AutoTrader

Release 0.6.0

Kieran Mackle

Sep 26, 2023

CONTENTS

AutoTrader is a Python-based trading framework for the development, optimisation and deployment of automated trading systems. Here you will find everything you need to start algotrading with AutoTrader.

If you are new to Python, you may find the tutorials especially useful. For those who like no details spared, refer to the *complete strategy walkthrough*, where a popular MACD trading strategy is built and tested. The *condensed walkthrough* offers a more concise version of this tutorial.

If you are still deciding if AutoTrader is for you, check out the *feature showcase* to see what on AutoTrader has to offer. Otherwise, head on over to the *Getting Started* guide.

CHAPTER

ONE

SUPPORTED BROKERS AND EXCHANGES

With AutoTrader `v0.7.0`, you can access over 100 cryptocurrency exchanges thanks to the integration of CCXT. The table below summarises the connection to supported brokers.

Broker	Asset classes	Integration status	Docs page
Oanda	Forex CFDs	Complete	*link*
Interactive Brokers	Many	In progress	*link*
dYdX	Cryptocurrencies	Complete	*link*
CCXT	Cryptocurrencies	In progress	*link*

CHAPTER TWO

LATEST CHANGES

AutoTrader `v0.7.0` has been released! Make sure to check out the *changelog* when upgrading for details on the breaking changes and latest features.

CHAPTER THREE

INDEX

Looking for something specific? Try the search bar on the left, or take a look through the index.

3.1 Getting Started with AutoTrader

This page has all the information required to download and install AutoTrader.

3.1.1 Installation

AutoTrader can be installed in two ways; through PyPI or by cloning the repository directly.

PyPI Install

The easiest (and recommended) way to get AutoTrader is by running the following command.

```
pip install autotrader
```

This will download AutoTrader from the Python Package Index and install it on your machine.

Install from Source

If you are interested in developing AutoTrader, or would like to view the source code while you work, cloning from GitHub is the way to go. Simply clone the Github repository onto your machine and run `pip install` locally to install.

```
git clone https://github.com/kieran-mackle/AutoTrader
cd AutoTrader
pip install .
```

Tip: If you plan on developing AutoTrader, you can also perform an editable install to avoid re-installing the source code each time you make a change. To do so, include the '-e' flag: `pip install -e .`.

3.1.2 Optional Dependencies

The installation methods described above will install the minimum required depencencies to get AutoTrader running. You can optionally install more dependencies, depending on where you plan to trade or where you would like to download price data from.

Options include:

- dydx: to include dYdX dependencies
- ccxt: to include CCXT dependencies
- oanda: : to include Oanda v20 dependencies
- ib: to include Interactive Broker dependencies
- yfinance: to include Yahoo Finance dependencies

To install AutoTrader with any of these extra dependencies, include them in the pip install command in square brackets. For example:

```
pip install autotrader[ccxt,yfinance]
```

A note on dYdX Dependency

The package dependencies of the dYdX V3 Python interface are very tightly defined. For example, if you try to install AutoTrader with both the `ccxt` and `dydx` dependencies, pip will not be able to resolve the conflicts. As such, it is recommended that you maintain separate environments for your trading.

3.1.3 Demo Repository

To make getting started with AutoTrader even easier, download the demo repository from here. This repo contains example run files, strategies and configuration files.

```
git clone https://github.com/kieran-mackle/autotrader-demo/
```

3.2 AutoTrader Features

AutoTrader is feature rich, with everything you need to go from concept to livetrading. Use the links below to see some of the features AutoTrader has to show for.

3.2.1 Backtesting With AutoTrader

Thanks to the powerful *virtual broker*, AutoTrader features a highly capable backtesting environment. In addition to supporting mulitple *order types*, AutoTrader supports backtesting mutliple strategies with multiple instruments on multiple timeframes - all against the same broker at the same time.

Single-Bot Backtest

Shown below is the output for a backtest on EUR/USD using the MACD strategy developed in the *walkthrough*.

```
    _         _        ____             _    _            _
   / \  _   _| |_ ___ | __ )  __ _  ___| | _| |_ ___  ___| |_
  / _ \| | | | __/ _ \|  _ \ / _` |/ __| |/ / __/ _ \/ __| __|
 / ___ \ |_| | || (_) | |_) | (_| | (__|   <| ||  __/\__ \ |_
/_/   \_\__,_|\__\___/|____/ \__,_|\___|_|\_\\__\___||___/\__|



Beginning new backtest.
[*********************100%***********************]  1 of 1 completed

AutoTraderBot assigned to trade EURUSD=X with virtual broker using MACD Trend Strategy.

Trading...

Backtest complete (runtime 4.642 s).

----------------------------------------------
               Backtest Results
----------------------------------------------
Start date:              Jan 20 2021 05:00:00
End date:                Dec 31 2021 13:00:00
Starting balance:        $1000.0
Ending balance:          $1255.11
Ending NAV:              $1270.05
Total return:            $255.11 (25.5%)
Total no. trades:        96
Total fees:              $0.0
Backtest win rate:       46.9%
Maximum drawdown:        -18.1%
Max win:                 $40.5
Average win:             $26.53
Max loss:                -$43.81
Average loss:            -$18.41
Longest win streak:      6 trades
Longest losing streak:   6 trades
Average trade duration:  1 day, 2:37:30
Orders still open:       1
Cancelled orders:        3

            Summary of long trades
----------------------------------------------
Number of long trades:   40
Long win rate:           50.0%
Max win:                 $40.5
Average win:             $26.99
Max loss:                -$21.91
Average loss:            -$17.96

             Summary of short trades
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------
Number of short trades:  59
short win rate:          42.4%
Max win:                 $35.06
Average win:             $26.17
Max loss:                -$43.81
Average loss:            -$18.65
```

This output is useful, but as the saying goes, a picture is worth a thousand words. Running a backtest is no exception, as when visualising backtest results in AutoTrader, you can see exactly where the stop loss and take profit levels are being placed for each and every trade. This is incredibly useful when assessing how effective your exit strategy is. By visualising the exit targets, you can see if you are being stopped out too early on otherwise good trades. The chart interactive chart below is automatically generated using the automated plotting module, *AutoPlot*, after running a backtest.

Multi-Bot Backtest

As mentioned above, AutoTrader supports running backtests on multiple products and strategies at the same time. This is referred to as a 'multi-bot backtest', since a *trading bot* is deployed for each unique strategy trading a unique product.

Specifying Multiple Products

To trade multiple products with a strategy, simply add them to the `WATCHLIST` in the *strategy configuration*. In the example below, four FOREX pairs will be traded by the strategy they are assigned to.

```
WATCHLIST: ['EURUSD=X', 'EURJPY=X', 'EURAUD=X', 'AUDJPY=X']
```

Specifying Multiple Strategies

Trading multiple strategies is equally as straightforward; simply use the *`add_strategy`* method of the active *`AutoTrader`* instance to provide each strategy. This is shown in the snippet below.

```
at.add_strategy('macd_strategy')
at.add_strategy('ema_crossover_strategy')
```

Output

Now let's look at the backtest results for two strategies, each trading two different products. The strategies used here are a MACD trend strategy and an EMA crossover strategy - both of which can be found in the AutoTrader demo repository. As you can see in the output below, four *trading bots* are deployed:

1. to trade EUR/USD with the virtual broker using the MACD Trend Strategy;
2. to trade EUR/JPY with the virtual broker using the MACD Trend Strategy;
3. to trade AUD/JPY with the virtual broker using the EMA Crossover Strategy; and
4. to trade EUR/AUD with the virtual broker using the EMA Crossover Strategy.

Each of these bots is therefore responsible for trading a single product, with the knowledge of a single strategy. This is extremely useful, as you can see how different strategies will interact with each other when trading them at the same time.

```
    _         _        ____             _    _            _   
   / \  _   _| |_ ___ | __ )  __ _  ___| | _| |_ ___  ___| |_ 
  / _ \| | | | __/ _ \|  _ \ / _` |/ __| |/ / __/ _ \/ __| __|
 / ___ \ |_| | || (_) | |_) | (_| | (__|   <| ||  __/\__ \ |_ 
/_/   \_\__,_|\__\___/|____/ \__,_|\___|_|\_\\__\___||___/\__|


Beginning new backtest.
[*********************100%***********************]  1 of 1 completed

AutoTraderBot assigned to trade EURUSD=X with virtual broker using MACD Trend Strategy.
[*********************100%***********************]  1 of 1 completed

AutoTraderBot assigned to trade EURJPY=X with virtual broker using MACD Trend Strategy.
[*********************100%***********************]  1 of 1 completed

AutoTraderBot assigned to trade AUDJPY=X with virtual broker using EMA Crossover.
[*********************100%***********************]  1 of 1 completed

AutoTraderBot assigned to trade EURAUD=X with virtual broker using EMA Crossover.

Trading...

Warning: mismatched data lengths detected. Correcting via row reduction.
  Done.

Backtest complete (runtime 13.154 s).

----------------------------------------------
               Backtest Results
----------------------------------------------
Start date:              Aug 18 2021 14:00:00
End date:                Dec 31 2021 13:00:00
Starting balance:        $1000.0
Ending balance:          $1517.52
Ending NAV:              $1517.52
Total return:            $517.52 (51.8%)
Total no. trades:        120
Total fees:              $0.0
Backtest win rate:       42.5%
Maximum drawdown:        -13.97%
Max win:                 $57.45
Average win:             $37.77
Max loss:                -$29.58
Average loss:            -$20.41
Longest win streak:      4 trades
Longest losing streak:   7 trades
Average trade duration:  1 day, 3:59:00
Orders still open:       3
```

(continues on next page)

(continued from previous page)

```
Cancelled orders:        10

            Summary of long trades
----------------------------------------------
Number of long trades:   53
Long win rate:           34.0%
Max win:                 $57.01
Average win:             $38.86
Max loss:                -$29.58
Average loss:            -$21.38

             Summary of short trades
----------------------------------------------
Number of short trades:  70
short win rate:          47.1%
Max win:                 $57.45
Average win:             $37.17
Max loss:                -$28.79
Average loss:            -$19.58
```

Now *AutoPlot* will create a dashboard-like output, showing the performance of each bot, as well as account metrics such as net asset value and margin available for the duration of the backtest.

Tip: You can also pull out individual trading bots using the *get_bots_deployed* method to analyse them (and the trades they took) individually. You can also plot them individually using `at.plot_backtest(bot)`!

