderers module

class tensortrade.env.default.renderers.**BaseRenderer**
Bases: *tensortrade.env.generic.components.renderer.Renderer*

The abstract base renderer to be subclassed when making a renderer the incorporates a *Portfolio*.

render(*env: tensortrade.env.generic.environment.TradingEnv*, ***kwargs*)
Renders a view of the environment at the current step of an episode.

Parameters

- **env** (*'TradingEnv'*) – The trading environment.
- **kwargs** (*keyword arguments*) – Additional keyword arguments for rendering the environment.

render_env(*episode: int = None*, *max_episodes: int = None*, *step: int = None*, *max_steps: int = None*, *price_history: Optional[pandas.core.frame.DataFrame] = None*, *net_worth: Optional[pandas.core.series.Series] = None*, *performance: Optional[pandas.core.frame.DataFrame] = None*, *trades: Optional[collections.OrderedDict] = None*) → None
Renderers the current state of the environment.

Parameters

- **episode** (*int*) – The episode that the environment is being rendered for.
- **max_episodes** (*int*) – The maximum number of episodes that will occur.
- **step** (*int*) – The step of the current episode that is happening.
- **max_steps** (*int*) – The maximum number of steps that will occur in an episode.

- **price_history** (*pd.DataFrame*) – The history of instrument involved with the environment. The required columns are: date, open, high, low, close, and volume.
- **net_worth** (*pd.Series*) – The history of the net worth of the *portfolio*.
- **performance** (*pd.Series*) – The history of performance of the *portfolio*.
- **trades** (*OrderedDict*) – The history of trades for the current episode.

reset() → None
Resets the renderer.

save() → None
Saves the rendering of the *TradingEnv*.

class tensortrade.env.default.renderers.**EmptyRenderer**
Bases: *tensortrade.env.generic.components.renderer.Renderer*

A renderer that does renders nothing.

Needed to make sure that environment can function without requiring a renderer.

render(*env*, ***kwargs*)
Renders a view of the environment at the current step of an episode.

Parameters

- **env** (*'TradingEnv'*) – The trading environment.
- **kwargs** (*keyword arguments*) – Additional keyword arguments for rendering the environment.

class tensortrade.env.default.renderers.**FileLogger**(*filename: str = None*, *path: str = 'log'*, *log_format: str = None*, *timestamp_format: str = None*)
Bases: *tensortrade.env.default.renderers.BaseRenderer*

Logs information to a file.

Parameters

- **filename** (*str*) – The file name of the log file. If omitted, a file name will be created automatically.
- **path** (*str*) – The path to save the log files to. None to save to same script directory.
- **log_format** (*str*) – The log entry format as per Python logging. None for default. For more details, refer to https://docs.python.org/3/library/logging.html
- **timestamp_format** (*str*) – The format of the timestamp of the log entry. Node for default.

DEFAULT_LOG_FORMAT = '[%(asctime)-15s] %(message)s'

DEFAULT_TIMESTAMP_FORMAT = '%Y-%m-%d %H:%M:%S'

log_file
The filename information is being logged to. (str, read-only)

render_env(*episode: int = None*, *max_episodes: int = None*, *step: int = None*, *max_steps: int = None*, *price_history: pandas.core.frame.DataFrame = None*, *net_worth: pandas.core.series.Series = None*, *performance: pandas.core.frame.DataFrame = None*, *trades: Optional[collections.OrderedDict] = None*) → None
Renderers the current state of the environment.

Parameters

- **episode** (*int*) – The episode that the environment is being rendered for.
- **max_episodes** (*int*) – The maximum number of episodes that will occur.
- **step** (*int*) – The step of the current episode that is happening.
- **max_steps** (*int*) – The maximum number of steps that will occur in an episode.
- **price_history** (*pd.DataFrame*) – The history of instrument involved with the environment. The required columns are: date, open, high, low, close, and volume.
- **net_worth** (*pd.Series*) – The history of the net worth of the *portfolio*.
- **performance** (*pd.Series*) – The history of performance of the *portfolio*.
- **trades** (*OrderedDict*) – The history of trades for the current episode.

class tensortrade.env.default.renderers.**MatplotlibTradingChart**(*display: bool = True, save_format: str = None, path: str = 'charts', filename_prefix: str = 'chart_'*)

Bases: *tensortrade.env.default.renderers.BaseRenderer*

Trading visualization for TensorTrade using Matplotlib :param display: True to display the chart on the screen, False for not. :type display: bool :param save_format: A format to save the chart to. Acceptable formats are

png, jpg, svg, pdf.

Parameters

- **path** (*str*) – The path to save the char to if save_format is not None. The folder will be created if not found.
- **filename_prefix** (*str*) – A string that precedes automatically-created file name when charts are saved. Default '**chart_**'.

render_env(*episode: int = None, max_episodes: int = None, step: int = None, max_steps: int = None, price_history: Optional[pandas.core.frame.DataFrame] = None, net_worth: Optional[pandas.core.series.Series] = None, performance: Optional[pandas.core.frame.DataFrame] = None, trades: Optional[collections.OrderedDict] = None*) → None

Renderers the current state of the environment.

Parameters

- **episode** (*int*) – The episode that the environment is being rendered for.
- **max_episodes** (*int*) – The maximum number of episodes that will occur.
- **step** (*int*) – The step of the current episode that is happening.
- **max_steps** (*int*) – The maximum number of steps that will occur in an episode.
- **price_history** (*pd.DataFrame*) – The history of instrument involved with the environment. The required columns are: date, open, high, low, close, and volume.
- **net_worth** (*pd.Series*) – The history of the net worth of the *portfolio*.
- **performance** (*pd.Series*) – The history of performance of the *portfolio*.
- **trades** (*OrderedDict*) – The history of trades for the current episode.

reset() → None
Resets the renderer.

save() → None
Saves the rendering of the *TradingEnv*.

class tensortrade.env.default.renderers.**PlotlyTradingChart**(*display: bool = True, height: int = None, timestamp_format: str = '%Y-%m-%d %H:%M:%S', save_format: str = None, path: str = 'charts', filename_prefix: str = 'chart_', auto_open_html: bool = False, include_plotlyjs: Union[bool, str] = 'cdn'*)

Bases: *tensortrade.env.default.renderers.BaseRenderer*

Trading visualization for TensorTrade using Plotly.

Parameters

- **display** (*bool*) – True to display the chart on the screen, False for not.
- **height** (*int*) – Chart height in pixels. Affects both display and saved file charts. Set to None for 100% height. Default is None.
- **save_format** (*str*) – A format to save the chart to. Acceptable formats are html, png, jpeg, webp, svg, pdf, eps. All the formats except for 'html' require Orca. Default is None for no saving.
- **path** (*str*) – The path to save the char to if save_format is not None. The folder will be created if not found.
- **filename_prefix** (*str*) – A string that precedes automatically-created file name when charts are saved. Default '**chart_**'.
- **timestamp_format** (*str*) – The format of the date shown in the chart title.
- **auto_open_html** (*bool*) – Works for save_format='html' only. True to automatically open the saved chart HTML file in the default browser, False otherwise.
- **include_plotlyjs** (*Union[bool, str]*) – Whether to include/load the plotly.js library in the saved file. 'cdn' results in a smaller file by loading the library online but requires an Internet connect while True includes the library resulting in much larger file sizes. False to not include the library. For more details, refer to https://plot.ly/python-api-reference/generated/plotly.graph_objects.Figure.html

Notes

Possible Future Enhancements:

- Saving images without using Orca.
- Limit displayed step range for the case of a large number of steps and let the shown part of the chart slide after filling that range to keep showing recent data as it's being added.

References

render_env(*episode: int = None*, *max_episodes: int = None*, *step: int = None*, *max_steps: int = None*, *price_history: pandas.core.frame.DataFrame = None*, *net_worth: pandas.core.series.Series = None*, *performance: pandas.core.frame.DataFrame = None*, *trades: Optional[collections.OrderedDict] = None*) → None
Renderers the current state of the environment.

Parameters

- **episode** (*int*) – The episode that the environment is being rendered for.
- **max_episodes** (*int*) – The maximum number of episodes that will occur.
- **step** (*int*) – The step of the current episode that is happening.
- **max_steps** (*int*) – The maximum number of steps that will occur in an episode.
- **price_history** (*pd.DataFrame*) – The history of instrument involved with the environment. The required columns are: date, open, high, low, close, and volume.
- **net_worth** (*pd.Series*) – The history of the net worth of the *portfolio*.
- **performance** (*pd.Series*) – The history of performance of the *portfolio*.
- **trades** (*OrderedDict*) – The history of trades for the current episode.

reset() → None
Resets the renderer.

save() → None
Saves the current chart to a file.

Notes

All formats other than HTML require Orca installed and server running.

class tensortrade.env.default.renderers.**ScreenLogger**(*date_format: str = '%Y-%m-%d %H:%M:%S'*)
Bases: *tensortrade.env.default.renderers.BaseRenderer*

Logs information the screen of the user.

Parameters **date_format** (*str*) – The format for logging the date.

DEFAULT_FORMAT = '[%(asctime)-15s] %(message)s'

render_env(*episode: int = None*, *max_episodes: int = None*, *step: int = None*, *max_steps: int = None*, *price_history: pandas.core.frame.DataFrame = None*, *net_worth: pandas.core.series.Series = None*, *performance: pandas.core.frame.DataFrame = None*, *trades: Optional[collections.OrderedDict] = None*)
Renderers the current state of the environment.

Parameters

- **episode** (*int*) – The episode that the environment is being rendered for.
- **max_episodes** (*int*) – The maximum number of episodes that will occur.
- **step** (*int*) – The step of the current episode that is happening.
- **max_steps** (*int*) – The maximum number of steps that will occur in an episode.
- **price_history** (*pd.DataFrame*) – The history of instrument involved with the environment. The required columns are: date, open, high, low, close, and volume.

- **net_worth** (*pd.Series*) – The history of the net worth of the *portfolio*.
- **performance** (*pd.Series*) – The history of performance of the *portfolio*.
- **trades** (*OrderedDict*) – The history of trades for the current episode.

tensortrade.env.default.renderers.**get**(*identifier: str*) → tensortrade.env.default.renderers.BaseRenderer

Gets the *BaseRenderer* that matches the identifier.

Parameters **identifier** (*str*) – The identifier for the *BaseRenderer*

Returns *BaseRenderer* – The renderer associated with the *identifier*.

Raises KeyError: – Raised if identifier is not associated with any *BaseRenderer*

tensortrade.env.default.rewards module

class tensortrade.env.default.rewards.**PBR**(*price: tensortrade.feed.core.base.Stream*)

Bases: *tensortrade.env.default.rewards.TensorTradeRewardScheme*

A reward scheme for position-based returns.

- Let p_t denote the price at time t.
- Let x_t denote the position at time t.
- Let R_t denote the reward at time t.

Then the reward is defined as, $R_t = (p_t - p_{t-1}) \cdot x_t$.

Parameters **price** (*Stream*) – The price stream to use for computing rewards.

get_reward(*portfolio: Portfolio*) → float

Gets the reward associated with current step of the episode.

Parameters **portfolio** (*Portfolio*) – The portfolio associated with the *TensorTradeActionScheme*.

Returns *float* – The reward for the current step of the episode.

on_action(*action: int*) → None

registered_name = 'pbr'

reset() → None

Resets the *position* and *feed* of the reward scheme.

class tensortrade.env.default.rewards.**RiskAdjustedReturns**(*return_algorithm: str = 'sharpe', risk_free_rate: float = 0.0, target_returns: float = 0.0, window_size: int = 1*)

Bases: *tensortrade.env.default.rewards.TensorTradeRewardScheme*

A reward scheme that rewards the agent for increasing its net worth, while penalizing more volatile strategies.

Parameters

- **return_algorithm** (*{'sharpe', 'sortino'}, Default 'sharpe'.*) – The risk-adjusted return metric to use.
- **risk_free_rate** (*float, Default 0.*) – The risk free rate of returns to use for calculating metrics.

- **target_returns** (*float, Default 0*) – The target returns per period for use in calculating the sortino ratio.
- **window_size** (*int*) – The size of the look back window for computing the reward.

get_reward(*portfolio: Portfolio*) → float
Computes the reward corresponding to the selected risk-adjusted return metric.

Parameters **portfolio** (*Portfolio*) – The current portfolio being used by the environment.

Returns *float* – The reward corresponding to the selected risk-adjusted return metric.

class tensortrade.env.default.rewards.**SimpleProfit**(*window_size: int = 1*)
Bases: *tensortrade.env.default.rewards.TensorTradeRewardScheme*

A simple reward scheme that rewards the agent for incremental increases in net worth.

Parameters **window_size** (*int*) – The size of the look back window for computing the reward.

window_size
The size of the look back window for computing the reward.

Type int

get_reward(*portfolio: Portfolio*) → float
Rewards the agent for incremental increases in net worth over a sliding window.

Parameters **portfolio** (*Portfolio*) – The portfolio being used by the environment.

Returns *float* – The cumulative percentage change in net worth over the previous *window_size* time steps.

class tensortrade.env.default.rewards.**TensorTradeRewardScheme**
Bases: *tensortrade.env.generic.components.reward_scheme.RewardScheme*

An abstract base class for reward schemes for the default environment.

get_reward(*portfolio*) → float
Gets the reward associated with current step of the episode.

Parameters **portfolio** (*Portfolio*) – The portfolio associated with the *TensorTradeActionScheme*.

Returns *float* – The reward for the current step of the episode.

reward(*env: tensortrade.env.generic.environment.TradingEnv*) → float
Computes the reward for the current step of an episode.

Parameters **env** (*TradingEnv*) – The trading environment

Returns *float* – The computed reward.

tensortrade.env.default.rewards.**get**(*identifier: str*) → tensortrade.env.default.rewards.TensorTradeRewardScheme
Gets the *RewardScheme* that matches with the identifier.

Parameters **identifier** (*str*) – The identifier for the *RewardScheme*

Returns *TensorTradeRewardScheme* – The reward scheme associated with the *identifier*.

Raises KeyError: – Raised if identifier is not associated with any *RewardScheme*

tensortrade.env.default.stoppers module

class tensortrade.env.default.stoppers.**MaxLossStopper**(*max_allowed_loss: float*)
Bases: *tensortrade.env.generic.components.stopper.Stopper*

A stopper that stops an episode if the portfolio has lost a particular percentage of its wealth.

Parameters **max_allowed_loss** (*float*) – The maximum percentage of initial funds that is willing to be lost before stopping the episode.

max_allowed_loss
The maximum percentage of initial funds that is willing to be lost before stopping the episode.

Type float

Notes

This stopper also stops if it has reached the end of the observation feed.

stop(*env: tensortrade.env.generic.environment.TradingEnv*) → bool
Computes if the environment satisfies the defined stopping criteria.

Parameters **env** (*TradingEnv*) – The trading environment.

Returns *bool* – If the environment should stop or continue.

tensortrade.env.generic package

Subpackages

tensortrade.env.generic.components package

Submodules

tensortrade.env.generic.components.action_scheme module

class tensortrade.env.generic.components.action_scheme.**ActionScheme**
Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimeIndexed*

A component for determining the action to take at each step of an episode.

action_space
The action space of the *TradingEnv*. (*Space*, read-only)

perform(*env: TradingEnv*, *action: Any*) → None
Performs an action on the environment.

Parameters

- **env** (*TradingEnv*) – The trading environment to perform the *action* on.
- **action** (*Any*) – The action to perform on *env*.

registered_name = 'actions'

reset() → None
Resets the action scheme.

tensortrade.env.generic.components.informer module

class tensortrade.env.generic.components.informer.**Informer**
Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimeIndexed*

A component to provide information at each step of an episode.

info(*env: TradingEnv*) → dict
Provides information at a given step of an episode.

Parameters **env** (*'TradingEnv'*) – The trading environment.

Returns *dict* – A dictionary of information about the portfolio and net worth.

registered_name = 'monitor'

reset()
Resets the informer.

tensortrade.env.generic.components.observer module

class tensortrade.env.generic.components.observer.**Observer**
Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimeIndexed*

A component to generate an observation at each step of an episode.

observation_space
The observation space of the *TradingEnv*. (*Space*, read-only)

observe(*env: TradingEnv*) → numpy.array
Gets the observation at the current step of an episode

Parameters **env** (*'TradingEnv'*) – The trading environment.

Returns *np.array* – The current observation of the environment.

registered_name = 'observer'

reset(*random_start=0*)
Resets the observer.

tensortrade.env.generic.components.renderer module

class tensortrade.env.generic.components.renderer.**AggregateRenderer**(*renderers: List[tensortrade.env.generic.comp*
Bases: *tensortrade.env.generic.components.renderer.Renderer*

A renderer that aggregates compatible renderers so they can all be used to render a view of the environment.

Parameters **renderers** (*List[Renderer]*) – A list of renderers to aggregate.

renderers
A list of renderers to aggregate.

Type List[*Renderer*]

close() → None
Closes the renderer.

render (*env: TradingEnv*, ***kwargs*) → None
Renders a view of the environment at the current step of an episode.

Parameters

- **env** (*'TradingEnv'*) – The trading environment.
- **kwargs** (*keyword arguments*) – Additional keyword arguments for rendering the environment.

reset () → None
Resets the renderer.

save () → None
Saves the rendered view of the environment.

class tensortrade.env.generic.components.renderer.**Renderer**
Bases: *tensortrade.core.component.Component*

A component for rendering a view of the environment at each step of an episode.

close () → None
Closes the renderer.

registered_name = 'renderer'

render (*env: TradingEnv*, ***kwargs*)
Renders a view of the environment at the current step of an episode.

Parameters

- **env** (*'TradingEnv'*) – The trading environment.
- **kwargs** (*keyword arguments*) – Additional keyword arguments for rendering the environment.

reset () → None
Resets the renderer.

save () → None
Saves the rendered view of the environment.

tensortrade.env.generic.components.reward_scheme module

class tensortrade.env.generic.components.reward_scheme.**RewardScheme**
Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimeIndexed*

A component to compute the reward at each step of an episode.

registered_name = 'rewards'

reset () → None
Resets the reward scheme.

reward (*env: TradingEnv*) → float
Computes the reward for the current step of an episode.

Parameters **env** (*TradingEnv*) – The trading environment

Returns *float* – The computed reward.

tensortrade.env.generic.components.stopper module

class tensortrade.env.generic.components.stopper.**Stopper**
Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimeIndexed*

A component for determining if the environment satisfies a defined stopping criteria.

registered_name = 'stopper'

reset() → None
Resets the stopper.

stop(*env: TradingEnv*) → bool
Computes if the environment satisfies the defined stopping criteria.

Parameters **env** (*TradingEnv*) – The trading environment.

Returns *bool* – If the environment should stop or continue.

Submodules

tensortrade.env.generic.environment module

class tensortrade.env.generic.environment.**TradingEnv**(*action_scheme: tensortrade.env.generic.components.action_scheme.ActionSch*, *reward_scheme: tensortrade.env.generic.components.reward_scheme.RewardSc*, *observer: tensortrade.env.generic.components.observer.Observer*, *stopper: tensortrade.env.generic.components.stopper.Stopper*, *informer: tensortrade.env.generic.components.informer.Informer*, *renderer: tensortrade.env.generic.components.renderer.Renderer*, *min_periods: int = None*, *max_episode_steps: int = None*, *random_start_pct: float = 0.0*, ***kwargs*)
Bases: gym.core.Env, *tensortrade.core.base.TimeIndexed*

A trading environment made for use with Gym-compatible reinforcement learning algorithms.

Parameters

- **action_scheme** (*ActionScheme*) – A component for generating an action to perform at each step of the environment.
- **reward_scheme** (*RewardScheme*) – A component for computing reward after each step of the environment.
- **observer** (*Observer*) – A component for generating observations after each step of the environment.
- **informer** (*Informer*) – A component for providing information after each step of the environment.
- **renderer** (*Renderer*) – A component for rendering the environment.

- **kwargs** (*keyword arguments*) – Additional keyword arguments needed to create the environment.

agent_id = None

close() → None
Closes the environment.

components
The components of the environment. (*Dict[str,Component]*, read-only)

episode_id = None

render(**args*, ***kwargs*) → Union[RenderFrame, List[RenderFrame], None]

reset() → numpy.array
Resets the environment.

Returns **obs** (*np.array*) – The first observation of the environment.

save() → None
Saves the rendered view of the environment.

step(*action: Any*) → Tuple[numpy.array, float, bool, dict]
Makes one step through the environment.

Parameters **action** (*Any*) – An action to perform on the environment.

Returns

- *np.array* – The observation of the environment after the action being performed.
- *float* – The computed reward for performing the action.
- *bool* – Whether or not the episode is complete.
- *dict* – The information gathered after completing the step.

tensortrade.feed package

Subpackages

tensortrade.feed.api package

Subpackages

tensortrade.feed.api.boolean package

class tensortrade.feed.api.boolean.**Boolean**
Bases: object

A class to register accessor and instance methods.

classmethod register(*names: List[str]*)
A function decorator that adds accessor and instance methods for specified data type.

Parameters **names** (*List[str]*) – A list of names used to register the function as a method.

Returns *Callable* – A decorated function.

class tensortrade.feed.api.boolean.**BooleanMethods**(*stream: Stream*)
Bases: *tensortrade.feed.core.methods.Methods*

invert(**args*, ***kwargs*)

class tensortrade.feed.api.boolean.**BooleanMixin**
Bases: *tensortrade.feed.core.mixins.DataTypeMixin*

invert(**args*, ***kwargs*)

Submodules

tensortrade.feed.api.boolean.operations module

tensortrade.feed.api.boolean.operations.**invert**(*s: tensortrade.feed.core.base.Stream[bool][bool]*) → tensortrade.feed.core.base.Stream[bool][bool]

Inverts the truth value of the given stream.

Parameters **s** (*Stream[bool]*) – A boolean stream.

Returns *Stream[bool]* – An inverted stream of *s*.

tensortrade.feed.api.float package

class tensortrade.feed.api.float.**Float**
Bases: object

A class to register accessor and instance methods.

classmethod **register**(*names: List[str]*)
A function decorator that adds accessor and instance methods for specified data type.

Parameters **names** (*List[str]*) – A list of names used to register the function as a method.

Returns *Callable* – A decorated function.

class tensortrade.feed.api.float.**FloatMethods**(*stream: Stream*)
Bases: *tensortrade.feed.core.methods.Methods*

abs(**args*, ***kwargs*)

add(**args*, ***kwargs*)

ceil(**args*, ***kwargs*)

clamp(**args*, ***kwargs*)

clamp_max(**args*, ***kwargs*)

clamp_min(**args*, ***kwargs*)

cummax(**args*, ***kwargs*)

cummin(**args*, ***kwargs*)

cumprod(**args*, ***kwargs*)

cumsum(**args*, ***kwargs*)

diff(**args*, ***kwargs*)

div(**args*, ***kwargs*)

ewm(**args*, ***kwargs*)

expanding(**args*, ***kwargs*)

ffill(**args*, ***kwargs*)

fillna(**args*, ***kwargs*)

floor(**args*, ***kwargs*)

log(**args*, ***kwargs*)

max(**args*, ***kwargs*)

min(**args*, ***kwargs*)

mul(**args*, ***kwargs*)

neg(**args*, ***kwargs*)

pct_change(**args*, ***kwargs*)

pow(**args*, ***kwargs*)

radd(**args*, ***kwargs*)

rdiv(**args*, ***kwargs*)

rmul(**args*, ***kwargs*)

rolling(**args*, ***kwargs*)

rsub(**args*, ***kwargs*)

sqrt(**args*, ***kwargs*)

square(**args*, ***kwargs*)

sub(**args*, ***kwargs*)

class tensortrade.feed.api.float.**FloatMixin**

Bases: *tensortrade.feed.core.mixins.DataTypeMixin*

abs(**args*, ***kwargs*)

add(**args*, ***kwargs*)

ceil(**args*, ***kwargs*)

clamp(**args*, ***kwargs*)

clamp_max(**args*, ***kwargs*)

clamp_min(**args*, ***kwargs*)

cummax(**args*, ***kwargs*)

cummin(**args*, ***kwargs*)

cumprod(**args*, ***kwargs*)

cumsum(**args*, ***kwargs*)

diff(**args*, ***kwargs*)

div(**args*, ***kwargs*)

ewm(**args*, ***kwargs*)

expanding(**args*, ***kwargs*)

ffill(**args*, ***kwargs*)

fillna(**args*, ***kwargs*)

floor(**args*, ***kwargs*)

log(**args*, ***kwargs*)

max(**args*, ***kwargs*)

min(**args*, ***kwargs*)

mul(**args*, ***kwargs*)

neg(**args*, ***kwargs*)

pct_change(**args*, ***kwargs*)

pow(**args*, ***kwargs*)

radd(**args*, ***kwargs*)

rdiv(**args*, ***kwargs*)

rmul(**args*, ***kwargs*)

rolling(**args*, ***kwargs*)

rsub(**args*, ***kwargs*)

sqrt(**args*, ***kwargs*)

square(**args*, ***kwargs*)

sub(**args*, ***kwargs*)

Subpackages

tensortrade.feed.api.float.window package

Submodules

tensortrade.feed.api.float.window.ewm module

ewm.py contains functions and classes for exponential weighted moving stream operations.

class tensortrade.feed.api.float.window.ewm.**EWM**(*com: float = None*, *span: float = None*, *halflife: float = None*, *alpha: float = None*, *min_periods: int = 0*, *adjust: bool = True*, *ignore_na: bool = False*)

Bases: *tensortrade.feed.core.base.Stream*

Provide exponential weighted (EW) functions.

Exactly one parameter: *com*, *span*, *halflife*, or *alpha* must be provided.

Parameters

- **com** (*float, optional*) – Specify decay in terms of center of mass, $\alpha = 1/(1+com)$, for $com \geq 0$.
- **span** (*float, optional*) – Specify decay in terms of span, $\alpha = 2/(span+1)$, for $span \geq 1$.

- **halflife** (*float, str, timedelta, optional*) – Specify decay in terms of half-life, $\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$. If `times` is specified, the time unit (str or timedelta) over which an observation decays to half its value. Only applicable to `mean()` and halflife value will not apply to the other functions.
- **alpha** (*float, optional*) – Specify smoothing factor α directly, $0 < \alpha \leq 1$.
- **min_periods** (*int, default 0*) – Minimum number of observations in window required to have a value (otherwise result is NA).
- **adjust** (*bool, default True*) – Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average). - When `adjust=True` (default), the EW function is calculated using weights

  $w_i = (1 - \alpha)^i$. For example, the EW moving average of the series $[x_0, x_1, ..., x_t]$ would be:

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ... + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^t}$$

  – When `adjust=False`, the exponentially weighted function is calculated recursively:

$$\begin{aligned} y_0 &= x_0 \\ y_t &= (1-\alpha)y_{t-1} + \alpha x_t, \end{aligned}$$

- **ignore_na** (*bool, default False*) – Ignore missing values when calculating weights. - When `ignore_na=False` (default), weights are based on absolute positions. - When `ignore_na=True`, weights are based on relative positions.

See also:

References

forward() → Tuple[List[float], List[float]]
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

mean() → tensortrade.feed.core.base.Stream[float][float]
Computes the exponential weighted moving average.

Returns *Stream[float]* – The exponential weighted moving average stream based on the underlying stream of values.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

std(*bias=False*) → tensortrade.feed.core.base.Stream[float][float]
Computes the exponential weighted moving standard deviation.

Returns *Stream[float]* – The exponential weighted moving standard deviation stream based on the underlying stream of values.

var(*bias=False*) → tensortrade.feed.core.base.Stream[float][float]
Computes the exponential weighted moving variance.

Returns *Stream[float]* – The exponential weighted moving variance stream based on the underlying stream of values.

class tensortrade.feed.api.float.window.ewm.**ExponentialWeightedMovingAverage**(*alpha: float*, *adjust: bool*, *ignore_na: bool*, *min_periods: int*)

Bases: *tensortrade.feed.core.base.Stream*

A stream operator that computes an exponential weighted moving average on a given float stream.

Parameters

- **alpha** (*float*) – The smoothing factor α directly, $0 < \alpha \leq 1$.
- **adjust** (*bool*) – Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).
- **ignore_na** (*bool*) – Ignore missing values when calculating weights.
- **min_periods** (*int*) – Minimum number of observations in window required to have a value (otherwise result is NA).

References

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

class tensortrade.feed.api.float.window.ewm.**ExponentialWeightedMovingCovariance**(*alpha: float*, *adjust: bool*, *ignore_na: bool*, *min_periods: int*, *bias: bool*)

Bases: *tensortrade.feed.core.base.Stream*

A stream operator that computes an exponential weighted moving average on a given float stream.