Parameter Optimisation

AutoTrader also offers (a somewhat brute-force method of) parameter optimisation for backtested strategies.

```python
from autotrader.autotrader import AutoTrader

at = AutoTrader()
at.configure(show_plot=True, verbosity=1)
at.add_strategy('macd')
at.backtest(start = '1/1/2020',
            end = '1/1/2021',
            initial_balance=1000,
            leverage = 30)
at.add_data(data_dict={'EURUSD=X': 'EUdata.csv'})
at.optimise(opt_params=['MACD_fast', 'MACD_slow'],
            bounds=[(5, 20), (20, 40)])
at.run()
```

3.2.2 Interactive Visualisation With AutoTrader

AutoTrader comes with its own automated charting module, *AutoPlot*. It is primarily used to visualise backtest results, but can also be used as a tool while developing indicators - as highlighted in this blog post

Template Script

The chart generated above was created with the script below.

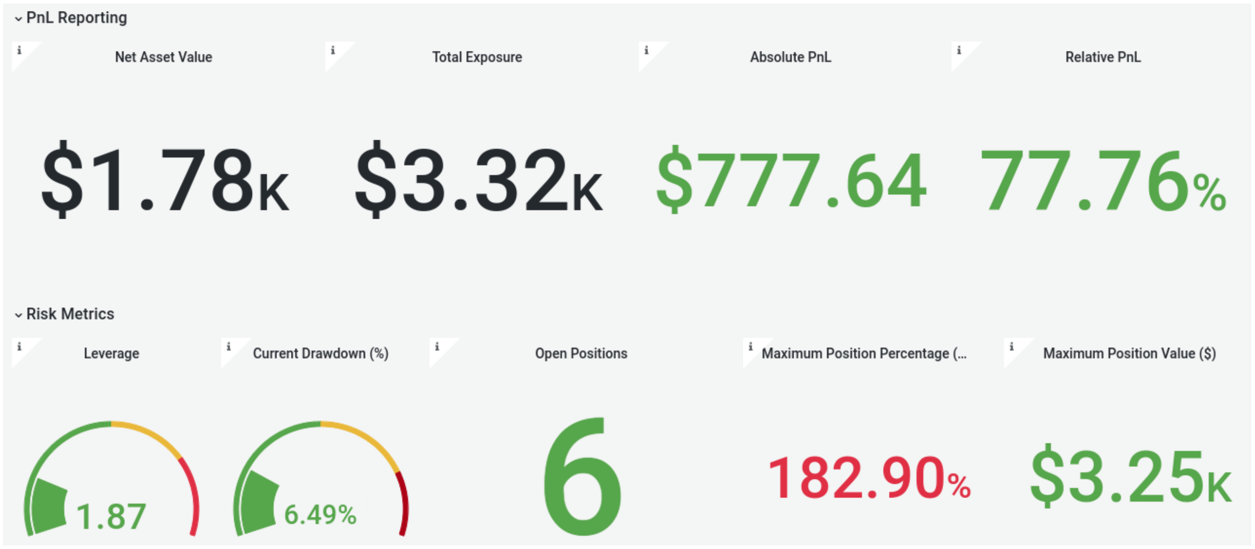
```python
from finta import TA
from autotrader.autodata import GetData
from autotrader.autoplot import AutoPlot
from autotrader.indicators import crossover, cross_values

# Instantiate GetData class
get_data = GetData()

# Get price data for EUR/USD
instrument = 'EURUSD=X'
data = get_data.yahoo(instrument, '1h',
                      start_time='2021-01-01',
                      end_time='2021-04-01')

# Calculate indicators
ema50 = TA.EMA(data, 50)
ema200 = TA.EMA(data, 200)
MACD_df = TA.MACD(data, 12, 26, 9)
MACD_CO = crossover(MACD_df.MACD, MACD_df.SIGNAL)
MACD_CO_vals = cross_values(MACD_df.MACD, MACD_df.SIGNAL, MACD_CO)

# Construct indicators dictionary
indicators = {'MACD (12/26/9)': {'type': 'MACD',
                                  'macd': MACD_df.MACD,
                                  'signal': MACD_df.SIGNAL,
                                  'crossvals': MACD_CO_vals},
            'EMA (50)': {'type': 'MA',
                          'data': ema50},
            'EMA (200)': {'type': 'MA',
                           'data': ema200},
            'MACD Crossovers': {'type': 'below',
                                 'data': MACD_CO}}

# Instantiate AutoPlot and plot
ap = AutoPlot(data)
ap.plot(indicators=indicators, instrument=instrument)
```

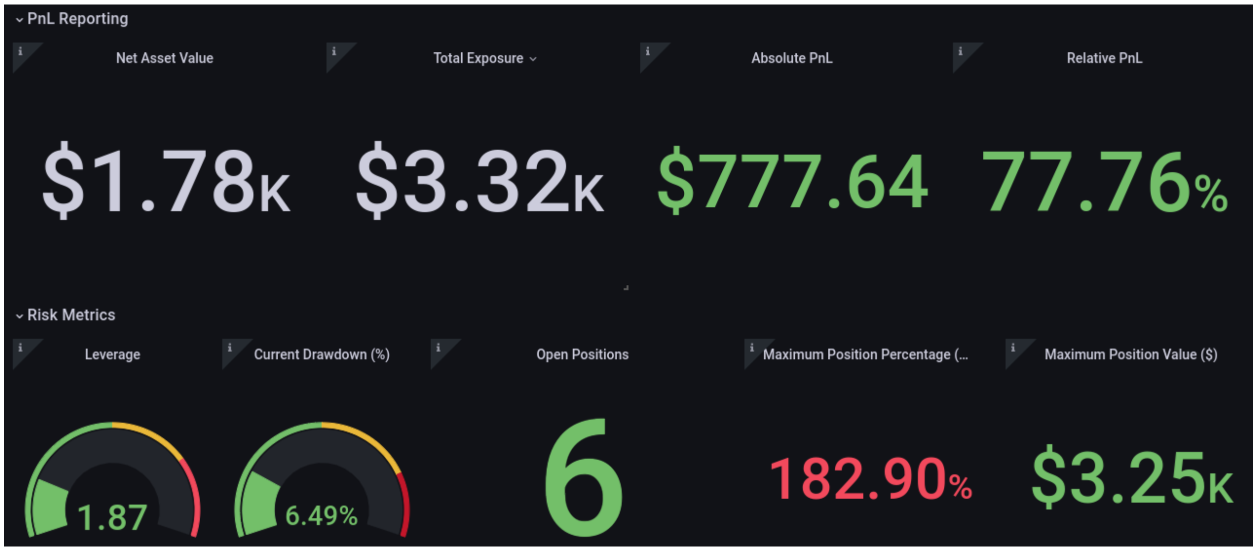
3.2.3 Live Trading With AutoTrader

What use would backtesting be if you can't take the same strategy live? This is no concern when you use AutoTrader, which features a seamless transition from backtesting to livetrading. Simply run your strategy through AutoTrader with `environment="live"`, and it will be live. Simple as that.

Live Trade Monitoring

AutoTrader also supports monitoring via Prometheus and Grafana. A template dashboard can be found in the `templates` directory of the AutoTrader repository, and is previewed below.





Supported Brokers

Check out the *supported brokers and exchanges*.

3.2.4 Data Pipelines in AutoTrader

AutoTrader supports custom data pipelines with the *DataStream* class utility. This is particularly useful when you plan to trade multiple contracts on the same underlying product, where you want a single trading bot to trade multiple contracts. This is possible with a custom DataStream.

3.2.5 AutoTrader Indicators Library

AutoTrader features a rich *custom indicator library*, with novel indicators to help you gain an edge. The chart below showcases the Halftrend indicator, ported from the indicator by *everget* on TradingView.

3.3 AutoTrader Overview

This page provides a high level overview of using AutoTrader.

3.3.1 User Inputs

At a minimum, AutoTrader requires you to provide a trading *strategy*. Given some data, your strategy is expected to output trading signals in the form of `Order`s. This is normally enough for you to run a backtest on your strategy, but if you would like to trade live, you will also need to provide your trading account API keys in a `keys.yaml` file. It is also recommended that you bring your own data, but this isn't essential.

Note: With AutoTrader `v0.7`, you can also click trade. This means you can connect to your broker without providing a strategy. See the tutorial *here* for more information.

3.3.2 General Deployment Process

To make things go as smoothly as possible, you should generally set up your instance of AutoTrader by calling the methods in the following order.

1. Create the AutoTrader instance

```
at = AutoTrader()
```

2. Configure the AutoTrader instance

```
at.configure()
```

3. (Optional) Add a trading strategy (or strategies)

```
at.add_strategy()
```

4. (Optional) Add data

```
at.add_data()
```

4. (Optional) Configure virtual account (for backtesting and paper trading)

```
at.virtual_account_config()
```

5. (Optional) Configure backtest (for backtesting)

```
at.backtest()
```

6. Run AutoTrader

```
at.run()
```

3.3.3 Recommended Directory Organisation

If you are just getting started with AutoTrader, it is recommended that you use the directory structure shown below when developing your strategies. This structure includes a `config/` directory (containing your configuration files) and a `strategies/` directory (containing your *trading strategies*). When you run AutoTrader, it will look for the appropriate files under these directories. If you cloned the demo repository, you will see these directories set up already.

Tip: You can use the AutoTrader Command Line Interface to initialise your trading directory to this structure using `autotrader init`.

```
your_trading_project/
├── runfile.py                  # Run script to deploy trading bots
├── config
│   ├── GLOBAL.yaml             # Global configuration file
│   ├── strategy1_config.yaml   # Strategy 1 configuration file
│   └── strategy2_config.yaml   # Strategy 2 configuration file
├── price_data
|   ├── dataset1.csv            # Local OHLC dataset
│   └── dataset2.csv            # Another dataset
└── strategies
    ├── strategy1.py            # Strategy 1 module, containing strategy 1 logic
    └── strategy2.py            # Strategy 2 module, containing strategy 2 logic
```

3.4 Trading Environments

There are two trading environments: `paper` and `live`. The environment being used can be specified in the `configure` method, but it will be overwritten to `paper` any time you call the *virtual account configuration* method.