Parameters

- **alpha** (*float*) – The smoothing factor α directly, $0 < \alpha \leq 1$.
- **adjust** (*bool*) – Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).
- **ignore_na** (*bool*) – Ignore missing values when calculating weights.
- **min_periods** (*int*) – Minimum number of observations in window required to have a value (otherwise result is NA).
- **bias** (*bool*) – Use a standard estimation bias correction

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

tensortrade.feed.api.float.window.ewm.**ewm**(*s: tensortrade.feed.core.base.Stream[float][float], com: float = None, span: float = None, halflife: float = None, alpha: float = None, min_periods: int = 0, adjust: bool = True, ignore_na: bool = False*) → tensortrade.feed.core.base.Stream[typing.Tuple[typing.List[float], typing.List[float]]][Tuple[List[float], List[float]]]

Computes the weights and values in order to perform an exponential weighted moving operation.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **com** (*float, optional*) – Specify decay in terms of center of mass, $\alpha = 1/(1+com)$, for $com \geq 0$.
- **span** (*float, optional*) – Specify decay in terms of span, $\alpha = 2/(span+1)$, for $span \geq 1$.
- **halflife** (*float, optional*) – Specify decay in terms of half-life, $\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$.
- **alpha** (*float, optional*) – Specify smoothing factor α directly, $0 < \alpha \leq 1$.
- **min_periods** (*int, default 0*) – Minimum number of observations in window required to have a value (otherwise result is NA).
- **adjust** (*bool, default True*) – Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).
- **ignore_na** (*bool, default False*) – Ignore missing values when calculating weights.

Returns *Stream[Tuple[List[float], List[float]]]* – A stream of weights and values to be used for computation of exponential weighted moving operations.

tensortrade.feed.api.float.window.expanding module

expanding.py contains functions and classes for expanding stream operations.

class tensortrade.feed.api.float.window.expanding.**Expanding**(*min_periods: int = 1*)

Bases: *tensortrade.feed.core.base.Stream*

A stream that generates the entire history of a stream at each time step.

Parameters **min_periods** (*int, default 1*) – The number of periods to wait before producing values from the aggregation function.

agg(*func: Callable[[List[float]], float]*) → tensortrade.feed.core.base.Stream[float][float]
Computes an aggregation of a stream's history.

Parameters **func** (*Callable[[List[float]], float]*) – A aggregation function.

Returns *Stream[float]* – A stream producing aggregations of the stream history at each time step.

count() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding count fo the underlying stream.

Returns *Stream[float]* – An expanding count stream.

forward() → List[float]
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'expanding'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

max() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding maximum fo the underlying stream.

Returns *Stream[float]* – An expanding maximum stream.

mean() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding mean fo the underlying stream.

Returns *Stream[float]* – An expanding mean stream.

median() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding median fo the underlying stream.

Returns *Stream[float]* – An expanding median stream.

min() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding minimum fo the underlying stream.

Returns *Stream[float]* – An expanding minimum stream.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

std() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding standard deviation fo the underlying stream.

Returns *Stream[float]* – An expanding standard deviation stream.

sum() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding sum fo the underlying stream.

Returns *Stream[float]* – An expanding sum stream.

var() → tensortrade.feed.core.base.Stream[float][float]
Computes an expanding variance fo the underlying stream.

Returns *Stream[float]* – An expanding variance stream.

class tensortrade.feed.api.float.window.expanding.**ExpandingCount**
Bases: *tensortrade.feed.api.float.window.expanding.ExpandingNode*

A stream operator that counts the number of non-missing values.

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

class tensortrade.feed.api.float.window.expanding.**ExpandingNode**(*func: Callable[[List[float]], float]*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator for aggregating an entire history of a stream.

Parameters **func** (*Callable[[List[float]], float]*) – A function that aggregates the history of a stream.

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next()
Checks if there is another value.

Returns *bool* – If there is another value or not.

tensortrade.feed.api.float.window.expanding.**expanding**(*s: tensortrade.feed.core.base.Stream[float][float], min_periods: int = 1*) → tensortrade.feed.core.base.Stream[typing.List[float]][List[flo
Computes a stream that generates the entire history of a stream at each time step.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **min_periods** (*int, default 1*) – The number of periods to wait before producing values from the aggregation function.

tensortrade.feed.api.float.window.rolling module

rolling.py contains functions and classes for rolling stream operations.

class tensortrade.feed.api.float.window.rolling.**Rolling**(*window: int, min_periods: int = 1*)
Bases: *tensortrade.feed.core.base.Stream*

A stream that generates a rolling window of values from a stream.

Parameters

- **window** (*int*) – The size of the rolling window.
- **min_periods** (*int, default 1*) – The number of periods to wait before producing values from the aggregation function.

agg (*func: Callable[[List[float]], float]*) → tensortrade.feed.core.base.Stream[float][float]
Computes an aggregation of a rolling window of values.

Parameters **func** (*Callable[[List[float]], float]*) – A aggregation function.

Returns *Stream[float]* – A stream producing aggregations of a rolling window of values.

count () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling count from the underlying stream.

Returns *Stream[float]* – A rolling count stream.

forward () → List[float]
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'rolling'

has_next () → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

max () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling maximum from the underlying stream.

Returns *Stream[float]* – A rolling maximum stream.

mean () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling mean from the underlying stream.

Returns *Stream[float]* – A rolling mean stream.

median () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling median from the underlying stream.

Returns *Stream[float]* – A rolling median stream.

min () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling minimum from the underlying stream.

Returns *Stream[float]* – A rolling minimum stream.

reset () → None
Resets all inputs to and listeners of the stream and sets stream value to None.

std () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling standard deviation from the underlying stream.

Returns *Stream[float]* – A rolling standard deviation stream.

sum () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling sum from the underlying stream.

Returns *Stream[float]* – A rolling sum stream.

var () → tensortrade.feed.core.base.Stream[float][float]
Computes a rolling variance from the underlying stream.

Returns *Stream[float]* – A rolling variance stream.

class tensortrade.feed.api.float.window.rolling.**RollingCount**
Bases: *tensortrade.feed.api.float.window.rolling.RollingNode*

A stream operator that counts the number of non-missing values in the rolling window.

forward()
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

class tensortrade.feed.api.float.window.rolling.**RollingNode**(*func: Callable[[List[float]], float]*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator for aggregating a rolling window of a stream.

Parameters **func** (*Callable[[List[float]], float]*) – A function that aggregates a rolling window.

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

tensortrade.feed.api.float.window.rolling.**rolling**(*s: tensortrade.feed.core.base.Stream[float][float], window: int, min_periods: int = 1*) → tensortrade.feed.core.base.Stream[typing.List[float]][List[float]]
Creates a stream that generates a rolling window of values from a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **window** (*int*) – The size of the rolling window.
- **min_periods** (*int, default 1*) – The number of periods to wait before producing values from the aggregation function.

Submodules

tensortrade.feed.api.float.accumulators module

class tensortrade.feed.api.float.accumulators.**CumMax**(*skip_na: bool = True*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that creates a cumulative maximum of values.

Parameters **skip_na** (*bool, default True*) – Exclude NA/null values. If a value is NA, the result will be NA.

References

[1] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.cummax.html

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.api.float.accumulators.**CumMin**(*skip_na: bool = True*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that creates a cumulative minimum of values.

Parameters **skip_na** (*bool, default True*) – Exclude NA/null values. If a value is NA, the result will be NA.

References

[1] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.cummin.html

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.api.float.accumulators.**CumProd**
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that creates a cumulative product of values.

References

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.api.float.accumulators.**CumSum**
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that creates a cumulative sum of values.

References

forward() → float
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

tensortrade.feed.api.float.accumulators.**cummax**(*s: tensortrade.feed.core.base.Stream[float][float], skipna: bool = True*) → tensortrade.feed.core.base.Stream[float][float]

Computes the cumulative maximum of a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **skipna** (*bool, default True*) – Exclude NA/null values. If a value is NA, the result will be NA.

Returns *Stream[float]* – The cumulative maximum stream of *s*.

tensortrade.feed.api.float.accumulators.**cummin**(*s: tensortrade.feed.core.base.Stream[float][float], skipna: bool = True*) → tensortrade.feed.core.base.Stream[float][float]

Computes the cumulative minimum of a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **skipna** (*bool, default True*) – Exclude NA/null values. If a value is NA, the result will be NA.

Returns *Stream[float]* – The cumulative minimum stream of *s*.

tensortrade.feed.api.float.accumulators.**cumprod**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the cumulative product of a stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The cumulative product stream of *s*.

tensortrade.feed.api.float.accumulators.**cumsum**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the cumulative sum of a stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The cumulative sum stream of *s*.

tensortrade.feed.api.float.imputation module

tensortrade.feed.api.float.imputation.**ffill**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Fill in missing values by forward filling.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – An imputed stream via forward filling.

tensortrade.feed.api.float.imputation.**fillna**(*s: tensortrade.feed.core.base.Stream[float][float], fill_value: float = 0.0*) → tensortrade.feed.core.base.Stream[float][float]

Fill in missing values with a fill value.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **fill_value** (*float*) – A value to fill in missing values with.

Returns *Stream[float]* – An imputed stream via padding.

tensortrade.feed.api.float.operations module

operations.py contains functions for computing arithmetic operations on float streams.

tensortrade.feed.api.float.operations.**abs**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the absolute value of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The absolute value stream of *s*.

tensortrade.feed.api.float.operations.**add**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the addition of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from adding *s1* and *s2*.

tensortrade.feed.api.float.operations.**mul**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the multiplication of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from multiplying *s1* and *s2*.

tensortrade.feed.api.float.operations.**neg**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the negation of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The negated stream of *s*.

tensortrade.feed.api.float.operations.**pow**(*s: tensortrade.feed.core.base.Stream[float][float]*, *power: float*) → tensortrade.feed.core.base.Stream[float][float]

Computes the power of a float stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **power** (*float*) – The power to raise *s* by.

Returns *Stream[float]* – The power stream of *s*.

tensortrade.feed.api.float.operations.**radd**(*s1: tensortrade.feed.core.base.Stream[float][float]*, *s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the reversed addition of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from adding *s1* and *s2*.

tensortrade.feed.api.float.operations.**rmul**(*s1: tensortrade.feed.core.base.Stream[float][float]*, *s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the reverse multiplication of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from multiplying *s2* and *s1*.

tensortrade.feed.api.float.operations.**rsub**(*s1: tensortrade.feed.core.base.Stream[float][float]*, *s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the reverse subtraction of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from subtracting *s1* from *s2*.

tensortrade.feed.api.float.operations.**rtruediv**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the reverse division of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from dividing *s2* by *s1*.

tensortrade.feed.api.float.operations.**sub**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the subtraction of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from subtracting *s2* from *s1*.

tensortrade.feed.api.float.operations.**truediv**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the division of two float streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]* or float) – The second float stream.

Returns *Stream[float]* – A stream created from dividing *s1* by *s2*.

tensortrade.feed.api.float.ordering module

tensortrade.feed.api.float.ordering.**clamp**(*s: tensortrade.feed.core.base.Stream[float][float], c_min: float, c_max: float*) → tensortrade.feed.core.base.Stream[float][float]

Clamps the minimum and maximum value of a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **c_min** (*float*) – The mimimum value to clamp by.
- **c_max** (*float*) – The maximum value to clamp by.

Returns *Stream[float]* – The clamped stream of *s*.

tensortrade.feed.api.float.ordering.**clamp_max**(*s: tensortrade.feed.core.base.Stream[float][float], c_max: float*) → tensortrade.feed.core.base.Stream[float][float]

Clamps the maximum value of a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **c_max** (*float*) – The maximum value to clamp by.

Returns *Stream[float]* – The maximum clamped stream of *s*.

tensortrade.feed.api.float.ordering.**clamp_min**(*s: tensortrade.feed.core.base.Stream[float][float], c_min: float*) → tensortrade.feed.core.base.Stream[float][float]

Clamps the minimum value of a stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **c_min** (*float*) – The mimimum value to clamp by.

Returns *Stream[float]* – The minimum clamped stream of *s*.

tensortrade.feed.api.float.ordering.**max**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the maximum of two streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]*) – The second float stream.

Returns *Stream[float]* – The maximum stream of *s1* and *s2*.

tensortrade.feed.api.float.ordering.**min**(*s1: tensortrade.feed.core.base.Stream[float][float], s2: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the minimum of two streams.

Parameters

- **s1** (*Stream[float]*) – The first float stream.
- **s2** (*Stream[float]*) – The second float stream.

Returns *Stream[float]* – The minimum stream of *s1* and *s2*.

tensortrade.feed.api.float.utils module

tensortrade.feed.api.float.utils.**ceil**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the ceiling of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The ceiling stream of *s*.

tensortrade.feed.api.float.utils.**diff**(*s: tensortrade.feed.core.base.Stream[float][float], periods: int = 1*) → tensortrade.feed.core.base.Stream[float][float]

Computes the difference of a float stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **periods** (*int, default 1*) – The number of periods to lag for until computing the difference.

Returns *Stream[float]* – The difference stream of *s*.

tensortrade.feed.api.float.utils.**floor**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the floor of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The floor stream of *s*.

tensortrade.feed.api.float.utils.**log**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the log of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The log stream of *s*.

tensortrade.feed.api.float.utils.**pct_change**(*s: tensortrade.feed.core.base.Stream[float][float], periods: int = 1, fill_method: str = 'pad'*) → tensortrade.feed.core.base.Stream[float][float]

Computes the percent change of a float stream.

Parameters

- **s** (*Stream[float]*) – A float stream.
- **periods** (*int, default 1*) – The number of periods to lag for until computing the percent change.
- **fill_method** (*str, default "pad"*) – The fill method to use for missing values.

Returns *Stream[float]* – The percent change stream of *s*.

tensortrade.feed.api.float.utils.**sqrt**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the square root of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The square root stream of *s*.

tensortrade.feed.api.float.utils.**square**(*s: tensortrade.feed.core.base.Stream[float][float]*) → tensortrade.feed.core.base.Stream[float][float]

Computes the square of a float stream.

Parameters **s** (*Stream[float]*) – A float stream.

Returns *Stream[float]* – The square stream of *s*.

tensortrade.feed.api.generic package

Submodules

tensortrade.feed.api.generic.imputation module

imputation.py contains classes for imputation stream operations.

class tensortrade.feed.api.generic.imputation.**FillNa**(*fill_value: T*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that computes the padded imputation of a stream.

Parameters **fill_value** (*T*) – The fill value to use for missing values in the stream.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'fillna'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.api.generic.imputation.**ForwardFill**
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that computes the forward fill imputation of a stream.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'ffill'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

tensortrade.feed.api.generic.operators module

operators.py contains function for generic stream operators.

tensortrade.feed.api.generic.reduce module

reduce.py contains functions and classes for reducing multiple streams into a single stream.

class tensortrade.feed.api.generic.reduce.**Aggregate**(*func: Callable[[List[T]], T]*)
Bases: *tensortrade.feed.core.base.Stream*

A multi-stream operator for aggregating multiple streams into a single stream.

Parameters **func** (*Callable[[List[Stream]], T]*) – A function for aggregating the value of multiple streams.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'reduce'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.api.generic.reduce.**Reduce**(*dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

A stream for reducing multiple streams of the same type.

Parameters **dtype** (*str, optional*) – The data type of the aggregated stream.

agg(*func: Callable[[List[T]], T]*) → tensortrade.feed.core.base.Stream[~T][T]
Computes the aggregation of the input streams.

Returns *Stream[T]* – An aggregated stream of the input streams.

forward() → List[T]
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

max() → tensortrade.feed.core.base.Stream[~T][T]
Computes the reduced maximum of the input streams.

Returns *Stream[T]* – A reduced maximum stream.

min() → tensortrade.feed.core.base.Stream[~T][T]
Computes the reduced minimum of the input streams.

Returns *Stream[T]* – A reduced minimum stream.

prod() → tensortrade.feed.core.base.Stream[~T][T]
Computes the reduced product of the input streams.

Returns *Stream[T]* – A reduced product stream.

sum() → tensortrade.feed.core.base.Stream[~T][T]
Computes the reduced sum of the input streams.

Returns *Stream[T]* – A reduced sum stream.

tensortrade.feed.api.generic.warmup module

warmup.py contains classes for warm up stream operations.

class tensortrade.feed.api.generic.warmup.**WarmUp**(*periods: int*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator for warming up a given stream.

Parameters **periods** (*int*) – Number of periods to warm up.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

tensortrade.feed.api.string package

class tensortrade.feed.api.string.**String**
Bases: object

A class to register accessor and instance methods.

classmethod register(*names: List[str]*)
A function decorator that adds accessor and instance methods for specified data type.

Parameters **names** (*List[str]*) – A list of names used to register the function as a method.

Returns *Callable* – A decorated function.

class tensortrade.feed.api.string.**StringMethods**(*stream: Stream*)
Bases: *tensortrade.feed.core.methods.Methods*

capitalize(**args*, ***kwargs*)

cat(**args*, ***kwargs*)

endswith(**args*, ***kwargs*)

lower(**args*, ***kwargs*)

slice(**args*, ***kwargs*)

startswith(**args*, ***kwargs*)

upper(**args*, ***kwargs*)

class tensortrade.feed.api.string.**StringMixin**
Bases: *tensortrade.feed.core.mixins.DataTypeMixin*

capitalize(**args*, ***kwargs*)

cat(**args*, ***kwargs*)

endswith(**args*, ***kwargs*)

lower(**args*, ***kwargs*)

slice(**args*, ***kwargs*)

startswith(**args*, ***kwargs*)

upper(**args*, ***kwargs*)

Submodules

tensortrade.feed.api.string.operations module

operations.py contain functions for streaming string operations.

tensortrade.feed.api.string.operations.**capitalize**(*s: tensortrade.feed.core.base.Stream[str][str]*) → tensortrade.feed.core.base.Stream[str][str]

Computes the capitalization of a stream.

Parameters **s** (*Stream[str]*) – A string stream.

Returns *Stream[str]* – A capitalized string stream.

tensortrade.feed.api.string.operations.**cat**(*s: tensortrade.feed.core.base.Stream[str][str], word: str*) → tensortrade.feed.core.base.Stream[str][str]

Computes the concatenation of a stream with a word.

Parameters

- **s** (*Stream[str]*) – A string stream.
- **word** (*str*) – A word to concatenate with the *s*.

Returns *Stream[str]* – A concatenated string stream.

tensortrade.feed.api.string.operations.**endswith**(*s: tensortrade.feed.core.base.Stream[str][str], word: str*) → tensortrade.feed.core.base.Stream[bool][bool]

Computes the boolean stream of a string ending with a specific value.

Parameters

- **s** (*Stream[str]*) – A string stream.
- **word** (*str*) – A word that a string value can end with.

Returns *Stream[bool]* – A boolean stream.

tensortrade.feed.api.string.operations.**lower**(*s: tensortrade.feed.core.base.Stream[str][str]*) → tensortrade.feed.core.base.Stream[str][str]

Computes the lowercase of a string stream.

Parameters **s** (*Stream[str]*) – A string stream.

Returns *Stream[str]* – A lowercase string stream.

tensortrade.feed.api.string.operations.**slice**(*s: tensortrade.feed.core.base.Stream[str][str], start: int, end: int*) → tensortrade.feed.core.base.Stream[str][str]

Computes the substring of a string stream.

Parameters

- **s** (*Stream[str]*) – A string stream.
- **start** (*int*) – The start of the slice.
- **end** (*int*) – The end of the slice.

Returns *Stream[str]* – A substring stream.

tensortrade.feed.api.string.operations.**startswith**(*s: tensortrade.feed.core.base.Stream[str][str], word: str*) → tensortrade.feed.core.base.Stream[bool][bool]

Computes the boolean stream of a string starting with a specific value.

Parameters

- **s** (*Stream[str]*) – A string stream.
- **word** (*str*) – A word that a string value can start with.

Returns *Stream[bool]* – A boolean stream.

tensortrade.feed.api.string.operations.**upper**(*s: tensortrade.feed.core.base.Stream[str][str]*) → tensortrade.feed.core.base.Stream[str][str]

Computes the uppercase of a string stream.

Parameters **s** (*Stream[str]*) – A string stream.

Returns *Stream[str]* – A uppercase string stream.

tensortrade.feed.core package

Submodules

tensortrade.feed.core.accessors module

class tensortrade.feed.core.accessors.**CachedAccessor**(*name: str*, *accessor*)

Bases: object

Custom property-like object.

A descriptor for caching accessors.

Parameters

- **name** (*str*) – Namespace that will be accessed under, e.g. `df.foo`.
- **accessor** (*cls*) – Class with the extension methods.

References

tensortrade.feed.core.base module

class tensortrade.feed.core.base.**Constant**(*value*, *dtype: str = None*)

Bases: *tensortrade.feed.core.base.Stream*

A stream that generates a constant value.

forward()

Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'constant'

has_next()
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.base.**Group**
Bases: *tensortrade.feed.core.base.Stream*

A stream that groups together other streams into a dictionary.

forward() → Dict[T]
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.base.**IterableStream**(*source: Iterable[T], dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

A private class used the *Stream* class for creating data sources.

Parameters

- **source** (*Iterable[T]*) – The iterable to be used for providing the data.
- **dtype** (*str, optional*) – The data type of the source.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'stream'

has_next()
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset(*random_start=0*)
Resets all inputs to and listeners of the stream and sets stream value to None.

class tensortrade.feed.core.base.**NameSpace**(*name: str*)
Bases: *tensortrade.feed.core.base.Named*

A class providing a context in which to create names.

This becomes useful in cases where *Named* object would like to use the same name in a different context. In order to resolve naming conflicts in a *DataFeed*, this class provides a way to solve it.

Parameters **name** (*str*) – The name for the *NameSpace*.

class tensortrade.feed.core.base.**Named**(*name: str = None*)
Bases: object

A class for controlling the naming of objects.

The purpose of this class is to control the naming of objects with respect to the *NameSpace* to which they belong to. This prevents conflicts that arise in the naming of similar objects under different contexts.

Parameters **name** (*str, optional*) – The name of the object.

name
The name of the object.

Type str, optional

generic_name = 'generic'

names = {}

namespaces = []

rename(*name: str*, *sep: str = ':/'*) → tensortrade.feed.core.base.Named
Renames the instance with respect to the current *NameSpace*.

Parameters

- **name** (*str*) – The new name to give to the instance.
- **sep** (*str*) – The separator to put between the name of the *NameSpace* and the new name of the instance (e.g. ns:/example).

Returns *Named* – The instance that was renamed.

class tensortrade.feed.core.base.**Placeholder**(*dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

A stream that acts as a placeholder for data to be provided at later date.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'placeholder'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

push(*value: T*) → None

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

class tensortrade.feed.core.base.**Sensor**(*obj*, *func*, *dtype=None*)
Bases: *tensortrade.feed.core.base.Stream*

A stream that watches and generates from a particular object.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'sensor'

has_next()
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.base.**Stream**(*name: str = None*, *dtype: str = None*)
Bases: typing.Generic, *tensortrade.feed.core.base.Named*, *tensortrade.core.base.Observable*

A class responsible for creating the inputs necessary to work in a *DataFeed*.

Parameters

- **name** (*str, optional*) – The name fo the stream.
- **dtype** (*str, optional*) – The data type of the stream.

source (*iterable*, *dtype=None*)
Creates a stream from an iterable.

group (*streams*)
Creates a group of streams.

sensor (*obj*, *func*, *dtype=None*)
Creates a stream from observing a value from an object.

select (*streams*, *func*)
Selects a stream satisfying particular criteria from a list of streams.

constant (*value*, *dtype*)
Creates a stream to generate a constant value.

asdtype (*dtype*)
Converts the data type to *dtype*.

__call__ (**inputs*) → tensortrade.feed.core.base.Stream[~T][T]
Connects the inputs to this stream.

Parameters ***inputs** (*positional arguments*) – The positional arguments, each a stream to be connected as an input to this stream.

Returns *Stream[T]* – The current stream inputs are being connected to.

accumulate (**args*, ***kwargs*)

apply (**args*, ***kwargs*)

astype (*dtype: str*) → tensortrade.feed.core.base.Stream[~T][T]
Converts the data type to *dtype*.

Parameters **dtype** (*str*) – The data type to be converted to.

Returns *Stream[T]* – The same stream with the new underlying data type *dtype*.

bool
alias of *tensortrade.feed.api.boolean.BooleanMethods*

static constant (*value: T*, *dtype: str = None*) → tensortrade.feed.core.base.Stream[~T][T]
Creates a stream to generate a constant value.

Parameters

- **value** (*T*) – The constant value to be streamed.
- **dtype** (*str, optional*) – The data type of the value.

Returns *Stream[T]* – A stream of the constant value.

copy (**args*, ***kwargs*)

static extend_instance (*instance: tensortrade.feed.core.base.Stream[~T][T], mixin: tensortrade.feed.core.mixins.DataTypeMixin*) → tensortrade.feed.core.base.Stream[~T][T]
Apply mix-ins to a class instance after creation.

Parameters

- **instance** (*Stream[T]*) – An instantiation of *Stream* to be injected with mixin methods.

- **mixin** (*DataTypeMixin*) – The mixin holding the methods to be injected into the *instance*.

Returns *Stream[T]* – The *instance* with the injected methods provided by the *mixin*.

float
alias of *tensortrade.feed.api.float.FloatMethods*

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

freeze(**args*, ***kwargs*)

gather() → List[Tuple[tensortrade.feed.core.base.Stream, tensortrade.feed.core.base.Stream]]
Gathers all the edges of the DAG connected in ancestry with this stream.

Returns *List[Tuple[Stream, Stream]]* – The list of edges connected through ancestry to this stream.

generic_name = 'stream'

static group(*streams: List[tensortrade.feed.core.base.Stream[~T][T]]*) → tensortrade.feed.core.base.Stream[dict][dict]
Creates a group of streams.

Parameters **streams** (*List[Stream[T]]*) – Streams to be grouped together.

Returns *Stream[dict]* – A stream of dictionaries with each stream as a key/value in the dictionary being generated.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

lag(**args*, ***kwargs*)

static placeholder(*dtype: str = None*) → tensortrade.feed.core.base.Stream[~T][T]
Creates a placholder stream for data to provided to at a later date.

Parameters **dtype** (*str*) – The data type that will be provided.

Returns *Stream[T]* – A stream representing a placeholder.

reduce(**args*, ***kwargs*)

classmethod register_accessor(*name: str*)
A class decorator that registers an accessor providing useful methods for a particular data type..

Sets the data type accessor to be an attribute of this class.

Parameters **name** (*str*) – The name of the data type.

classmethod register_generic_method(*names: List[str]*)
A function decorator that registers the decorated function with the names provided as a method to the *Stream* class.

These methods can be used for any instance of *Stream*.

Parameters **names** (*List[str]*) – The list of names to be used as aliases for the same method.

classmethod register_mixin(*dtype: str*)
A class decorator the registers a data type mixin providing useful methods directly to the instance of the class.

Parameters **dtype** (*str*) – The name of the data type the mixin is being registered for.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

run() → None
Runs the underlying streams once and iterates forward.

static select(*streams: List[tensortrade.feed.core.base.Stream[~T][T]], func: Callable[[tensortrade.feed.core.base.Stream[~T][T]], bool]*) → tensortrade.feed.core.base.Stream[~T][T]
Selects a stream satisfying particular criteria from a list of streams.

Parameters

- **streams** (*List[Stream[T]]*) – A list of streams to select from.
- **func** (*Callable[[Stream[T]], bool]*) – The criteria to be used for finding the particular stream.

Returns *Stream[T]* – The particular stream being selected.

Raises Exception – Raised if no stream is found to satisfy the given criteria.

static sensor(*obj: Any, func: Callable[[Any], T], dtype: str = None*) → tensortrade.feed.core.base.Stream[~T][T]
Creates a stream from observing a value from an object.

Parameters

- **obj** (*Any*) – An object to observe values from.
- **func** (*Callable[[Any], T]*) – A function to extract the data to be observed from the object being watched.
- **dtype** (*str, optional*) – The data type of the stream.

Returns *Stream[T]* – The stream of values being observed from the object.

static source(*iterable: Iterable[T], dtype: str = None*) → tensortrade.feed.core.base.Stream[~T][T]
Creates a stream from an iterable.

Parameters

- **iterable** (*Iterable[T]*) – The iterable to create the stream from.
- **dtype** (*str, optional*) – The data type of the stream.

Returns *Stream[T]* – The stream with the data type *dtype* created from *iterable*.

str
alias of *tensortrade.feed.api.string.StringMethods*

static toposort(*edges: List[Tuple[tensortrade.feed.core.base.Stream, tensortrade.feed.core.base.Stream]]*) → List[tensortrade.feed.core.base.Stream]
Sorts the order in which streams should be run.

Parameters **edges** (*List[Tuple[Stream, Stream]]*) – The connections that have been found in the DAG.

Returns *List[Stream]* – The list of streams sorted with respect to the order in which they should be run.

warmup(**args*, ***kwargs*)

tensortrade.feed.core.feed module

class tensortrade.feed.core.feed.**DataFeed**(*streams: List[tensortrade.feed.core.base.Stream]*)
Bases: *tensortrade.feed.core.base.Stream*

A stream the compiles together streams to be run in an organized manner.