3.4.1 Paper Trading

`environment="paper"`

Paper trading can fall into one of two categories:

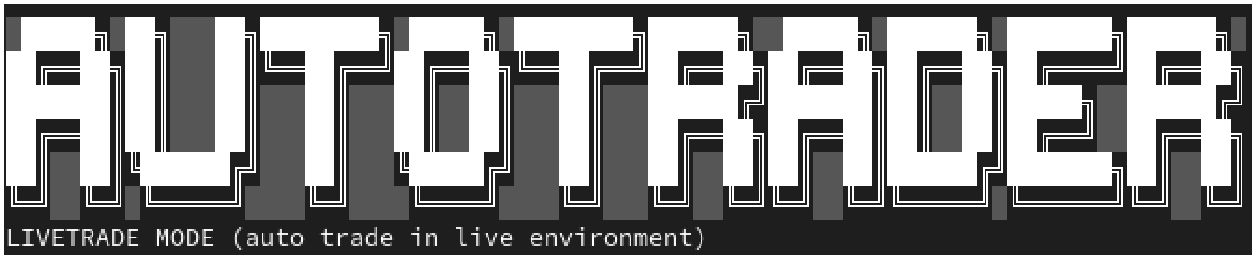
- fully simulated trading via the *virtual broker*, or;
- demo trading via an exchange/broker's dedicated demo API.

If the exchange you wish to use offers a "demo" trading API (or testnet on crypto-specific exchanges), this can be a good way to test that all the required functionality of your strategy is supported by the exchange. It should always be an intermediate step before deploying your strategy live. Simply make sure that you have provided the appropriate API keys in your *account configuration file*.

In some instances, the liquidity available on testnet API's is not reflective of that on the live exchange. In this case, you will not get a good indication of your strategy's performance. For this purpose, using the capabilities of the *virtual broker* to simulate trading is the way to go. To activate this functionality, simply configure a virtual trading account via the *virtual account configuration* method.

3.4.2 Live Trading

When you are ready to deploy your strategy with real money, set the `environment` argument in the `configure` method to `live`. This will switch all of the API pointers to the live endpoints, and fire your orders to the live platforms. When Autotrader is running in livetrade mode, you will see this indicated as shown below.



```
LIVETRADE MODE (auto trade in live environment)
```

3.5 Condensed AutoTrader Walkthrough

This page is a condensed version of the *detailed walkthrough*, which goes through the process of building and running a strategy in AutoTrader. If you are familiar with Python, it should be sufficient to get you up and running.

Tip: The code for the MACD crossover strategy shown in this tutorial can be found in the demo repository.

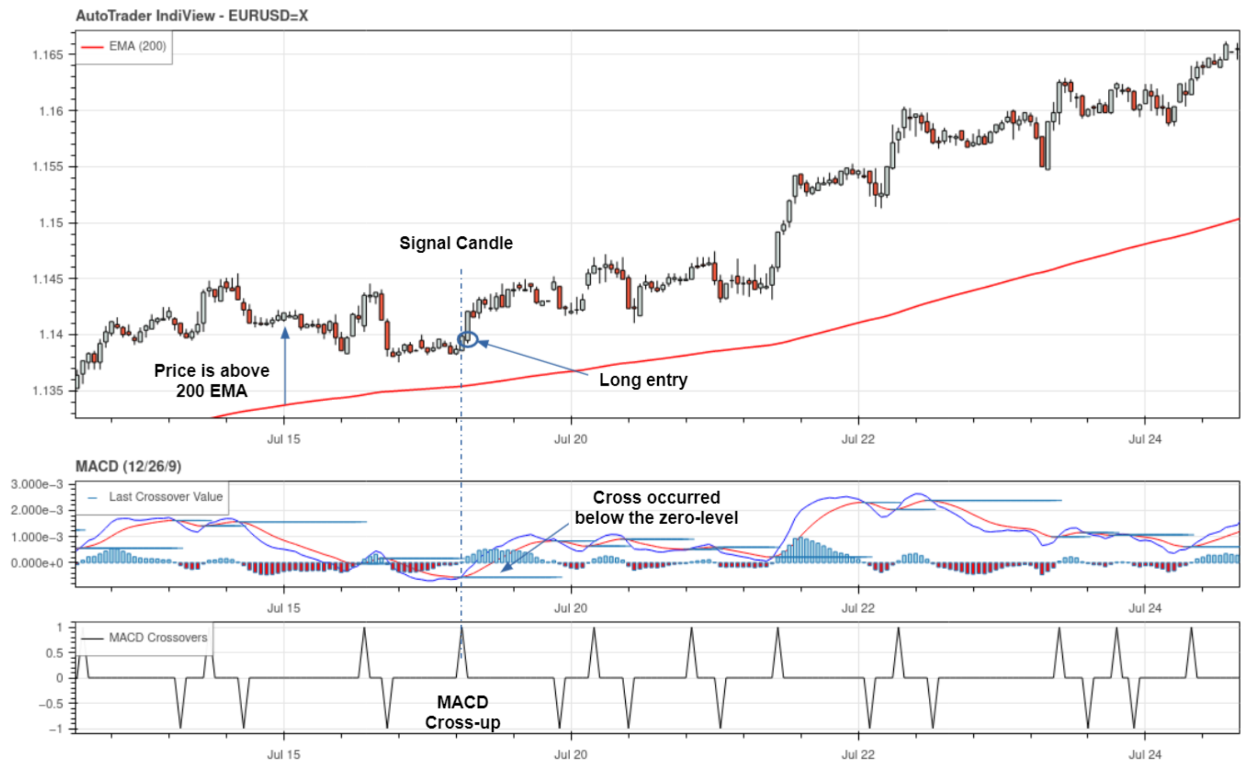
3.5.1 Strategy Rules

The rules for the MACD crossover strategy are as follows.

1. Trade in the direction of the trend, as determined by the 200EMA.
2. Enter a long position when the MACD line crosses *up* over the signal line, and enter a short when the MACD line crosses *down* below the signal line.
3. To ensure only the strongest MACD signals, the crossover must occur below the histogram zero line for long positions, and above the histogram zero line for short positions.

4. Stop losses are set at recent price swings/significant price levels.
5. Take profit levels are set at 1:1.5 risk-to-reward.

An example of a long entry signal from this strategy is shown in the image below (generated using *AutoTrader IndiView*).



3.5.2 Strategy Construction

Strategies in AutoTrader are built as class objects. They contain the logic required to transform data into a trading signals. Generally speaking, a strategy class will be instantiated with the name of the instrument being traded and the strategy parameters, but you can customise what gets passed in using the *strategy configuration*.

Configuration

Follow Along

Follow along in the demo repository: config/macd.yaml

The *strategy configuration* file defines all strategy parameters and instruments to trade with the strategy. The PARAMETERS of this file will be passed into your strategy for you to use there.

```
# macd.yaml
NAME: 'Simple Macd Strategy'    # strategy name
MODULE: 'macd'                  # strategy module
CLASS: 'SimpleMACD'             # strategy class
INTERVAL: '1h'                  # stategy timeframe
PERIOD: 300                     # candles required by strategy
```

(continues on next page)

(continued from previous page)

```
SIZING: 'risk'                  # sizing method
RISK_PC: 1.5                    # risk per trade (%)
PARAMETERS:                     # strategy parameters
  ema_period: 200
  MACD_fast: 12
  MACD_slow: 26
  MACD_smoothing: 9

  # Exit level parameters
  RR: 1.5

WATCHLIST: ['EURUSD=X']         # strategy watchlist
```

Class Object

Although strategy construction is extremely flexible, the class **must** contain an `__init__` method, and a method named `generate_signal`. The first of these methods is called whenever the strategy is instantiated.

By default, strategies in AutoTrader are instantiated with three named arguments:

1. The name of the instrument being traded in this specific instance (`instrument`).
2. The strategy parameters (`parameters`)
3. The trading instruments data (`data`)

When backtesting, the `data` provided to `__init__` is for the entire backtest period. This allows you to calculate all indicators for plotting purposes down the line, but it shouldn't be used in the `generate_signal` method, as this could introduce look-ahead.

Aside from the `__init__` method, your strategy must have a method named `generate_signal`. This method gets called by AutoTrader everytime new data becomes available, and expects a trading *Order* in return.

A long order can be created by specifying `direction=1` when creating the `Order`, whereas a short order can be created by specifying `direction=-1`. If there is no trading signal this update, you can create an *empty order* with just `Order()`. We also define our exit targets by the `stop_loss` and `take_profit` arguments. The strategy below uses the `generate_exit_levels` helper method to calculate these prices.

Tip: Take a look at the template strategy provided in the Github repository.

```python
# macd.py
from finta import TA
from autotrader import Order, indicators


class SimpleMACD:
    """Simple MACD Strategy

    Rules
    ------
    1. Trade in direction of trend, as per 200EMA.
    2. Entry signal on MACD cross below/above zero line.
    3. Set stop loss at recent price swing.
```

(continues on next page)

(continued from previous page)

```
    4. Target 1.5 take profit.
    """

    def __init__(self, parameters, data, instrument):
        """Define all indicators used in the strategy.
        """
        self.name = "Simple MACD Trend Strategy"
        self.params = parameters
        self.instrument = instrument

        # Initial feature generation (for plotting only)
        self.generate_features(data)

        # Construct indicators dict for plotting
        self.indicators = {'MACD (12/26/9)': {'type': 'MACD',
                                              'macd': self.MACD.MACD,
                                              'signal': self.MACD.SIGNAL},
                           'EMA (200)': {'type': 'MA',
                                         'data': self.ema}
                        }

    def generate_features(self, data):
        """Updates MACD indicators and saves them to the class attributes."""
        # Save data for other functions
        self.data = data

        # 200EMA
        self.ema = TA.EMA(self.data, self.params['ema_period'])

        # MACD
        self.MACD = TA.MACD(self.data, self.params['MACD_fast'],
                            self.params['MACD_slow'], self.params['MACD_smoothing'])
        self.MACD_CO = indicators.crossover(self.MACD.MACD, self.MACD.SIGNAL)
        self.MACD_CO_vals = indicators.cross_values(self.MACD.MACD,
                                                    self.MACD.SIGNAL,
                                                    self.MACD_CO)

        # Price swings
        self.swings = indicators.find_swings(self.data)

    def generate_signal(self, data):
        """Define strategy to determine entry signals."""
        # Feature calculation
        self.generate_features(data)

        # Look for entry signals (index -1 for the latest data)
        if self.data.Close.values[-1] > self.ema[-1] and \
            self.MACD_CO[-1] == 1 and \
            self.MACD_CO_vals[-1] < 0:
                # Long entry signal detected! Calculate SL and TP prices
                stop, take = self.generate_exit_levels(signal=1)
                new_order = Order(direction=1, stop_loss=stop, take_profit=take)
```