Parameters **streams** (*List[Stream]*) – A list of streams to be used in the data feed.

compile() → None
Compiles all the given stream together.

Organizes the order in which streams should be run to get valid output.

forward() → dict
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

next() → dict

reset(*random_start=0*) → None
Resets all inputs to and listeners of the stream and sets stream value to None.

run() → None
Runs all the streams in processing order.

class tensortrade.feed.core.feed.**PushFeed**(*streams: List[tensortrade.feed.core.base.Stream]*)
Bases: *tensortrade.feed.core.feed.DataFeed*

A data feed for working with live data in an online manner.

All sources of data to be used with this feed must be a *Placeholder*. This ensures that the user can wait until all of their data has been loaded for the next time step.

Parameters **streams** (*List[Stream]*) – A list of streams to be used in the data feed.

is_loaded

next() → dict

push(*data: dict*) → dict
Generates the values from the data feed based on the values being provided in *data*.

Parameters **data** (*dict*) – The data to be pushed to each of the placholders in the feed.

Returns *dict* – The next data point generated from the feed based on *data*.

tensortrade.feed.core.methods module

class tensortrade.feed.core.methods.**Methods**(*stream: Stream*)
Bases: object

A class used to hold the accessor methods for a particular data type.

Parameters **stream** ("*Stream*") – The stream to injected with the method accessor.

classmethod **register_method**(*func: Callable, names: List[str]*)
Injects an accessor into a specific stream instance.

Parameters

- **func** (*Callable*) – The function to be injected as an accessor method.
- **names** (*List[str]*) – The names to be given to the function.

tensortrade.feed.core.mixins module

class tensortrade.feed.core.mixins.**DataTypeMixin**
Bases: object

classmethod register_method(*func: Callable, names: List[str]*)
Injects methods into a specific stream instance.

Parameters

- **func** (*Callable*) – The function to be injected as a method.
- **names** (*List[str]*) – The names to be given to the function.

tensortrade.feed.core.operators module

class tensortrade.feed.core.operators.**Accumulator**(*func: Callable[[T, T], T], dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

An operator stream that accumulates values of a given stream.

Parameters

- **func** (*Callable[[T,T], T]*) – An accumulator function.
- **dtype** (*str*) – The data type of accumulated value.

forward()
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

class tensortrade.feed.core.operators.**Apply**(*func: Callable[[T], K], dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

An operator stream that applies a specific function to the values of a given stream.

Parameters

- **func** (*Callable[[T], ...]*) – A function to be applied to the values of a stream.
- **dtype** (*str, optional*) – The data type of the values after function is applied.

forward() → K
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.operators.**BinOp**(*op: Callable[[T, T], T], dtype: str = None*)
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that combines the values of two given streams into one value of the same type.

Parameters

- **op** (*Callable[[T, T], T]*) – The binary operation to be applied.
- **dtype** (*str, optional*) – The data type of the stream.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'bin_op'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.operators.**Copy**
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that copies the values of a given stream.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'copy'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

class tensortrade.feed.core.operators.**Freeze**
Bases: *tensortrade.feed.core.base.Stream*

A stream operator that freezes the value of a given stream and generates that value.

forward() → T
Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'freeze'

has_next() → bool
Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None
Resets all inputs to and listeners of the stream and sets stream value to None.

class tensortrade.feed.core.operators.**Lag**(*lag: int = 1*, *dtype: str = None*)

Bases: *tensortrade.feed.core.base.Stream*

An operator stream that returns the lagged value of a given stream.

Parameters

- **lag** (*int*) – The number of steps to lag behind by
- **dtype** (*str, optional*) – The data type of the stream

forward() → T

Generates the next value from the underlying data streams.

Returns *T* – The next value in the stream.

generic_name = 'lag'

has_next() → bool

Checks if there is another value.

Returns *bool* – If there is another value or not.

reset() → None

Resets all inputs to and listeners of the stream and sets stream value to None.

tensortrade.oms package

Subpackages

tensortrade.oms.exchanges package

Submodules

tensortrade.oms.exchanges.exchange module

class tensortrade.oms.exchanges.exchange.**Exchange**(*name: str*, *service: Callable*, *options: tensortrade.oms.exchanges.exchange.ExchangeOptions = None*)

Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimedIdentifiable*

An abstract exchange for use within a trading environment.

Parameters

- **name** (*str*) – The name of the exchange.
- **service** (*Union[Callable, str]*) – The service to be used for filling orders.
- **options** (*ExchangeOptions*) – The options used to specify the setting of the exchange.

__call__(**streams*) → tensortrade.oms.exchanges.exchange.Exchange

Sets up the price streams used to generate the prices.

Parameters ***streams** – The positional arguments each being a price stream.

Returns *Exchange* – The exchange the price streams were passed in for.

execute_order(*order: Order*, *portfolio: Portfolio*) → None
Execute an order on the exchange.

Parameters

- **order** (*Order*) – The order to execute.
- **portfolio** (*Portfolio*) – The portfolio to use.

is_pair_tradable(*trading_pair: tensortrade.oms.instruments.trading_pair.TradingPair*) → bool
Whether or not the specified trading pair is tradable on this exchange.

Parameters **trading_pair** (*TradingPair*) – The trading pair to test the tradability of.

Returns *bool* – Whether or not the pair is tradable.

quote_price(*trading_pair: tensortrade.oms.instruments.trading_pair.TradingPair*) → decimal.Decimal
The quote price of a trading pair on the exchange, denoted in the core instrument.

Parameters **trading_pair** (*TradingPair*) – The trading pair to get the quote price for.

Returns *Decimal* – The quote price of the specified trading pair, denoted in the core instrument.

registered_name = 'exchanges'

streams() → List[Stream[float]]
Gets the price streams for the exchange.

Returns *List[Stream[float]]* – The price streams for the exchange.

class tensortrade.oms.exchanges.exchange.**ExchangeOptions**(*commission: float = 0.003*, *min_trade_size: float = 1e-06*, *max_trade_size: float = 1000000.0*, *min_trade_price: float = 1e-08*, *max_trade_price: float = 100000000.0*, *is_live: bool = False*)

Bases: object

An options class to specify the settings of an exchange.

Parameters

- **commission** (*float, default 0.003*) – The percentage of the order size taken by the exchange.
- **min_trade_size** (*float, default 1e-6*) – The minimum trade size an order can have.
- **max_trade_size** (*float, default 1e6*) – The maximum trade size an order can have.
- **min_trade_price** (*float, default 1e-8*) – The minimum price an exchange can have.
- **max_trade_price** (*float, default 1e8*) – The maximum price an exchange can have.
- **is_live** (*bool, default False*) – Whether live orders should be submitted to the exchange.

tensortrade.oms.instruments package

Submodules

tensortrade.oms.instruments.exchange_pair module

class tensortrade.oms.instruments.exchange_pair.**ExchangePair**(*exchange: Exchange*, *pair: TradingPair*)

Bases: object

A pair of financial instruments to be traded on a specific exchange.

Parameters

- **exchange** (*Exchange*) – An exchange that contains the *pair* for trading.
- **pair** (*TradingPair*) – A trading pair available on the *exchange*.

inverse_price
The inverse price of the trading pair. (‘Decimal, read-only)

price
The quoted price of the trading pair. (*Decimal*, read-only)

tensortrade.oms.instruments.instrument module

class tensortrade.oms.instruments.instrument.**Instrument**(*symbol: str*, *precision: int*, *name: str = None*)

Bases: object

A financial instrument for use in trading.

Parameters

- **symbol** (*str*) – The symbol used on an exchange for a particular instrument. (e.g. AAPL, BTC, TSLA)
- **precision** (*int*) – The precision the amount of the instrument is denoted with. (e.g. BTC=8, AAPL=1)
- **name** (*str, optional*) – The name of the instrument being created.

__eq__(*other: Any*) → bool
Checks if two instruments are equal.

Parameters **other** (*Any*) – The instrument being compared.

Returns *bool* – Whether the instruments are equal.

__ne__(*other: Any*) → bool
Checks if two instruments are not equal.

Parameters **other** (*Any*) – The instrument being compared.

Returns *bool* – Whether the instruments are not equal.

__rmul__(*other: float*) → tensortrade.oms.instruments.quantity.Quantity
Enables reverse multiplication.

Parameters **other** (*float*) – The number used to create a quantity.

Returns *Quantity* – The quantity created by the number and the instrument involved with this operation.

__truediv__(*other: tensortrade.oms.instruments.instrument.Instrument*) → tensortrade.oms.instruments.trading_pair.TradingPair

Creates a trading pair through division.

Parameters **other** (*Instrument*) – The instrument that will be the quote of the pair.

Returns *TradingPair* – The trading pair created from the two instruments.

Raises

- `InvalidTradingPair` – Raised if *other* is the same instrument as *self*.
- `Exception` – Raised if *other* is not an instrument.

tensortrade.oms.instruments.quantity module

class tensortrade.oms.instruments.quantity.**Quantity**(*instrument: Instrument*, *size: decimal.Decimal*, *path_id: str = None*)

Bases: `object`

A size of a financial instrument for use in trading.

Parameters

- **instrument** (*Instrument*) – The unit of the quantity.
- **size** (*Decimal*) – The number of units of the instrument.
- **path_id** (*str, optional*) – The path order_id that this quantity is allocated for and associated with.

Raises `InvalidNegativeQuantity` – Raised if the *size* of the quantity being created is negative.

as_float() → float

Gets the size as a *float*.

Returns *float* – The size as a floating point number.

contain(*exchange_pair: ExchangePair*)

Contains the size of the quantity to be compatible with the settings of a given exchange.

Parameters **exchange_pair** (*ExchangePair*) – The exchange pair containing the exchange the quantity must be compatible with.

Returns *Quantity* – A quantity compatible with the given exchange.

convert(*exchange_pair: ExchangePair*) → Quantity

Converts the quantity into the value of another instrument based on its exchange rate from an exchange.

Parameters **exchange_pair** (*ExchangePair*) – The exchange pair to use for getting the quoted price to perform the conversion.

Returns *Quantity* – The value of the current quantity in terms of the quote instrument.

free() → tensortrade.oms.instruments.quantity.Quantity

Gets the free version of this quantity.

Returns *Quantity* – The free version of the quantity.

is_locked
If quantity is locked for an order. (bool, read-only)

lock_for(*path_id: str*) → tensortrade.oms.instruments.quantity.Quantity
Locks a quantity for an *Order* identified associated with *path_id*.

Parameters **path_id** (*str*) – The identification of the order path.

Returns *Quantity* – A locked quantity for an order path.

quantize() → tensortrade.oms.instruments.quantity.Quantity
Computes the quantization of current quantity in terms of the instrument's precision.

Returns *Quantity* – The quantized quantity.

static validate(*left: Union[tensortrade.oms.instruments.quantity.Quantity, numbers.Number], right: Union[tensortrade.oms.instruments.quantity.Quantity, numbers.Number]*) → Tuple[tensortrade.oms.instruments.quantity.Quantity, tensortrade.oms.instruments.quantity.Quantity]
Validates the given left and right arguments of a numeric or boolean operation.

Parameters

- **left** (*Union[Quantity, Number]*) – The left argument of an operation.
- **right** (*Union[Quantity, Number]*) – The right argument of an operation.

Returns *Tuple[Quantity, Quantity]* – The validated quantity arguments to use in a numeric or boolean operation.

Raises

- IncompatibleInstrumentOperation – Raised if the instruments left and right quantities are not equal.
- **QuantityOpPathMismatch – Raised if**
  - One argument is locked and the other argument is not.
  - Both arguments are locked quantities with unequal path_ids.
- InvalidNonNumericQuantity – Raised if either argument is a non-numeric object.
- Exception – If the operation is not valid.

tensortrade.oms.instruments.trading_pair module

class tensortrade.oms.instruments.trading_pair.**TradingPair**(*base: Instrument, quote: Instrument*)
Bases: object

A pair of financial instruments to be traded on an exchange.

Parameters

- **base** (*Instrument*) – The base instrument of the trading pair.
- **quote** (*Instrument*) – The quote instrument of the trading pair.

Raises InvalidTradingPair – Raises if base and quote instrument are equal.

tensortrade.oms.orders package

Submodules

tensortrade.oms.orders.broker module

class tensortrade.oms.orders.broker.**Broker**

Bases: *tensortrade.oms.orders.order_listener.OrderListener*, *tensortrade.core.base.TimeIndexed*

A broker for handling the execution of orders on multiple exchanges. Orders are kept in a virtual order book until they are ready to be executed.

unexecuted

The list of orders the broker is waiting to execute, when their criteria is satisfied.

Type *List[Order]*

executed

The dictionary of orders the broker has executed since resetting, organized by order id.

Type *Dict[str, Order]*

trades

The dictionary of trades the broker has executed since resetting, organized by order id.

Type *Dict[str, Trade]*

cancel(*order: tensortrade.oms.orders.order.Order*) → None

Cancels an order.

Parameters **order** (*Order*) – The order to be canceled.

on_fill(*order: Order*, *trade: Trade*) → None

Updates the broker after an order has been filled.

Parameters

- **order** (*Order*) – The order that is being filled.
- **trade** (*Trade*) – The trade that is being made to fill the order.

reset() → None

Resets the broker.

submit(*order: tensortrade.oms.orders.order.Order*) → None

Submits an order to the broker.

Adds *order* to the queue of orders waiting to be executed.

Parameters **order** (*Order*) – The order to be submitted.

update() → None

Updates the brokers order management system.

The broker will look through the unexecuted orders and if an order is ready to be executed the broker will submit it to the executed list and execute the order.

Then the broker will find any orders that are active, but expired, and proceed to cancel them.

tensortrade.oms.orders.create module

tensortrade.oms.orders.create.**hidden_limit_order**(*side: tensortrade.oms.orders.trade.TradeSide*, *exchange_pair: tensortrade.oms.instruments.exchange_pair.ExchangePair*, *limit_price: float*, *size: float*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*, *start: int = None*, *end: int = None*)

Creates a hidden limit order.

Parameters

- **side** (*TradeSide*) – The side of the order.
- **exchange_pair** (*ExchangePair*) – The exchange pair to perform the order for.
- **limit_price** (*float*) – The limit price of the order.
- **size** (*float*) – The size of the order.
- **portfolio** (*Portfolio*) – The portfolio being used in the order.
- **start** (*int, optional*) – The start time of the order.
- **end** (*int, optional*) – The end time of the order.