(continues on next page)

(continued from previous page)

```
        elif self.data.Close.values[-1] < self.ema[-1] and \
            self.MACD_CO[-1] == -1 and \
            self.MACD_CO_vals[-1] > 0:
                # Short entry signal detected! Calculate SL and TP prices
                stop, take = self.generate_exit_levels(signal=-1)
                new_order = Order(direction=-1, stop_loss=stop, take_profit=take)

        else:
            # No trading signal, return a blank Order
            new_order = Order()

        return new_order

    def generate_exit_levels(self, signal):
        """Function to determine stop loss and take profit prices."""
        RR = self.params['RR']
        if signal == 1:
            # Long signal
            stop = self.swings.Lows[-1]
            take = self.data.Close[-1] + RR*(self.data.Close[-1] - stop)
        else:
            # Short signal
            stop = self.swings.Highs[-1]
            take = self.data.Close[-1] - RR*(stop - self.data.Close[-1])
        return stop, take
```

3.5.3 Backtesting

An easy and organised way to deploy a trading bot is to set up a run file. Here you import AutoTrader, configure the run settings and deploy your bot. This is all achieved in the example below.

```
# runfile.py
from autotrader import AutoTrader

at = AutoTrader()
at.configure(show_plot=True, verbosity=1, feed='yahoo',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.backtest(start = '1/1/2021', end = '1/1/2022')
at.virtual_account_config(leverage=30)
at.run()
```

Let's dive into this a bit more:

- We begin by importing AutoTrader and creating an instance using `at = AutoTrader()`.
- Next, we use the *`configure`* method to set the verbosity of the code and tell AutoTrader that you would like to see the backtest plot. We also define the *run mode* and update interval to 1h, meaning that we will step through the backtest data by 1 hour at a time.
- Next, we add our strategy using the `add_strategy` method. Here we pass the file prefix of the strategy configuration file, located (by default) in the `config/` *directory*. Since our strategy configuration file is named

macd.yaml, we pass in 'macd'.

- We then use the *backtest* method to define the backtest period. In this example, we set the start and end dates of the backtest.
- Since we will be simulating trading (by backtesting), we also need to configure the virtual trading account. We do this with the virtual_account_config method. Here we set the account leverage to 30. You can also configure trading costs, bid/ask spread, initial balance and other settings here.
- Finally, we run AutoTrader with the command at.run().

Simply run this file, and AutoTrader will do its thing.

Backtest Results

With a verbosity of 1, you will see an output similar to that shown below. As you can see, there is a detailed breakdown of trades taken during the backtest period. Since we told AutoTrader to plot the results, you will also see the interactive chart shown below.

```
    ___         __        ______              __
   /   | __  __/ /_____  /_  __/________ _____/ /__  _____
  / /| |/ / / / __/ __ \  / / / ___/ __ `/ __  / _ \/ ___/
 / ___ / /_/ / /_/ /_/ / / / / /  / /_/ / /_/ /  __/ /
/_/  |_\__,_/\__/\____/ /_/ /_/   \__,_/\__,_/\___/_/


[*********************100%***********************]  1 of 1 completed
BACKTEST MODE

AutoTraderBot assigned to trade EURUSD=X with virtual broker using Simple Macd Strategy.

Trading...

31539600.0it [00:19, 1630112.41it/s]
Backtest complete (runtime 19.348 s).

----------------------------------------------
               Trading Results
----------------------------------------------
Start date:              Jan 20 2021 04:00:00
End date:                Dec 31 2021 13:00:00
Duration:                345 days 09:00:00
Starting balance:        $1000.0
Ending balance:          $1140.75
Ending NAV:              $1170.16
Total return:            $140.75 (14.1%)
Maximum drawdown:        -18.97%
Total no. trades:        175
Total fees paid:         $0.0
Win rate:                21.7%
Max win:                 $36.51
Average win:             $25.26
Max loss:                -$21.57
Average loss:            -$16.38
Longest winning streak:  4 trades
```

(continues on next page)

(continued from previous page)

```
Longest losing streak:   11 trades
Average trade duration:  1 day, 3:43:38
Positions still open:    1
Cancelled orders:        5

            Summary of long trades
----------------------------------------------
Number of long trades:   36
Win rate:                41.7%
Max win:                 $36.51
Average win:             $25.22
Max loss:                -$21.18
Average loss:            -$17.23

             Summary of short trades
----------------------------------------------
Number of short trades:  54
Win rate:                42.6%
Max win:                 $31.85
Average win:             $25.28
Max loss:                -$21.57
Average loss:            -$15.86
```

3.5.4 Going Live

Taking a strategy live is as easy as changing a few lines in your runfile. Say you would like to trade your strategy on the cryptocurrency exchange dYdX. Then, all you need to do is specify this as the `broker` in the `configure` method, as shown below. You will just need to make sure you have provided the relevant API keys in your `keys.yaml` file to connect to your exchange.

```python
from autotrader import AutoTrader

at = AutoTrader()
at.configure(verbosity=1, broker='dydx',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.run()
```

What if you wanted to paper trade your strategy before putting real money into it? Simply configure a virtual trading account and specify the exchange as `dydx` (or whatever `broker` you specify in `configure`) and then you will be paper trading! Doing this, AutoTrader's virtual broker mirrors the real-time orderbook of the exchange specified, making execution of orders as accurate as possible.

```python
from autotrader import AutoTrader

at = AutoTrader()
at.configure(verbosity=1, broker='dydx',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.virtual_account_config(leverage=30, exchange='dydx')
at.run()
```

3.6 Detailed AutoTrader Walkthrough

A popular MACD crossover strategy will be developed in this section. An important note here is that this strategy assumes that the trading instrument can be short-sold.

Warning: This walkthrough is intentionally very thorough and detailed. If you don't mind g lossing over the finer details, or are comfortable programming in Python, the *condensed walkthrough* might be better suited to you.

3.6.1 Strategy Rules

The rules for the MACD strategy are defined as follows.

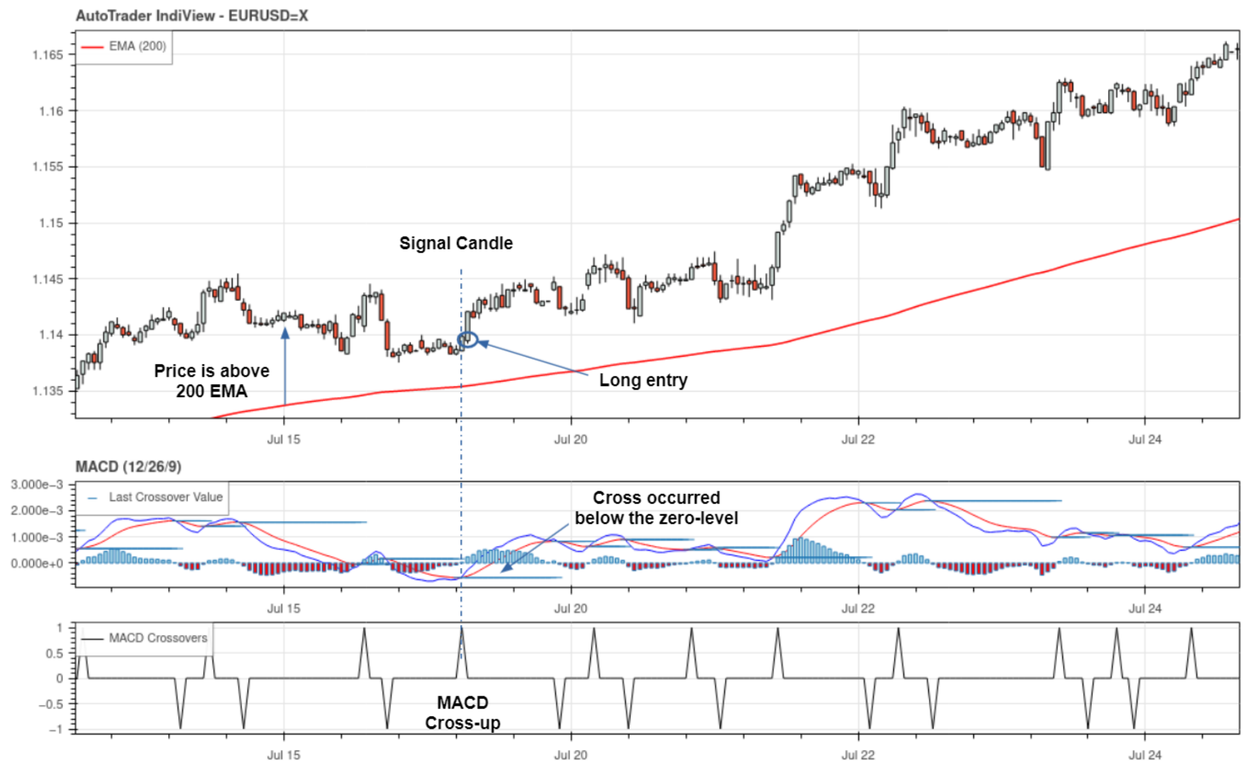
1. Trade in the direction of the trend, as determined by the 200-period exponential moving average (EMA) (if the current price above 200EMA, there is an up-trend and only long trades should be made, and if price is below, there is a down-trend and only short trades should be made).
2. Enter a long position when the MACD line crosses *up* over the signal line, and enter short when the MACD line crosses *down* below the signal line.
3. To ensure only the strongest signals, the MACD crossover must occur below the histogram zero line for long positions, and above the histogram zero line for short positions.
4. Stop losses are set at recent price swings/significant price levels. We will use AutoTrader's *swing detection* indicator for this.
5. Take profit levels are set at 1:1.5 risk-to-reward, meaning that winning trades make 1.5 times more than losing trades lose.

Strategy Parameters

From these rules, the following strategy parameters can be defined:

Parameter	Nominal value
ema_period	200
MACD_fast	12
MACD_slow	26
MACD_smoothing	9
RR	1.5

An example of a long entry signal from this strategy is shown in the image below (generated using *AutoTrader IndiView*).



Building a Strategy

So you have an idea for a trading strategy. How do you code this up to use with AutoTrader? Read on to find out, where we code the MACD strategy described *here*.

Tip: The code for the MACD crossover strategy shown in this tutorial can be found in the demo repository.

Strategy Construction

Strategies in AutoTrader are built as class objects. They contain the logic required to transform data into a trading signals. Generally speaking, a strategy class will be instantiated with the name of the instrument being traded and the strategy parameters, but you can customise what gets passed in using the *strategy configuration*.

Strategy Configuration

Follow Along

Follow along in the demo repository: config/macd.yaml

Lets start by writing the *strategy configuration* `.yaml` file. Call this `macd.yaml`, refering to our MACD strategy. This file is a convenient place to define your strategy parameters (using the `PARAMETERS` key) and which instruments to trade using this strategy (using the `WATCHLIST` key). It is also used to tell AutoTrader where to find your strategy - the `MODULE` key defines the prefix of the file where your strategy is written, and the `CLASS` key defines the actual class name of your strategy. You can also define the strategy configuration using a dictionary instead of a yaml file, and pass that in when *adding your strategy*.

By separating the strategy parameters from the strategy iteself, you are able to easily tweak the strategy to your liking, or perform hyperparameter optimisation.

Attention: YAML cares about whitespace; each nested key must be indented by two spaces more than its parent.

We will put our *strategy parameters* under the PARAMETERS key. You can call these parameters whatever you want, and that is how they will appear in your strategy. As you can see below, we define the EMA period by the ema_period key, the MACD settings by the MACD_fast, MACD_slow and MACD_smoothing keys, and the risk-to-reward ratio is by the RR key.

```yaml
# macd.yaml
NAME: 'Simple Macd Strategy'    # strategy name
MODULE: 'macd'                  # strategy module
CLASS: 'SimpleMACD'             # strategy class
INTERVAL: '1h'                  # stategy timeframe
PERIOD: 300                     # candles required by strategy
SIZING: 'risk'                  # sizing method
RISK_PC: 1.5                    # risk per trade (%)
PARAMETERS:                     # strategy parameters
  ema_period: 200
  MACD_fast: 12
  MACD_slow: 26
  MACD_smoothing: 9

  # Exit level parameters
  RR: 1.5

WATCHLIST: ['EURUSD=X']         # strategy watchlist
```

This file is read by AutoTrader and passed into your strategy when it is instantiated. We will start by backtesting on the EUR/USD currency pair alone, as specified by the WATCHLIST key. Note that the format of the instruments provided here must match your data feed (in this case, Yahoo Finance, which denotes FX with '=X').

It is worth noting that we are taking advantage of AutoTrader's automatic position size calculation, by defining the SIZING: 'risk' and RISK_PC: 1.5 keys. These keys tell AutoTrader to use a risk-based approach to position sizing. As such, when an order is submitted from the strategy, AutoTrader will use the current price and stop-loss price to calculate the appropriate position size, capping the maximium loss to the percentage defined by RISK_PC. In this case, any single trade can only ever lose 1.5% of the account.

We also define the INTERVAL: '1h' key, meaning that our strategy will run on the 1-hour timeframe. This value is used when retrieving price data through *AutoData*. This is discussed more in the next section.

Tip: You can find a template strategy configuration file in the Github repository.

Strategy Class

Follow Along

Follow along in the demo repository: strategies/macd.py

Now we can write the *strategy class* `SimpleMACD`, and place it in a module (Python file) called `macd.py` (hence `MODULE: 'macd'` and `CLASS: 'SimpleMACD'` in our strategy configuration). Although strategy construction is extremely flexible, the class **must** contain an `__init__` method, and a method named `generate_signal`. The first of these methods is called whenever the strategy is instantiated.

By default, strategies in AutoTrader are instantiated with three named arguments:

1. The name of the instrument being traded in this specific instance (`instrument`).
2. The strategy parameters (`parameters`)
3. The trading instruments data (`data`)

When backtesting, the `data` provided is for the entire backtest period. This is partly from older versions of AutoTrader, but it remains this way to allow you to calculate all indicators for plotting purposes (as you will see below). You shouldn't use this data in your strategy, however, since you risk introducing look-ahead.

Template Strategy

We will start by filling out the template strategy shown below.

```python
from autotrader.brokers.trading import Order


class Strategy:
    def __init__(self, instrument, parameters, **kwargs):
        """Initialise the strategy."""
        self.name = "Template Strategy"
        self.instrument = instrument
        self.params = parameters

        # Construct indicators dict for plotting
        self.indicators = {
            "Indicator Name": {"type": "indicatortype", "data": "indicatordata"},
        }

    def generate_signal(self, data):
        """Define strategy logic to transform data into trading signals."""

        # Initialise empty order list
        orders = []

        # The data passed into this method is the most up-to-date data.

        # Example long market order:
        long_market_order = Order(direction=1)
        orders.append(long_market_order)
```

(continues on next page)

(continued from previous page)

```
        # Example short limit order:
        short_limit = Order(direction=-1, order_type="limit", order_limit_price=1.0221)
        orders.append(short_limit)

        # Return any orders generated
        # If no orders are generated, return an empty list [], an empty dict {},
        # or a blank order Order().
        return orders
```

Instantiation

For the MACD crossover strategy, instantiation will look as shown below. Note the following:

- we save the instrument and strategy parameters to the class attributes (using the `self` variable).
- the finta technical analysis package is used to calculate the 200-period EMA and the MACD. This calculation happens in a method called `generate_features`, defined later.
- the custom indicator `crossover` is used from the *built-in indicators*.
- an `indicators` dictionary is defined to tell *AutoPlot* which indicators to plot, and what to name them.

```
from finta import TA
from autotrader import Order, indicators


class SimpleMACD:
    """Simple MACD Strategy

    Rules
    ------
    1. Trade in direction of trend, as per 200EMA.
    2. Entry signal on MACD cross below/above zero line.
    3. Set stop loss at recent price swing.
    4. Target 1.5 take profit.
    """

    def __init__(self, parameters, data, instrument):
        """Define all indicators used in the strategy.
        """
        self.name = "Simple MACD Trend Strategy"
        self.params = parameters
        self.instrument = instrument

        # Initial feature generation (for plotting only)
        self.generate_features(data)

        # Construct indicators dict for plotting
        self.indicators = {'MACD (12/26/9)': {'type': 'MACD',
                                              'macd': self.MACD.MACD,
                                              'signal': self.MACD.SIGNAL},
                           'EMA (200)': {'type': 'MA',
```

(continues on next page)

(continued from previous page)

```
                                  'data': self.ema}
                         }
```

Strategy Signals

The next step is to define the signal generation function, `generate_signal`. Make sure to use this name for all your strategies, as AutoTrader will call it when you run it. This method is where the logic of the strategy sits. It gets passed some data using the `data` argument, which you can use to generate any trading signals. This data will always be the most up-to-date data, so you should only create trading signals for the latest information it provides. In most cases, this data will be OHLC data, in the form of a Pandas DataFrame.

Important: Strategies must contain a `generate_signal` method! AutoTrader calls this method to get trading signals from your strategy.

For the MACD strategy, this method translates the *strategy rule* into code.

AutoTrader supports multiple *order types*, multiple *stop loss types* and anything else you might encounter with your live-trade broker. To create a new order, we can use the `Order` object, imported from the `autotrader.trading` module. For this strategy, we will only be placing market orders when we get the entry signal, which are the default order type.

A long order can be created by specifying `direction=1` when creating the `Order`, whereas a short order can be created by specifying `direction=-1`. If there is no trading signal this update, you can create an *empty order* with just `Order()`. We also define our exit targets by the `stop_loss` and `take_profit` arguments. These price targets come from our exit strategy, defined in the next section.

```python
def generate_features(self, data):
    """Updates MACD indicators and saves them to the class attributes."""
    # Save data for other functions
    self.data = data

    # 200EMA
    self.ema = TA.EMA(self.data, self.params['ema_period'])

    # MACD
    self.MACD = TA.MACD(self.data, self.params['MACD_fast'],
                        self.params['MACD_slow'], self.params['MACD_smoothing'])
    self.MACD_CO = indicators.crossover(self.MACD.MACD, self.MACD.SIGNAL)
    self.MACD_CO_vals = indicators.cross_values(self.MACD.MACD,
                                                self.MACD.SIGNAL,
                                                self.MACD_CO)

    # Price swings
    self.swings = indicators.find_swings(self.data)


def generate_signal(self, data):
    """Define strategy to determine entry signals."""
    # Feature calculation
    self.generate_features(data)

```

(continues on next page)

(continued from previous page)

```
    # Look for entry signals (index -1 for the latest data)
    if self.data.Close.values[-1] > self.ema[-1] and \
        self.MACD_CO[-1] == 1 and \
        self.MACD_CO_vals[-1] < 0:
            # Long entry signal detected! Calculate SL and TP prices
            stop, take = self.generate_exit_levels(signal=1)
            new_order = Order(direction=1, stop_loss=stop, take_profit=take)

    elif self.data.Close.values[-1] < self.ema[-1] and \
        self.MACD_CO[-1] == -1 and \
        self.MACD_CO_vals[-1] > 0:
            # Short entry signal detected! Calculate SL and TP prices
            stop, take = self.generate_exit_levels(signal=-1)
            new_order = Order(direction=-1, stop_loss=stop, take_profit=take)

    else:
        # No trading signal, return a blank Order
        new_order = Order()

    return new_order
```

Exit Signals

As with any good strategy, we must define an exit strategy to manage our risk. In this strategy, stop losses are set at recent swings in price. Since this is a trend following strategy, market structure tells us that price is unlikely to break past a recent swing level, unless of course the trend is reversing. The `find_swings` indicator built into AutoTrader's *indicator library* makes this an easy task. As per our rules, take profits are set at 1:1.5 risk-to-reward ratio. This is all completed with the code below, which returns our target exit prices, used by our signal generation method in the section above.

```
def generate_exit_levels(self, signal):
    """Function to determine stop loss and take profit prices."""
    RR = self.params['RR']
    if signal == 1:
        # Long signal
        stop = self.swings.Lows[-1]
        take = self.data.Close[-1] + RR*(self.data.Close[-1] - stop)
    else:
        # Short signal
        stop = self.swings.Highs[-1]
        take = self.data.Close[-1] - RR*(stop - self.data.Close[-1])
    return stop, take
```

Time to Trade

You now have your very own algorithmic trading strategy ready to unleash on the world. In the next few pages, you will learn how to backtest, optimise and deploy your strategies. But first, a quick aside on data management.

Data Management

AuotTrader is compatible with all kinds of price data, regardless of the nature of the instrument being traded. This means that it can be used for stocks, cryptocurrencies, foreign exchange, futures, options, commodities and even Mars bars - provided that you can get historical price data for them.