Returns *Order* – A hidden limit order.

tensortrade.oms.orders.create.**limit_order**(*side: tensortrade.oms.orders.trade.TradeSide*, *exchange_pair: tensortrade.oms.instruments.exchange_pair.ExchangePair*, *limit_price: float*, *size: float*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*, *start: int = None*, *end: int = None*)

Creates a limit order.

Parameters

- **side** (*TradeSide*) – The side of the order.
- **exchange_pair** (*ExchangePair*) – The exchange pair to perform the order for.
- **limit_price** (*float*) – The limit price of the order.
- **size** (*float*) – The size of the order.
- **portfolio** (*Portfolio*) – The portfolio being used in the order.
- **start** (*int, optional*) – The start time of the order.
- **end** (*int, optional*) – The end time of the order.

Returns *Order* – A limit order.

tensortrade.oms.orders.create.**market_order**(*side: tensortrade.oms.orders.trade.TradeSide*, *exchange_pair: tensortrade.oms.instruments.exchange_pair.ExchangePair*, *price: float*, *size: float*, *portfolio: tensortrade.oms.wallets.portfolio.Portfolio*) → tensortrade.oms.orders.order.Order

Creates a market order.

Parameters

- **side** (*TradeSide*) – The side of the order.
- **exchange_pair** (*ExchangePair*) – The exchange pair to perform the order for.
- **price** (*float*) – The current price.
- **size** (*float*) – The size of the order.
- **portfolio** (*Portfolio*) – The portfolio being used in the order.

Returns *Order* – A market order.

tensortrade.oms.orders.create.**proportion_order**(*portfolio: Portfolio*, *source: Wallet*, *target: Wallet*, *proportion: float*) → Order

Creates an order that sends a proportion of funds from one wallet to another.

Parameters

- **portfolio** (*Portfolio*) – The portfolio that contains both wallets.
- **source** (*Wallet*) – The source wallet for the funds.
- **target** (*Wallet*) – The target wallet for the funds.
- **proportion** (*float*) – The proportion of funds to send.

tensortrade.oms.orders.create.**risk_managed_order**(*side: TradeSide*, *trade_type: TradeType*, *exchange_pair: ExchangePair*, *price: float*, *quantity: Quantity*, *down_percent: float*, *up_percent: float*, *portfolio: Portfolio*, *start: int = None*, *end: int = None*)

Create a stop order that manages for percentages above and below the entry price of the order.

Parameters

- **side** (*TradeSide*) – The side of the order.
- **trade_type** (*TradeType*) – The type of trade to make when going in.
- **exchange_pair** (*ExchangePair*) – The exchange pair to perform the order for.
- **price** (*float*) – The current price.
- **down_percent** (*float*) – The percentage the price is allowed to drop before exiting.
- **up_percent** (*float*) – The percentage the price is allowed to rise before exiting.
- **quantity** (*Quantity*) – The quantity of the order.
- **portfolio** (*Portfolio*) – The portfolio being used in the order.
- **start** (*int, optional*) – The start time of the order.
- **end** (*int, optional*) – The end time of the order.

Returns *Order* – A stop order controlling for the percentages above and below the entry price.

tensortrade.oms.orders.criteria module

class tensortrade.oms.orders.criteria.**Criteria**

Bases: object

A criteria to be satisfied before an order will be executed.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*) → bool
Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

class tensortrade.oms.orders.criteria.**CriteriaBinOp**(*left: Callable[[tensortrade.oms.orders.order.Order, tensortrade.oms.exchanges.exchange.Exchange], bool], right: Callable[[tensortrade.oms.orders.order.Order, tensortrade.oms.exchanges.exchange.Exchange], bool], op: Callable[[bool, bool], bool], op_str: str*)
Bases: *tensortrade.oms.orders.criteria.Criteria*

A class for using a binary operation for criteria.

Parameters

- **left** (*Callable[[Order, Exchange], bool]*) – The left criteria argument.
- **right** (*Callable[[Order, Exchange], bool]*) – The right criteria argument.
- **op** (*Callable[[bool, bool], bool]*) – The binary boolean operation.
- **op_str** (*str*) – The string representing the op.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*) → bool
Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

class tensortrade.oms.orders.criteria.**Limit**(*limit_price: float*)
Bases: *tensortrade.oms.orders.criteria.Criteria*

An order criteria that allows execution when the quote price for a trading pair is at or below a specific price, hidden from the public order book.

Parameters **limit_price** (*float*) – The quote price to check for execution.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*) → bool
Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

class tensortrade.oms.orders.criteria.**NotCriteria**(*criteria: Callable[[tensortrade.oms.orders.order.Order, tensortrade.oms.exchanges.exchange.Exchange], bool]*)

Bases: *tensortrade.oms.orders.criteria.Criteria*

A criteria to invert the truth value of another criteria.

Parameters **criteria** (*Callable[[Order, Exchange], bool]*) – The criteria to invert the truth value of.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*) → bool

Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

class tensortrade.oms.orders.criteria.**Stop**(*direction: Union[tensortrade.oms.orders.criteria.StopDirection, str], percent: float*)

Bases: *tensortrade.oms.orders.criteria.Criteria*

An order criteria that allows execution when the quote price for a trading pair is above or below a specific price.

Parameters

- **direction** (*Union[StopDirection, str]*) – The direction to watch for the stop criteria.
- **percent** (*float*) – The percentage of the current price to use for watching.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*) → bool

Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

class tensortrade.oms.orders.criteria.**StopDirection**

Bases: enum.Enum

An enumeration for the directions of a stop criteria.

DOWN = 'down'

UP = 'up'

class tensortrade.oms.orders.criteria.**Timed**(*duration: float*)

Bases: *tensortrade.oms.orders.criteria.Criteria*

An order criteria for waiting a certain amount of time for execution.

Parameters **duration** (*float*) – The amount of time to wait.

check(*order: tensortrade.oms.orders.order.Order*, *exchange: tensortrade.oms.exchanges.exchange.Exchange*)

Checks whether the *order* is executable on *exchange*.

Parameters

- **order** (*Order*) – An order.
- **exchange** (*Exchange*) – The exchange to check.

Returns *bool* – Whether *order* is executable on *exchange*.

tensortrade.oms.orders.order module

class tensortrade.oms.orders.order.**Order**(*step: int, side: tensortrade.oms.orders.trade.TradeSide, trade_type: tensortrade.oms.orders.trade.TradeType, exchange_pair: ExchangePair, quantity: Quantity, portfolio: Portfolio, price: float, criteria: Callable[[Order, Exchange], bool] = None, path_id: str = None, start: int = None, end: int = None*)

Bases: *tensortrade.core.base.TimedIdentifiable*, *tensortrade.core.base.Observable*

A class to represent ordering an amount of a financial instrument.

Responsibilities of the Order:

1. Confirming its own validity.
2. Tracking its trades and reporting it back to the broker.
3. Managing movement of quantities from order to order.
4. Generating the next order in its path given that there is a 'OrderSpec' for how to make the next order.
5. Managing its own state changes when it can.

Parameters **side** (*TradeSide*) – The side of the order.

exchange_pair [*ExchangePair*] The exchange pair to perform the order for.

price [float] The price of the order.

trade_type [*TradeType*] The type of trade being made.

exchange_pair [*ExchangePair*] The exchange pair that the order is made for.

quantity [*Quantity*] The quantity of the order.

portfolio [*Portfolio*] The portfolio being used in the order.

criteria [*Callable[[Order, Exchange], bool]*, optional] The criteria under which the order will be considered executable.

path_id [str, optional] The path order id.

start [int, optional] The start time of the order.

end [int, optional] The end time of the order.

Raises InvalidOrderQuantity – Raised if the given quantity has a size of 0.

add_order_spec(*order_spec: OrderSpec*) → Order
Adds an order specification to the order.

Parameters **order_spec** (*OrderSpec*) – An order specification.

Returns *Order* – The current order.

base_instrument
The base instrument of the pair being traded.

cancel (*reason: str = 'CANCELLED'*) → None
Cancels an order.

Parameters **reason** (*str, default 'CANCELLED'*) – The reason for canceling the order.

complete () → tensortrade.oms.orders.order.Order
Completes an order.

Returns *Order* – The completed order.

execute () → None
Executes the order.

fill (*trade: tensortrade.oms.orders.trade.Trade*) → None
Fills the order.

Parameters **trade** (*Trade*) – A trade to fill the order.

is_active
If this order is active. (bool, read-only)

is_buy
If this is a buy order. (bool, read-only)

is_cancelled
If this order is cancelled. (bool, read-only)

is_complete
If this order is complete. (bool, read-only)

is_executable
If this order is executable. (bool, read-only)

is_expired
If this order is expired. (bool, read-only)

is_limit_order
If this is a limit order. (bool, read-only)

is_market_order
If this is a market order. (bool, read-only)

is_sell
If this is a sell order. (bool, read-only)

pair
The trading pair of the order. (*TradingPair*, read-only)

quote_instrument
The quote instrument of the pair being traded.

release (*reason: str = 'RELEASE (NO REASON)'*) → None
Releases all quantities from every wallet that have been allocated for this order.

Parameters **reason** (*str, default 'RELEASE (NO REASON)'*) – The reason for releasing all locked quantities associated with the order.

size
The size of the order. (*Decimal*, read-only)

to_dict() → dict
Creates a dictionary representation of the order.

Returns *dict* – The dictionary representation of the order.

to_json() → dict
Creates a json dictionary representation of the order.

Returns *dict* – The json dictionary representation of the order

class tensortrade.oms.orders.order.**OrderStatus**
Bases: enum.Enum

An enumeration for the status of an order.

CANCELLED = 'cancelled'

FILLED = 'filled'

OPEN = 'open'

PARTIALLY_FILLED = 'partially_filled'

PENDING = 'pending'

tensortrade.oms.orders.order_listener module

class tensortrade.oms.orders.order_listener.**OrderListener**
Bases: object

A callback class for an order.

on_cancel(*order: Order*) → None
Callback for an order after cancellation.

Parameters **order** (*Order*) – The cancelled order.

on_complete(*order: Order*) → None
Callback for an order after being completed.

Parameters **order** (*Order*) – The completed order.

on_execute(*order: Order*) → None
Callback for an order after execution.

Parameters **order** (*Order*) – The executed order.

on_fill(*order: Order*, *trade: Trade*) → None
Callback for an order after being filled.

Parameters

- **order** (*Order*) – The order being filled.
- **trade** (*Trade*) – The trade that is filling the order.

tensortrade.oms.orders.order_spec module

class tensortrade.oms.orders.order_spec.**OrderSpec**(*side: TradeSide, trade_type: TradeType, exchange_pair: ExchangePair, criteria: Callable[[Order, Exchange], bool] = None*)

Bases: *tensortrade.core.base.Identifiable*

A class for order creation following an order being complete.

Parameters

- **side** (*TradeSide*) – The trading side of the specification.
- **trade_type** (*TradeType*) – The type of trade for the specification.
- **exchange_pair** (*ExchangePair*) – The exchange pair for the specification.
- **criteria** (*Callable[[Order, Exchange], bool]*) – The criteria for executing the order after its been created.

create_order(*order: tensortrade.oms.orders.order.Order*) → tensortrade.oms.orders.order.Order
Creates an order following from another order.

Parameters **order** (*Order*) – The previous order in the order path.

Returns *Order* – The order created from the specification parameters and the parameters of *order*.

to_dict() → dict
Creates dictionary representation of specification.

Returns *dict* – The dictionary representation of specification.

tensortrade.oms.orders.trade module

class tensortrade.oms.orders.trade.**Trade**(*order_id: str, step: int, exchange_pair: ExchangePair, side: tensortrade.oms.orders.trade.TradeSide, trade_type: tensortrade.oms.orders.trade.TradeType, quantity: Quantity, price: float, commission: Quantity*)

Bases: *tensortrade.core.base.TimedIdentifiable*

A trade object for use within trading environments.

__init__(*order_id: str, step: int, exchange_pair: ExchangePair, side: tensortrade.oms.orders.trade.TradeSide, trade_type: tensortrade.oms.orders.trade.TradeType, quantity: Quantity, price: float, commission: Quantity*)

Parameters

- **order_id** – The id of the order that created the trade.
- **step** – The timestep the trade was made during the trading episode.
- **exchange_pair** – The exchange pair of instruments in the trade.
- **BTC/USDT, ETH/BTC, ADA/BTC, AAPL/USD, NQ1!/USD, CAD/USD, etc)** (*e.g.*) –
- **side** – Whether the quote instrument is being bought or sold.

- **BUY = trade the base_instrument for the quote_instrument in the pair. SELL = trade the quote_instrument for the base_instrument)** (*e.g.*) –
- **size** – The size of the core instrument in the trade.
- **1000 shares, 6.50 satoshis, 2.3 contracts, etc)** (*e.g.*) –
- **price** – The price paid per quote instrument in terms of the core instrument.
- **10000 represents $10,000.00 if the base_instrument is "USD")** (*e.g.*) –
- **commission** – The commission paid for the trade in terms of the core instrument.
- **10000 represents $10,000.00 if the base_instrument is "USD")** –

base_instrument

commission

is_buy

is_limit_order

is_market_order

is_sell

price

quote_instrument

size

to_dict()

to_json()

class tensortrade.oms.orders.trade.**TradeSide**

Bases: enum.Enum

An enumeration.

BUY = 'buy'

SELL = 'sell'

instrument(*pair: TradingPair*) → Instrument

class tensortrade.oms.orders.trade.**TradeType**

Bases: enum.Enum

An enumeration.

LIMIT = 'limit'

MARKET = 'market'

tensortrade.oms.services package

Subpackages

tensortrade.oms.services.execution package

Submodules

tensortrade.oms.services.execution.ccxt module

tensortrade.oms.services.execution.interactive_brokers module

tensortrade.oms.services.execution.robinhood module

tensortrade.oms.services.execution.simulated module

tensortrade.oms.services.execution.simulated.**execute_buy_order**(*order: tensortrade.oms.orders.order.Order*, *base_wallet: tensortrade.oms.wallets.wallet.Wallet*, *quote_wallet: tensortrade.oms.wallets.wallet.Wallet*, *current_price: float*, *options: tensortrade.oms.exchanges.exchange.Exchange*, *clock: tensortrade.core.clock.Clock*) → Union[None, tensortrade.oms.orders.trade.Trade]

Executes a buy order on the exchange.

Parameters

- **order** (*Order*) – The order that is being filled.
- **base_wallet** (*Wallet*) – The wallet of the base instrument.
- **quote_wallet** (*Wallet*) – The wallet of the quote instrument.
- **current_price** (*float*) – The current price of the exchange pair.
- **options** (*ExchangeOptions*) – The exchange options.
- **clock** (*Clock*) – The clock for the trading process..