Luckily for you, AutoTrader is ready to automatically fetch data for various instruments from all of the *supported brokers and exchanges* using *AutoData*. All you have to do is provide a few details in your *configuration files*, and AutoTrader will take care of the rest.

Of course, if you would prefer to provide your own data, you can do this too. The default search path for local data is the `price_data` *directory*.

For the MACD Strategy

For our MACD strategy, we have already done everything we need to do to automatically download price data for EUR/USD. Recall the `INTERVAL` and `WATCHLIST` keys of our *strategy configuration file*:

```
# macd.yaml
INTERVAL: '1h'
WATCHLIST: ['EURUSD=X']
```

That's all we need to specify to automatically get 1-hour bars of price data from Yahoo Finance API. Just note that the way the instrument is written (eg. 'EURUSD=X') will depend on the data feed you are using. In this case we will be using the Yahoo Finance API. As such, we must specify EUR/USD exactly as it appears on the Yahoo Finance website.

Now that we have a way to pass data to our strategy, we are ready to start backtesting!

Backtesting with AutoTrader

Now that you have a strategy, you can have some fun with backtesting.

Creating a Runfile

An easy and organised way to deploy a trading bot is to set up a run file. Here you import AutoTrader, configure the run settings and deploy your bot. This is all achieved in the example below.

```
# runfile.py
from autotrader import AutoTrader

at = AutoTrader()
at.configure(show_plot=True, verbosity=1, feed='yahoo',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.backtest(start = '1/1/2021', end = '1/1/2022')
```

(continues on next page)

(continued from previous page)

```
at.virtual_account_config(leverage=30)
at.run()
```

Let's dive into this a bit more:

- We begin by importing AutoTrader and creating an instance using `at = AutoTrader()`.
- Next, we use the `configure` method to set the verbosity of the code and tell AutoTrader that you would like to see the backtest plot. We also define the *run mode* and update interval to `1h`, meaning that we will step through the backtest data by 1 hour at a time.
- Next, we add our strategy using the `add_strategy` method. Here we pass the file prefix of the strategy configuration file, located (by default) in the `config/` *directory*. Since our strategy configuration file is named `macd.yaml`, we pass in 'macd'.
- We then use the `backtest` method to define the backtest period. In this example, we set the start and end dates of the backtest.
- Since we will be simulating trading (by backtesting), we also need to configure the virtual trading account. We do this with the `virtual_account_config` method. Here we set the account leverage to 30. You can also configure trading costs, bid/ask spread, initial balance and other settings here.
- Finally, we run AutoTrader with the command `at.run()`.

Simply run this file, and AutoTrader will do its thing.

Backtest Results

With a verbosity of 1, you will see an output similar to that shown below. As you can see, there is a breakdown of trades taken during the backtest period. Since we told AutoTrader to plot the results, you will also see the interactive chart shown *below*.

Performance Breakdown

```
    ___         __        ______               __
   /   | __  __/ /_____  /_  __/________ _____/ /__  _____
  / /| |/ / / / __/ __ \  / / / ___/ __ `/ __  / _ \/ ___/
 / ___ / /_/ / /_/ /_/ / / / / /  / /_/ / /_/ /  __/ /
/_/  |_\__,_/\__/\____/_/ /_/   \__,_/\__,_/\___/_/


[*********************100%***********************]  1 of 1 completed
BACKTEST MODE

AutoTraderBot assigned to trade EURUSD=X with virtual broker using Simple Macd Strategy.

Trading...

31539600.0it [00:19, 1630112.41it/s]
Backtest complete (runtime 19.348 s).

----------------------------------------------
               Trading Results
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------
Start date:              Jan 20 2021 04:00:00
End date:                Dec 31 2021 13:00:00
Duration:                345 days 09:00:00
Starting balance:        $1000.0
Ending balance:          $1140.75
Ending NAV:              $1170.16
Total return:            $140.75 (14.1%)
Maximum drawdown:        -18.97%
Total no. trades:        175
Total fees paid:         $0.0
Win rate:                21.7%
Max win:                 $36.51
Average win:             $25.26
Max loss:                -$21.57
Average loss:            -$16.38
Longest winning streak:  4 trades
Longest losing streak:   11 trades
Average trade duration:  1 day, 3:43:38
Positions still open:    1
Cancelled orders:        5

            Summary of long trades
----------------------------------------------
Number of long trades:   36
Win rate:                41.7%
Max win:                 $36.51
Average win:             $25.22
Max loss:                -$21.18
Average loss:            -$17.23

             Summary of short trades
----------------------------------------------
Number of short trades:  54
Win rate:                42.6%
Max win:                 $31.85
Average win:             $25.28
Max loss:                -$21.57
Average loss:            -$15.86
```

Interactive Chart

The interactive chart will look something like the one shown below.

Accessing Backtest Data

We can also access even more details related to the backtest. Without getting into too many details, every time AutoTrader is run, it deploys one trading bot per instrument in the watchlist, per strategy. In our example, we only used one strategy (MACD) and one instrument (EUR/USD). Therefore, one trading bot was deployed. The details of this bot is stored in the AutoTrader instance we created. We can access the bots deployed by an instance of AutoTrader using the `get_bots_deployed` method, as shown below.

```
bot = at.get_bots_deployed()
```

You will now have access to *bot*, an instance of *AutoBot* which traded the MACD strategy on EUR/USD. Of interest now is the backtest summary of the bot, written to `bot.backtest_summary`. This is a dictionary containing a history of trades taken, orders cancelled, trades still open, and more. Exploring this is left as an exercise to the reader.

Live Trading with AutoTrader

If you have a strategy and have been able to run a backtest on it, you are able to take it live with no extra effort. Live trading is also known as 'forward testing', since you are running the strategy in real-time. You can do this in two different environments:

1. A simulated environment, where trades are simulated in real-time ("paper trading")
2. The live environment, where trades are submitted to real brokers and exchanges for execution with real money.

If you want to do the latter, you will need to make sure you have your API keys defined in your `keys.yaml` *file*. This isn't necessary for paper trading, since the environment is completely simulated.

Live Runfile

To take our MACD strategy live, we can modify the run file to that shown below.

Paper Trade with Virtual Broker

```
from autotrader import AutoTrader

at = AutoTrader()
at.configure(verbosity=1, feed='yahoo',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.virtual_account_config(leverage=30)
at.run()
```

This will launch AutoTrader into live papertrade mode. Every 1 hour, AutoTrader will refresh your strategy to get the latest signals. If an order is recieved, it will be executed in the virtual broker.

When you run AutoTrader in livetrade mode, it will create a new directory named `active_bots`, and store a text file for each actively running AutoTrader instance. To kill an instance, simply delete the instance file, or do a keyboard interrupt. You can open these files to remind yourself which strategy they are running. In the example above, a file named something like "autotrader_instance_1" will be created, and contain the following text.

```
This instance of AutoTrader contains the following bots:
Simple Macd Strategy (EURUSD=X)
```

What if you want to get a bit more accurate with your paper trading?

Paper Trade with Real Broker

Sometime brokers/exchanges offer a paper trading API endpoint, such as Oanda. In this case, you can use their paper-trading API to test your strategies in livetrade mode. What if the broker doesn't offer this? In this case, AutoTrader can mirror the real-time orderbook of the exchange you would like to simulate, executing trades in an instance of AutoTrader's virtual broker.

The runfile below is an example of papertrading on the crypto exchange dYdX. The first difference is that we specify the broker/exchange to trade on in the `configure` method using the `broker` argument. Since we would like to papertrade, we need to configure the virtual account as before. Now, however, you should specify that the account you are configuring is for the 'dydx' exchange, as specified in the `configure` method. This is especially important when trading with multiple brokers at once. When setting up AutoTrader like this, the virtual broker will retrieve the real-time orderbook from 'dydx' in order to simulate trade execution.

```python
from autotrader import AutoTrader

at = AutoTrader()
at.configure(verbosity=1, broker='dydx',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.virtual_account_config(leverage=30, exchange='dydx')
at.run()
```

Live Trade with Real Broker

If you are ready to trade directly on a real exchange, make sure you have defined your API keys in the `keys.yaml` file. Then, set up your run file like the one shown below. It is that easy.

```python
from autotrader import AutoTrader

at = AutoTrader()
at.configure(verbosity=1, broker='dydx',
             mode='continuous', update_interval='1h')
at.add_strategy('macd')
at.run()
```

Note: If the broker you are trading on supports native paper trading (such as Oanda), you can use the same runfile shown above, but pass in 'paper' as the `environment` argument to the `configure` method.

Automated Running

If you are running on a server, you might want to use nohup (short for 'no hangup') to prevent your system from stopping Python when you log out.

3.7 Miscellaneous tutorials

Here you can find a collection of miscellaneous tutorials - things which are useful to know, but not essential to using AutoTrader.

3.7.1 Periodic versus Continuous Mode

AutoTrader has two run modes which control how data is handled and how frequently strategies are instantiated. The active run mode is controlled using `configure` method. A summary of these modes is provided in the table below, but the following sections discuss them in more detail.

	Periodic Mode	Continuous Mode
Strategy instantiation	Every update	Once during deployment
Data indexing	Index based	Time based
Lookahead risk	High	Low
Backtest time	Very fast	Slow

Important: The input arguments to your strategy's `generate_signal` method change slightly depending on the run mode you are using. See the *example* below.

Periodic Update Mode

In periodic update mode, an integer index `i` is used to iterate through the data set to provide trading signals at different points in time. When backtesting, this index will vary from `0` to `len(data)`. Upon each iteration, the method `generate_signal` from the strategy module is called to obtain a signal corresponding to the current timestep. When livetrading or scanning, this index will be `-1`, corresponding to the most recent data, as required for livetrading. This is adequate for most strategies, but carries the risk of accidental data leakage when backtesting, since the strategy is instantiated with the entire dataset.

After the trading bots are updated with the latest data in periodic update mode, they will self-terminate and the AutoTrader instance will become inactive. For this reason, AutoTrader must be run periodically to repeatedly deploy trading bots and act on the latest signal - hence the name 'periodic update mode'. For example, a strategy running on the 4-hour timeframe, AutoTrader should be scheduled to run every 4 hours. Each time it runs, the trading bots will be provided with data of the latest 4-hour candles to run the strategy on. This task is easily automated using cron, or even with a `while True` loop and `time.sleep`. A single bot update in this mode is illustrated in the chart below.

Noting this should bring to attention a point of difference between backtesting and livetrading using periodic update mode: strategies are instantiated once in backtests, but multiple times when livetrading. For some strategies this does not matter, but for others where you would like to maintain the strategies attributes it does. In such cases, continuous update mode may be better suited.