Returns *Trade* – The executed trade that was made.

tensortrade.oms.services.execution.simulated.**execute_order**(*order: Order*, *base_wallet: Wallet*, *quote_wallet: Wallet*, *current_price: float*, *options: Options*, *clock: Clock*) → Trade

Executes an order on the exchange.

Parameters

- **order** (*Order*) – The order that is being filled.
- **base_wallet** (*Wallet*) – The wallet of the base instrument.
- **quote_wallet** (*Wallet*) – The wallet of the quote instrument.
- **current_price** (*float*) – The current price of the exchange pair.
- **options** (*ExchangeOptions*) – The exchange options.
- **clock** (*Clock*) – The clock for the trading process..

Returns *Trade* – The executed trade that was made.

tensortrade.oms.services.execution.simulated.**execute_sell_order**(*order: tensortrade.oms.orders.order.Order*, *base_wallet: tensortrade.oms.wallets.wallet.Wallet*, *quote_wallet: tensortrade.oms.wallets.wallet.Wallet*, *current_price: float*, *options: tensortrade.oms.exchanges.exchange.Exchang*
clock: tensortrade.core.clock.Clock) → Union[None, tensortrade.oms.orders.trade.Trade]

Executes a sell order on the exchange.

Parameters

- **order** (*Order*) – The order that is being filled.
- **base_wallet** (*Wallet*) – The wallet of the base instrument.
- **quote_wallet** (*Wallet*) – The wallet of the quote instrument.
- **current_price** (*float*) – The current price of the exchange pair.
- **options** (*ExchangeOptions*) – The exchange options.
- **clock** (*Clock*) – The clock for the trading process..

Returns *Trade* – The executed trade that was made.

tensortrade.oms.services.slippage package

tensortrade.oms.services.slippage.**get**(*identifier: str*) → tensortrade.oms.services.slippage.slippage_model.SlippageModel

Gets the *SlippageModel* that matches with the identifier.

Parameters **identifier** – The identifier for the *SlippageModel*

Raises KeyError – if identifier is not associated with any *SlippageModel*

Submodules

tensortrade.oms.services.slippage.random_slippage_model module

class tensortrade.oms.services.slippage.random_slippage_model.**RandomUniformSlippageModel**(*ma*
flo
=
3.

Bases: *tensortrade.oms.services.slippage.slippage_model.SlippageModel*

A uniform random slippage model.

Parameters **max_slippage_percent** (*float, default 3.0*) – The maximum random slippage to be applied to the fill price.

adjust_trade(*trade: tensortrade.oms.orders.trade.Trade*, ***kwargs*) → tensortrade.oms.orders.trade.Trade
Simulate slippage on a trade ordered on a specific exchange.

Parameters

- **trade** (*Trade*) – The trade executed on the exchange.
- ****kwargs** (*keyword arguments*) – Any other arguments necessary for the model.

Returns *Trade* – A filled trade with the *price* and *size* adjusted for slippage.

tensortrade.oms.services.slippage.slippage_model module

class tensortrade.oms.services.slippage.slippage_model.**SlippageModel**
Bases: *tensortrade.core.component.Component*

A model for simulating slippage on an exchange trade.

adjust_trade(*trade: tensortrade.oms.orders.trade.Trade*, ***kwargs*) → tensortrade.oms.orders.trade.Trade
Simulate slippage on a trade ordered on a specific exchange.

Parameters

- **trade** (*Trade*) – The trade executed on the exchange.
- ****kwargs** (*keyword arguments*) – Any other arguments necessary for the model.

Returns *Trade* – A filled trade with the *price* and *size* adjusted for slippage.

registered_name = 'slippage'

tensortrade.oms.wallets package

tensortrade.oms.wallets.**get**(*identifier: str*) → tensortrade.oms.wallets.portfolio.Portfolio
Gets the *TradingStrategy* that matches with the identifier.

Parameters **identifier** – The identifier for the *TradingStrategy*

Raises KeyError – if identifier is not associated with any *TradingStrategy*

Submodules

tensortrade.oms.wallets.ledger module

class tensortrade.oms.wallets.ledger.**Ledger**
Bases: object

A ledger to keep track of transactions that occur in the order management system.

as_frame(*sort_by_order_seq: bool = False*) → pandas.core.frame.DataFrame
Converts the ledger records into a data frame.

Parameters **sort_by_order_seq** (*bool, default False*) – If records should be sorted by each order path.

Returns *pd.DataFrame* – A data frame containing all the records in the ledger.

commit(*wallet: Wallet*, *quantity: Quantity*, *source: str*, *target: str*, *memo: str*) → None
Commits a transaction to the ledger records.

Parameters

- **wallet** (*Wallet*) – The wallet involved in this transaction.
- **quantity** (*Quantity*) – The amount being used.
- **source** (*str*) – The source of funds being transferred.
- **target** (*str*) – The destination the funds are being transferred to.
- **memo** (*str*) – A description of the transaction.

reset()
Resets the ledger.

class tensortrade.oms.wallets.ledger.**Transaction**(*poid*, *step*, *source*, *target*, *memo*, *amount*, *free*, *locked*, *locked_poid*)
Bases: tuple

amount
Alias for field number 5

free
Alias for field number 6

locked
Alias for field number 7

locked_poid
Alias for field number 8

memo
Alias for field number 4

poid
Alias for field number 0

source
Alias for field number 2

step
Alias for field number 1

target
Alias for field number 3

tensortrade.oms.wallets.portfolio module

class tensortrade.oms.wallets.portfolio.**Portfolio**(*base_instrument: tensortrade.oms.instruments.instrument.Instrument, wallets: List[WalletType] = None, order_listener: Optional[tensortrade.oms.orders.order_listener.OrderListener] = None, performance_listener: Callable[[collections.OrderedDict], None] = None*)

Bases: *tensortrade.core.component.Component*, *tensortrade.core.base.TimedIdentifiable*

A portfolio of wallets on exchanges.

Parameters

- **base_instrument** (*Instrument*) – The exchange instrument used to measure value and performance statistics.
- **wallets** (*List[WalletType]*) – The wallets to be used in the portfolio.
- **order_listener** (*OrderListener*) – The order listener to set for all orders executed by this portfolio.
- **performance_listener** (*Callable[[OrderedDict], None]*) – The performance listener to send all portfolio updates to.

add(*wallet: WalletType*) → None
Adds a wallet to the portfolio.

Parameters **wallet** (*WalletType*) – The wallet to add to the portfolio.

balance(*instrument: tensortrade.oms.instruments.instrument.Instrument*) → tensortrade.oms.instruments.quantity.Quantity
Gets the total balance of the portfolio in a specific instrument available for use.

Parameters **instrument** (*Instrument*) – The instrument to compute the balance for.

Returns *Quantity* – The balance of the instrument over all wallets.

balances
The current unlocked balance of each instrument over all wallets. (*List[Quantity]*, read-only)

base_balance
The current balance of the base instrument over all wallets. (*Quantity*, read-only)

exchange_pairs
All the exchange pairs in the portfolio. (*List[ExchangePair]*, read-only)

exchanges
All the exchanges in the portfolio. (*List[Exchange]*, read-only)

get_wallet(*exchange_id: str, instrument: tensortrade.oms.instruments.instrument.Instrument*) → tensortrade.oms.wallets.wallet.Wallet
Gets wallet by the *exchange_id* and *instrument*.

Parameters

- **exchange_id** (*str*) – The exchange id used to identify the wallet.

- **instrument** (*Instrument*) – The instrument used to identify the wallet.

Returns *Wallet* – The wallet associated with *exchange_id* and *instrument*.

initial_balance
The initial balance of the base instrument over all wallets. (*Quantity*, read-only)

initial_net_worth
The initial net worth of the portfolio. (float, read-only)

ledger
The ledger that keeps track of transactions. (*Ledger*, read-only)

locked_balance(*instrument: tensortrade.oms.instruments.instrument.Instrument*) → tensortrade.oms.instruments.quantity.Quantity
Gets the total balance a specific instrument locked in orders over the entire portfolio.

Parameters **instrument** (*Instrument*) – The instrument to find locked balances for.

Returns *Quantity* – The total locked balance of the instrument.

locked_balances
The current locked balance of each instrument over all wallets. (*List[Quantity]*, read-only)

net_worth
The current net worth of the portfolio. (float, read-only)

on_next(*data: dict*) → None
Updates the performance metrics.

Parameters **data** (*dict*) – The data produced from the observer feed that is used to update the performance metrics.

performance
The performance of the portfolio since the last reset. (*OrderedDict*, read-only)

profit_loss
The percent loss in net worth since the last reset. (float, read-only)

registered_name = 'portfolio'

remove(*wallet: tensortrade.oms.wallets.wallet.Wallet*) → None
Removes a wallet from the portfolio.

Parameters **wallet** (*Wallet*) – The wallet to be removed.

remove_pair(*exchange: tensortrade.oms.exchanges.exchange.Exchange*, *instrument: tensortrade.oms.instruments.instrument.Instrument*) → None
Removes a wallet from the portfolio by *exchange* and *instrument*.

Parameters

- **exchange** (*Exchange*) – The exchange of the wallet to be removed.
- **instrument** (*Instrument*) – The instrument of the wallet to be removed.

reset() → None
Resets the portfolio.

total_balance(*instrument: tensortrade.oms.instruments.instrument.Instrument*) → tensortrade.oms.instruments.quantity.Quantity
Gets the total balance of a specific instrument over the portfolio, both available for use and locked in orders.

Parameters **instrument** (*Instrument*) – The instrument to get total balance of.

Returns *Quantity* – The total balance of *instrument* over the portfolio.

total_balances
The current total balance of each instrument over all wallets. (*List[Quantity]*, read-only)

wallets
All the wallets in the portfolio. (*List[Wallet]*, read-only)

tensortrade.oms.wallets.wallet module

class tensortrade.oms.wallets.wallet.**Transfer**(*quantity*, *commission*, *price*)
Bases: tuple

commission
Alias for field number 1

price
Alias for field number 2

quantity
Alias for field number 0

class tensortrade.oms.wallets.wallet.**Wallet**(*exchange: tensortrade.oms.exchanges.exchange.Exchange*, *balance: tensortrade.oms.instruments.quantity.Quantity*)
Bases: *tensortrade.core.base.Identifiable*

A wallet stores the balance of a specific instrument on a specific exchange.

Parameters

- **exchange** (*Exchange*) – The exchange associated with this wallet.
- **balance** (*Quantity*) – The initial balance quantity for the wallet.

deposit(*quantity: tensortrade.oms.instruments.quantity.Quantity*, *reason: str*) → tensortrade.oms.instruments.quantity.Quantity
Deposits funds into the wallet.

Parameters

- **quantity** (*Quantity*) – The amount to deposit into this wallet.
- **reason** (*str*) – The reason for depositing the amount.

Returns *Quantity* – The deposited amount.

classmethod from_tuple(*wallet_tuple: Tuple[tensortrade.oms.exchanges.exchange.Exchange, tensortrade.oms.instruments.instrument.Instrument, float]*) → tensortrade.oms.wallets.wallet.Wallet
Creates a wallet from a wallet tuple.

Parameters **wallet_tuple** (*Tuple[Exchange, Instrument, float]*) – A tuple containing an exchange, instrument, and amount.

Returns *Wallet* – A wallet corresponding to the arguments given in the tuple.

ledger = <tensortrade.oms.wallets.ledger.Ledger object>

lock(*quantity*, *order: tensortrade.oms.orders.order.Order*, *reason: str*) → tensortrade.oms.instruments.quantity.Quantity
Locks funds for specified order.

Parameters

- **quantity** (*Quantity*) – The amount of funds to lock for the order.
- **order** (*Order*) – The order funds will be locked for.
- **reason** (*str*) – The reason for locking funds.

Returns *Quantity* – The locked quantity for *order*.

Raises

- `DoubleLockedQuantity` – Raised if the given amount is already a locked quantity.
- `InsufficientFunds` – Raised if amount is greater the current balance.

locked
The current quantities that are locked for orders. (*Dict[str, Quantity]*, read-only)

locked_balance
The total balance of the wallet locked in orders. (*Quantity*, read-only)

reset() → None
Resets the wallet.

total_balance
The total balance of the wallet available for use and locked in orders. (*Quantity*, read-only)

static transfer(*source: tensortrade.oms.wallets.wallet.Wallet*, *target: tensortrade.oms.wallets.wallet.Wallet*, *quantity: tensortrade.oms.instruments.quantity.Quantity*, *commission: tensortrade.oms.instruments.quantity.Quantity*, *exchange_pair: tensortrade.oms.instruments.exchange_pair.ExchangePair*, *reason: str*) → tensortrade.oms.wallets.wallet.Transfer
Transfers funds from one wallet to another.

Parameters

- **source** (*Wallet*) – The wallet in which funds will be transferred from
- **target** (*Wallet*) – The wallet in which funds will be transferred to
- **quantity** (*Quantity*) – The quantity to be transferred from the source to the target. In terms of the instrument of the source wallet.
- **commission** (*Quantity*) – The commission to be taken from the source wallet for performing the transfer of funds.
- **exchange_pair** (*ExchangePair*) – The exchange pair associated with the transfer
- **reason** (*str*) – The reason for transferring the funds.

Returns *Transfer* – A transfer object describing the transaction.

Raises `Exception` – Raised if an equation that describes the conservation of funds is broken.

unlock(*quantity: tensortrade.oms.instruments.quantity.Quantity*, *reason: str*) → tensortrade.oms.instruments.quantity.Quantity
Unlocks a certain amount from the locked funds of the wallet that are associated with the given *quantity* path id.

Parameters

- **quantity** (*Quantity*) – The quantity to unlock from the funds.

- **reason** (*str*) – The reason for unlocking funds.

Returns *Quantity* – The free quantity.

Raises

- `DoubleUnlockedFunds` – Raised if *quantity* is not a locked quantity.
- `QuantityNotLocked` – Raised if *quantity* has a path id that is not currently allocated in this wallet.
- `InsufficientFunds` – Raised if *quantity* is greater than the amount currently allocated for the associated path id.

withdraw(*quantity: tensortrade.oms.instruments.quantity.Quantity*, *reason: str*) → tensortrade.oms.instruments.quantity.Quantity

Withdraws funds from the wallet.

Parameters

- **quantity** (*Quantity*) – The amount to withdraw from this wallet.
- **reason** (*str*) – The reason for withdrawing the amount.

Returns *Quantity* – The withdrawn amount.

tensortrade.stochastic package

Subpackages

tensortrade.stochastic.processes package

Submodules

tensortrade.stochastic.processes.brownian_motion module

tensortrade.stochastic.processes.brownian_motion.**brownian_motion_levels**(*params: tensortrade.stochastic.utils.param* → numpy.array

Constructs a price sequence whose returns evolve according to brownian motion.