Continuous Update Mode

In continuous update mode, a time marching algorithm is used in place of the integer indexing method used in periodic update mode. That is, time is slowly incremented forwards, and data is slowly revealed to the trading bots. More importantly, there is practically no difference between backtesting and livetrading from the perspective of the trading bots; strategies are instantiated once in both mediums. This means that strategies will maintain attributes from the time it is deployed until the time it is terminated. Data is automatically checked for lookahead bias in this mode, ensuring that the strategy will not see any future data. This comes at the cost of extra processing, meaning that backtesting in this mode is significantly slower.

The charts below illustrate this mode.

Livetrading Bot Management

When bots are deployed for livetrading in continuous mode, a directory named 'active_bots' will be created in the working directory. In this directory, an 'instance file' will be created for each active instance of AutoTrader. The contents of the instance file includes the trading bots deployed in that instance, and the instruments they are trading. This provides a reference as to which AutoTrader instance contains which trading bots. To kill an active instance of AutoTrader, simply delete (or rename) the instance file. This will safely terminate the active bots, and proceed with the *shutdown routines*.

See also:

The name of the instance file can be customised using the `instance_str` argument of the *configure* method.

Example

Instantiation between each mode is the same, but instead of using an integer index `i` to iterate through the data, continous mode provides your strategy with the most up-to-date data at a given timestamp. In theory, if your code has no look-ahead, the same results will be achieved regardless of the run mode used. The code below shows the differences in the `generate_signal` method of the *strategy* class.

```
    def __init__(self, parameters, data, instrument):
        self.data = data

        # 200EMA
        self.ema = TA.EMA(data, parameters['ema_period'])

        # MACD
        self.MACD = TA.MACD(data, parameters['MACD_fast'],
                            parameters['MACD_slow'], parameters['MACD_smoothing'])
        self.MACD_CO = indicators.crossover(self.MACD.MACD, self.MACD.SIGNAL)
        self.MACD_CO_vals = indicators.cross_values(self.MACD.MACD,
                                                    self.MACD.SIGNAL,
                                                    self.MACD_CO)
```

(continues on next page)

(continued from previous page)

```
        # Price swings
        self.swings = indicators.find_swings(data)


    def generate_signal(self, i, **kwargs):
        """Define strategy to determine entry signals.
        """

        if self.data.Close.values[i] > self.ema[i] and \
            self.MACD_CO[i] == 1 and \
            self.MACD_CO_vals[i] < 0:
                new_order = Order(direction=1)

        elif self.data.Close.values[i] < self.ema[i] and \
            self.MACD_CO[i] == -1 and \
            self.MACD_CO_vals[i] > 0:
                new_order = Order(direction=-1)

        else:
            new_order = Order()

        return new_order
```

```
    def calculate_features(self, data):
        self.data = data

        # 200EMA
        self.ema = TA.EMA(data, self.parameters['ema_period'])

        # MACD
        self.MACD = TA.MACD(data, self.parameters['MACD_fast'],
                            self.parameters['MACD_slow'], self.parameters['MACD_smoothing
↪'])
        self.MACD_CO = indicators.crossover(self.MACD.MACD, self.MACD.SIGNAL)
        self.MACD_CO_vals = indicators.cross_values(self.MACD.MACD,
                                                    self.MACD.SIGNAL,
                                                    self.MACD_CO)

        # Price swings
        self.swings = indicators.find_swings(data)


    def generate_signal(self, data):
        """Define strategy to determine entry signals.
        """
        self.calculate_features(data) # Feature calculation on new data

        if self.data.Close.values[-1] > self.ema[-1] and \
            self.MACD_CO[-1] == 1 and \
            self.MACD_CO_vals[-1] < 0:
                new_order = Order(direction=1)
```

(continues on next page)

(continued from previous page)

```
        elif self.data.Close.values[-1] < self.ema[-1] and \
            self.MACD_CO[-1] == -1 and \
            self.MACD_CO_vals[-1] > 0:
                new_order = Order(direction=-1)

        else:
            new_order = Order()

        return new_order
```

Note that a helper function `calculate_features` has been used to calculate the strategies indicators each time new data comes in when using continuous mode. In contrast, this would usually happen in the `__init__` method of a strategy running in periodic mode.

3.7.2 Building a Strategy to Run From a Single File

If you would like to have everything relating to your strategy in a single file, then this page is for you. The example strategy below is for the MACD strategy developed in the *walkthrough*. However, instead of keeping the strategy, strategy configuration and runfile in separate files, we put it all in one.

The strategy class `SimpleMACD` is the same as before, but now we include `if __name__ == "__main__":` at the bottom of the file. This conditional will be `True` whenever we run the entire script. So this is where our new runfile will sit. Here we create an instance of AutoTrader just as before, and set it up for a backtest. The difference is, however, the arguments we provide to the *`add_strategy`* method. Instead of providing the file prefix of the strategy configuration yaml file, we provide the strategy configuration dictionary directly with `config_dict=config`, and provide the strategy class directly with `strategy=SimpleMACD`. Now when we run this module, the strategy will run!

```
import os
from finta import TA
from autotrader import indicators
from autotrader import Order


class SimpleMACD:
    """Simple MACD Strategy

    Rules
    ------
    1. Trade in direction of trend, as per 200EMA.
    2. Entry signal on MACD cross below/above zero line.
    3. Set stop loss at recent price swing.
    4. Target 1.5 take profit.
    """

    def __init__(self, params, data, instrument):
        """Define all indicators used in the strategy.
        """
        self.name = "MACD Trend Strategy"
        self.data = data
        self.params = params
        self.instrument = instrument
```

(continues on next page)

(continued from previous page)

```
        # 200EMA
        self.ema = TA.EMA(data, params['ema_period'])

        # MACD
        self.MACD = TA.MACD(data, self.params['MACD_fast'],
                            self.params['MACD_slow'], self.params['MACD_smoothing'])
        self.MACD_CO = indicators.crossover(self.MACD.MACD, self.MACD.SIGNAL)
        self.MACD_CO_vals = indicators.cross_values(self.MACD.MACD,
                                                    self.MACD.SIGNAL,
                                                    self.MACD_CO)

        # Price swings
        self.swings = indicators.find_swings(data)

        # Construct indicators dict for plotting
        self.indicators = {'MACD (12/26/9)': {'type': 'MACD',
                                              'macd': self.MACD.MACD,
                                              'signal': self.MACD.SIGNAL},
                           'EMA (200)': {'type': 'MA',
                                         'data': self.ema}}


    def generate_signal(self, i, **kwargs):
        """Define strategy to determine entry signals.
        """

        if self.data.Close.values[i] > self.ema[i] and \
            self.MACD_CO[i] == 1 and \
            self.MACD_CO_vals[i] < 0:
                exit_dict = self.generate_exit_levels(signal=1, i=i)
                new_order = Order(direction=1,
                                  stop_loss=exit_dict['stop_loss'],
                                  take_profit=exit_dict['take_profit'])

        elif self.data.Close.values[i] < self.ema[i] and \
            self.MACD_CO[i] == -1 and \
            self.MACD_CO_vals[i] > 0:
                exit_dict = self.generate_exit_levels(signal=-1, i=i)
                new_order = Order(direction=-1,
                                  stop_loss=exit_dict['stop_loss'],
                                  take_profit=exit_dict['take_profit'])

        else:
            new_order = Order()

        return new_order


    def generate_exit_levels(self, signal, i):
        """Function to determine stop loss and take profit levels.
        """
        stop_type = 'limit'
```

(continues on next page)

(continued from previous page)

```
        RR = self.params['RR']

        if signal == 0:
            stop = None
            take = None
        else:
            if signal == 1:
                stop = self.swings.Lows[i]
                take = self.data.Close[i] + RR*(self.data.Close[i] - stop)
            else:
                stop = self.swings.Highs[i]
                take = self.data.Close[i] - RR*(stop - self.data.Close[i])

        exit_dict = {'stop_loss': stop,
                     'stop_type': stop_type,
                     'take_profit': take}

        return exit_dict


if __name__ == "__main__":
    from autotrader import AutoTrader

    config = {'NAME': 'MACD Strategy',
              'MODULE': 'macd_strategy',
              'CLASS': 'SimpleMACD',
              'INTERVAL': 'H4',
              'PERIOD': 300,
              'RISK_PC': 1.5,
              'SIZING': 'risk',
              'PARAMETERS': {'ema_period': 200,
                             'MACD_fast': 5,
                             'MACD_slow': 19,
                             'MACD_smoothing': 9,
                             'RR': 1.5},
              'WATCHLIST': ['EUR_USD'],}

    at = AutoTrader()
    at.configure(verbosity=1, show_plot=False)
    at.add_strategy(config_dict=config, strategy=SimpleMACD)
    at.plot_settings(show_cancelled=False)
    at.add_data({'EUR_USD': 'EUR_USD_H4.csv'},
                data_directory=os.path.join(os.getcwd(), 'data'))
    at.backtest(start = '1/1/2015',
                end = '1/3/2022',
                initial_balance=1000,
                leverage=30,
                spread=0.5,
                commission=0.005)
    at.run()
```

3.7.3 Click Trading with AutoTrader

Although AutoTrader is a platform for automated trading systems, it can also be used to manually place orders over the command line. this can also provide a good means of interacting and becoming familiar with the *Broker Interface*.

Imports

All you need to get started is `AutoTrader` and the `Order` object.

```
from autotrader import AutoTrader, Order
```

Paper Click Trading

As a first example, lets look at papertrading, where we will use the virtual broker to simulate trading on our exchange of choice. As usual, you must configure the virtual trading account with the `virtual_account_config` method. In the example below, we are simulating trading on Binance throught the *CCXT* integration. After creating a new instance of AutoTrader, configure it to use `ccxt:binance` as the broker, then configure the virtual Binance account. Here we are using 10x leverage.