Parameters **params** (*ModelParameters*) – The parameters for the stochastic model.

Returns *np.array* – A price sequence which follows brownian motion.

tensortrade.stochastic.processes.brownian_motion.**brownian_motion_log_returns**(*params: tensortrade.stochastic.util* → numpy.array

Constructs a Wiener process (Brownian Motion).

Parameters **params** (*ModelParameters*) – The parameters for the stochastic model.

Returns *np.array* – Brownian motion log returns.

References

[1] http://en.wikipedia.org/wiki/Wiener_process

tensortrade.stochastic.processes.cox module

tensortrade.stochastic.processes.cox.**cox**(*base_price: int = 1*, *base_volume: int = 1*, *start_date: str = '2010-01-01'*, *start_date_format: str = '%Y-%m-%d'*, *times_to_generate: int = 1000*, *time_frame: str = '1h'*, *params: Optional[tensortrade.stochastic.utils.parameters.ModelParameters] = None*) → pandas.core.frame.DataFrame

Generates price data from the CIR process.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data
- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.
- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

References

[1] https://en.wikipedia.org/wiki/Cox%E2%80%93Ingersoll%E2%80%93Ross_model

tensortrade.stochastic.processes.cox.**cox_ingersoll_ross_levels**(*params: tensortrade.stochastic.utils.parameters.ModelP*
→ numpy.array

Constructs the rate levels of a mean-reverting Cox-Ingersoll-Ross process.

Used to model interest rates as well as stochastic volatility in the Heston model. We pass a correlated Brownian motion process into the method from which the interest rate levels are constructed because the returns between the underlying and the stochastic volatility should be correlated. The other correlated process is used in the Heston model.

Parameters **params** (*ModelParameters*) – The parameters for the stochastic model.

Returns *np.array* – The interest rate levels for the CIR process.

tensortrade.stochastic.processes.fbm module

tensortrade.stochastic.processes.fbm.**fbm**(*base_price: int = 1*, *base_volume: int = 1*, *start_date: str = '2010-01-01'*, *start_date_format: str = '%Y-%m-%d'*, *times_to_generate: int = 1000*, *hurst: float = 0.61*, *time_frame: str = '1h'*) → pandas.core.frame.DataFrame

Generates price data from the FBM process.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data
- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **hurst** (*float, default 0.61*) – The hurst parameter for the FBM process.
- **time_frame** (*str, default '1h'*) – The time frame.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

References

[1] https://en.wikipedia.org/wiki/Fractional_Brownian_motion

tensortrade.stochastic.processes.gbm module

tensortrade.stochastic.processes.gbm.**gbm**(*base_price: int = 1*, *base_volume: int = 1*, *start_date: str = '2010-01-01'*, *start_date_format: str = '%Y-%m-%d'*, *times_to_generate: int = 1000*, *time_frame: str = '1h'*, *params: Optional[tensortrade.stochastic.utils.parameters.ModelParameters] = None*) → pandas.core.frame.DataFrame

Generates price data from a GBM process.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data
- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.

- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

References

[1] https://en.wikipedia.org/wiki/Geometric_Brownian_motion

tensortrade.stochastic.processes.gbm.**geometric_brownian_motion_levels**(*params: tensortrade.stochastic.utils.paramete*

Constructs a sequence of price levels for an asset which evolves according to a geometric brownian motion process.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The price levels for the asset

tensortrade.stochastic.processes.gbm.**geometric_brownian_motion_log_returns**(*params: tensortrade.stochastic.utils.pa* → numpy.array

Constructs a sequence of log returns.

When log returns are exponentiated, it produces a random Geometric Brownian Motion (GBM). The GBM is the stochastic process underlying the Black-Scholes options pricing formula.

Parameters **params** (*ModelParameters*) – The parameters for the stochastic model.

Returns *np.array* – The log returns of a geometric brownian motion process

tensortrade.stochastic.processes.heston module

tensortrade.stochastic.processes.heston.**cox_ingersoll_ross_heston**(*params: tensortrade.stochastic.utils.parameters.Mo* → numpy.array

Constructs the rate levels of a mean-reverting cox ingersoll ross process.

Used to model interest rates as well as stochastic volatility in the Heston model. The returns between the underlying and the stochastic volatility should be correlated we pass a correlated Brownian motion process into the method from which the interest rate levels are constructed. The other correlated process are used in the Heston model.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The interest rate levels for the CIR process

tensortrade.stochastic.processes.heston.**geometric_brownian_motion_jump_diffusion_levels**(*par* → nu

Converts a sequence of gbm jmp returns into a price sequence which evolves according to a geometric brownian

motion but can contain jumps at any point in time.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The price levels.

tensortrade.stochastic.processes.heston.**geometric_brownian_motion_jump_diffusion_log_returr**

Constructs combines a geometric brownian motion process (log returns) with a jump diffusion process (log returns) to produce a sequence of gbm jump returns.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – A GBM process with jumps in it

tensortrade.stochastic.processes.heston.**get_correlated_geometric_brownian_motions**(*params: tensortrade.stocha correlation_matrix: numpy.array n: int*) → numpy.array

Constructs a basket of correlated asset paths using the Cholesky decomposition method.

Parameters

- **params** (*ModelParameters*) – The parameters for the stochastic model.
- **correlation_matrix** (*np.array*) – An n x n correlation matrix.
- **n** (*int*) – Number of assets (number of paths to return)

Returns *np.array* – n correlated log return geometric brownian motion processes.

tensortrade.stochastic.processes.heston.**heston**(*base_price: int = 1, base_volume: int = 1, start_date: str = '2010-01-01', start_date_format: str = '%Y-%m-%d', times_to_generate: int = 1000, time_frame: str = '1h', params: Optional[tensortrade.stochastic.utils.parameters.ModelParameters] = None*) → pandas.core.frame.DataFrame

Generates price data from the Heston model.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data

- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.
- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

tensortrade.stochastic.processes.heston.**heston_construct_correlated_path**(*params: tensortrade.stochastic.utils.parameters.ModelParameters*, *brownian_motion_one: numpy.array*) → numpy.array

A simplified version of the Cholesky decomposition method for just two assets. It does not make use of matrix algebra and is therefore quite easy to implement.

Parameters

- **params** (ModelParameters) – The parameters for the stochastic model.
- **brownian_motion_one** (*np.array*) – (Not filled)

Returns *np.array* – A correlated brownian motion path.

tensortrade.stochastic.processes.heston.**heston_model_levels**(*params: tensortrade.stochastic.utils.parameters.ModelParameters*) → numpy.array

Generates price levels corresponding to the Heston model.

The Heston model is the geometric brownian motion model with stochastic volatility. This stochastic volatility is given by the cox ingersoll ross process. Step one on this method is to construct two correlated GBM processes. One is used for the underlying asset prices and the other is used for the stochastic volatility levels.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The prices for an underlying following a Heston process

Warning: This method is dodgy! Need to debug!

tensortrade.stochastic.processes.heston.**jump_diffusion_process**(*params: tensortrade.stochastic.utils.parameters.ModelParameters*) → numpy.array

Produces a sequence of Jump Sizes which represent a jump diffusion process.

These jumps are combined with a geometric brownian motion (log returns) to produce the Merton model.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The jump sizes for each point in time (mostly zeroes if jumps are infrequent).

tensortrade.stochastic.processes.merton module

tensortrade.stochastic.processes.merton.**merton**(*base_price: int = 1*, *base_volume: int = 1*, *start_date: str = '2010-01-01'*, *start_date_format: str = '%Y-%m-%d'*, *times_to_generate: int = 1000*, *time_frame: str = '1h'*, *params: Optional[tensortrade.stochastic.utils.parameters.ModelParameters] = None*) → pandas.core.frame.DataFrame

Generates price data from the Merton Jump Diffusion model.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data
- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.
- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

tensortrade.stochastic.processes.ornstein_uhlenbeck module

tensortrade.stochastic.processes.ornstein_uhlenbeck.**ornstein**(*base_price: int = 1*, *base_volume: int = 1*, *start_date: str = '2010-01-01'*, *start_date_format: str = '%Y-%m-%d'*, *times_to_generate: int = 1000*, *time_frame: str = '1h'*, *params: Optional[tensortrade.stochastic.utils.parameters.ModelParameters] = None*) → pandas.core.frame.DataFrame

Generates price data from the OU process.

Parameters

- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data

- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.
- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The generated data frame containing the OHLCV bars.

References

[1] https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process

tensortrade.stochastic.processes.ornstein_uhlenbeck.**ornstein_uhlenbeck_levels**(*params: tensortrade.stochastic.ut* → numpy.array

Constructs the rate levels of a mean-reverting Ornstein-Uhlenbeck process.

Parameters **params** (ModelParameters) – The parameters for the stochastic model.

Returns *np.array* – The interest rate levels for the Ornstein Uhlenbeck process

tensortrade.stochastic.utils package

Submodules

tensortrade.stochastic.utils.helpers module

tensortrade.stochastic.utils.helpers.**convert_to_prices**(*param: tensortrade.stochastic.utils.parameters.ModelParameters, log_returns: numpy.array*) → numpy.array

Converts a sequence of log returns into normal returns (exponentiation) and then computes a price sequence given a starting price, param.all_s0.

Parameters

- **param** (*ModelParameters*) – The model parameters.
- **log_returns** (*np.array*) – The log returns.

Returns *np.array* – The price sequence.

tensortrade.stochastic.utils.helpers.**generate**(*price_fn: Callable[[tensortrade.stochastic.utils.parameters.Model numpy.array], base_price: int = 1, base_volume: int = 1, start_date: str = '2010-01-01', start_date_format: str = '%Y-%m-%d', times_to_generate: int = 1000, time_frame: str = '1h', params: tensortrade.stochastic.utils.parameters.ModelParameters = None*) → pandas.core.frame.DataFrame

Generates a data frame of OHLCV data based on the price model specified.

Parameters

- **price_fn** (*Callable[[ModelParameters], np.array]*) – The price function generate the prices based on the chosen model.
- **base_price** (*int, default 1*) – The base price to use for price generation.
- **base_volume** (*int, default 1*) – The base volume to use for volume generation.
- **start_date** (*str, default '2010-01-01'*) – The start date of the generated data
- **start_date_format** (*str, default '%Y-%m-%d'*) – The format for the start date of the generated data.
- **times_to_generate** (*int, default 1000*) – The number of bars to make.
- **time_frame** (*str, default '1h'*) – The time frame.
- **params** (*ModelParameters*, optional) – The model parameters.

Returns *pd.DataFrame* – The data frame containing the OHLCV bars.

tensortrade.stochastic.utils.helpers.**get_delta**(*time_frame: str*) → float

Gets the time delta for a given time frame.

Parameters **time_frame** (*str*) – The time frame for generating. (e.g. 1h, 1min, 1w, 1d)

Returns *float* – The time delta for the given time frame.

tensortrade.stochastic.utils.helpers.**scale_times_to_generate**(*times_to_generate: int, time_frame: str*) → int

Adjusts the number of times to generate the prices based on a time frame.

Parameters

- **times_to_generate** (*int*) – The number of time to generate prices.
- **time_frame** (*str*) – The time frame for generating. (e.g. 1h, 1min, 1w, 1d)

Returns *int* – The adjusted number of times to generate.

Raises ValueError – Raised if the *time_frame* provided does not match the correct format.

tensortrade.stochastic.utils.parameters module

class tensortrade.stochastic.utils.parameters.**ModelParameters**(*all_s0: float*, *all_time: int*, *all_delta: float*, *all_sigma: float*, *gbm_mu: float*, *jumps_lambda: float = 0.0*, *jumps_sigma: float = 0.0*, *jumps_mu: float = 0.0*, *cir_a: float = 0.0*, *cir_mu: float = 0.0*, *all_r0: float = 0.0*, *cir_rho: float = 0.0*, *ou_a: float = 0.0*, *ou_mu: float = 0.0*, *heston_a: float = 0.0*, *heston_mu: float = 0.0*, *heston_vol0: float = 0.0*)

Bases: `object`

Defines the model parameters of different stock price models.

Parameters

- **all_s0** (*float*) – The initial asset value.
- **all_time** (*int*) – The amount of time to simulate for.
- **all_delta** (*float*) – The rate of time. (e.g. 1/252 = daily, 1/12 = monthly)
- **all_sigma** (*float*) – The volatility of the stochastic processes.
- **gbm_mu** (*float*) – The annual drift factor for geometric brownian motion.
- **jumps_lambda** (*float, default 0.0*) – The probability of a jump happening at each point in time.
- **jumps_sigma** (*float, default 0.0*) – The volatility of the jump size.
- **jumps_mu** (*float, default 0.0*) – The average jump size.
- **cir_a** (*float, default 0.0*) – The rate of mean reversion for Cox Ingersoll Ross.
- **cir_mu** (*float, default 0.0*) – The long run average interest rate for Cox Ingersoll Ross.
- **all_r0** (*float, default 0.0*) – The starting interest rate value.
- **cir_rho** (*float, default 0.0*) – The correlation between the wiener processes of the Heston model.
- **ou_a** (*float, default 0.0*) – The rate of mean reversion for Ornstein Uhlenbeck.
- **ou_mu** (*float, default 0.0*) – The long run average interest rate for Ornstein Uhlenbeck.

- **heston_a** (*float, default 0.0*) – The rate of mean reversion for volatility in the Heston model.
- **heston_mu** (*float, default 0.0*) – The long run average volatility for the Heston model.
- **heston_vol0** (*float, default 0.0*) – The starting volatility value for the Heston model.

tensortrade.stochastic.utils.parameters.**default**(*base_price: float, t_gen: int, delta: float*) → tensortrade.stochastic.utils.parameters.ModelParameters

Creates a basic model parameter set with key parameters specified default parameters.

Parameters

- **base_price** (*float*) – The base price to use for price generation.
- **t_gen** (*int*) – The number of bars to generate.
- **delta** (*float*) – The time delta to use.

Returns *ModelParameters* – The default model parameters to use.

tensortrade.stochastic.utils.parameters.**random**(*base_price: float, t_gen: int, delta: float*) → tensortrade.stochastic.utils.parameters.ModelParameters

Creates a random model parameter set with key parameters specified default parameters.

Parameters

- **base_price** (*float*) – The base price to use for price generation.
- **t_gen** (*int*) – The number of bars to generate.
- **delta** (*int*) – The time delta to use.

Returns *ModelParameters* – The random model parameters to use.

Submodules

tensortrade.version module

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W