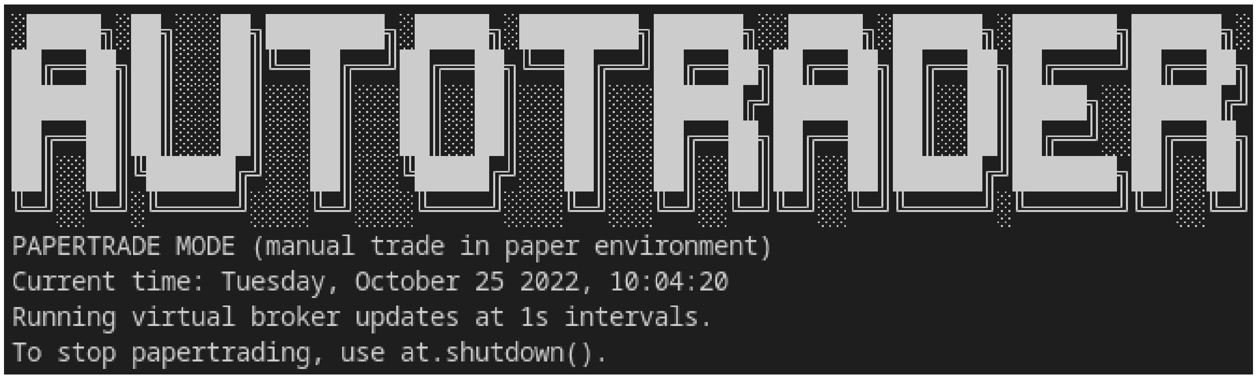
Since we are mirroring the exchange locally, you do not need to provide any API keys to simulate trading.

Start-Up

AutoTrader will go into manual mode whenever you call the `run` method without having added a strategy via the *`add_strategy`* method. Notice that `at.run()` will return the broker instance created.

```
at = AutoTrader()
at.configure(broker='ccxt:binance')
at.virtual_account_config(verbosity=1, exchange='ccxt:binance', leverage=10)
broker = at.run()
```

The following output will be produced. Here we can clearly see that we are manually trading in a paper trading environment. We can now interact directly with the broker instance returned.



```
PAPERTRADE MODE (manual trade in paper environment)
Current time: Tuesday, October 25 2022, 10:04:20
Running virtual broker updates at 1s intervals.
To stop papertrading, use at.shutdown().
```

Tip: While you are at it, why not spin up a *dashboard* to get a nice overview of your trading?

Realtime Updates

You might also notice from the output above that the virtual broker is running, with updates at 1 second intervals. This means that the virtual broker will refresh all prices and orderbook snapshots every second.

To make this all possible, the virtual broker needs to have a constant data feed from the exchange it is mirroring. When launching AutoTrader in manual papertrade mode, a new thread is spawned to do this. This means that the broker instance(s) created will constantly be updating in the background to see if any of your orders should be filled, and to update the value of your positions.

Placing an Order

Now lets look at placing an order. Start by creating the order using an `Order` object, then submit it to the broker using the *`place_order`* method.

```
# Create an order
order = Order(
    instrument="ETH/USDT",
    direction=1,
    size=0.1,
)

# Place order
broker.place_order(order)
```

After doing so, you will see something similar to the following output from the virtual broker.

```
2022-10-25 00:18:39.263975+00:00: Order 1 received: 0.1 unit ETH/USDT market buy order
2022-10-25 00:18:40.625583+00:00: Order 1 filled: 0.1 units of ETH/USDT @ 1342.12 (Buy)
```

A 0.1 unit long position in ETH is now being simulated, using the orderbook on Binance!

Interacting with the Broker

Now that you have a position open, you can interact with the broker and test interacting with the *unified methods*. For example, to get your current positions:

```
>>> broker.get_positions()
{'ETH/USDT': Position in ETH/USDT}
```

To close the position, simply submit another order:

```
# Create a closing order
order = Order(
    instrument="ETH/USDT",
    direction=-1,
    size=0.1,
)

# Place order
broker.place_order(order)
```

Shut-Down

When you are finished click trading, you should use the `shutdown` method to kill the update threads and run the shutdown routines. After doing this, you will be shown a summary of your trades taken during the trading period.

```
at.shutdown()
```

Live Click Trading

Now that we are live trading, we will also need to specify our API keys in a `keys.yaml` file.

Tip: You can use the *command line interface* to initialise the `keys.yaml` file.

Once you have done that, everything is basically the same as described above, with the exception of the `environment="live"` argument. So, to make things a bit more interesting, lets connect to two exchanges at the same time: Bybit and dYdX.

```
at = AutoTrader()
at.configure(verbosity=1, broker='ccxt:bybit, dydx', environment="live")
brokers = at.run()
```

Now, the object returned by `at.run()` is a dictionary with each broker instance:

```
{'ccxt:bybit': AutoTrader-Bybit interface, 'dydx': AutoTrader-dYdX interface}
```

To access the broker instances directly, you can simply index them:

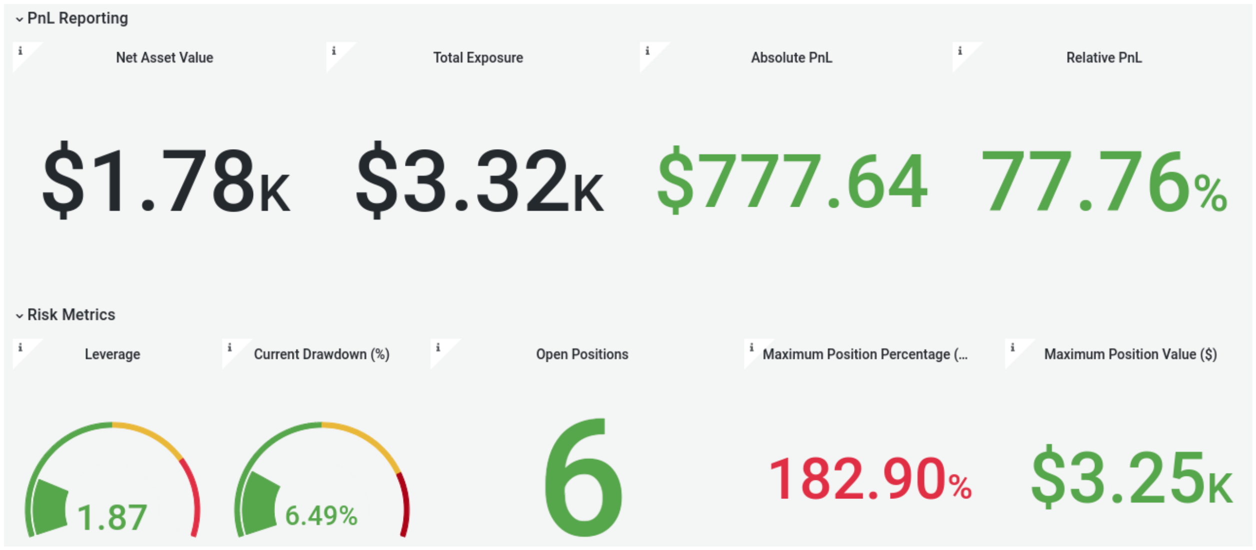
```
bitmex = brokers['ccxt:bitmex']
dydx = brokers['dydx']
```

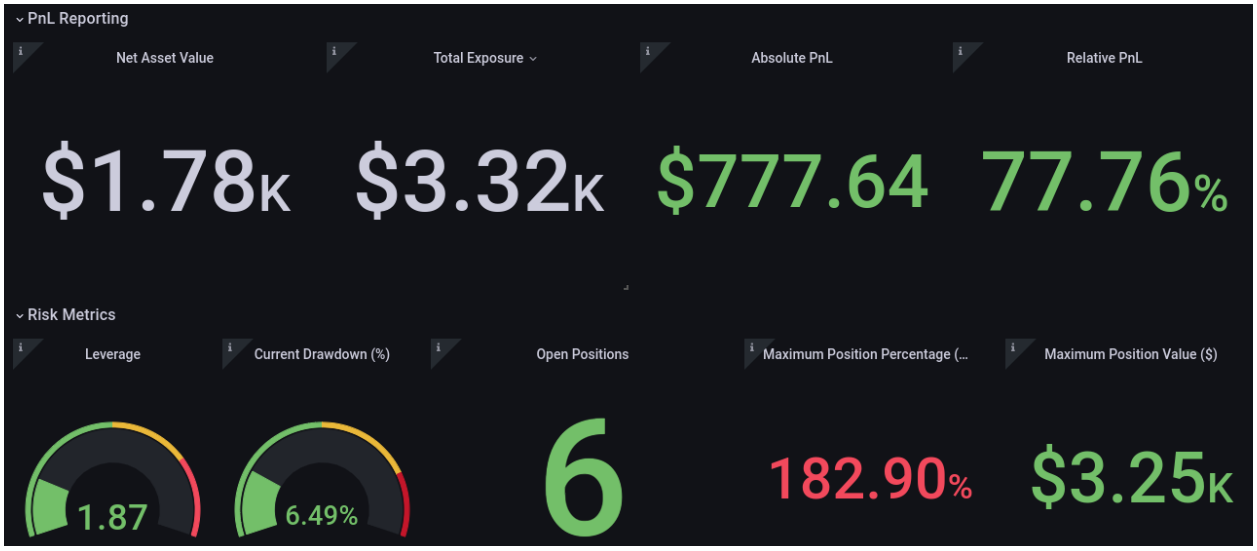
Now, any orders you submit will go directly to the real exchanges.

3.7.4 Trade Dashboard

An effective way of monitoring your trading bots performance while live trading is to use a dashboard. This provides a high level overview of what is happening, allowing you to monitor that everything is going as it should.

AutoTrader makes this an easy task, and even offers a template dashboard to use with Grafana. This template uses the Python client of the data monitoring software Prometheus. The template file can be found in the `templates` directory of the AutoTrader repository, and is previewed below.





Getting Started

In order to set the dashboard up, you must first install Prometheus and Grafana.

Prometheus

Follow the instructions here to install Prometheus. If you are on a Mac, you can run `brew install prometheus`.

Next, configure prometheus by editing `prometheus.yml`. This file is normally in the installation directory. If you installed via `brew`, it will usually be in `/opt/homebrew/etc/prometheus.yml`. Add the following to the `scrape_configs`. Set the port (eg. `8009` in the example below) to one which you plan to use, or leave as is.

```
scrape_configs:
  - job_name: 'autotrader'
    static_configs:
      - targets: ['127.0.0.1:8009']
```

Once you have completed the above, install the Python client:

```
pip install prometheus-client
```

Grafana

Follow the instructions here to install Grafana. You will also need to create an account.

Once complete, open Grafana and add a Prometheus data source:

- Click the settings wheel
- Select "Data Sources"
- Select Prometheus
- Set the URL to `http://localhost:9090/`
- Save.

Finally, select "Import" under Dashboards. Click the "Upload JSON file", and provide the dashboard template file. No data will be shown yet, until we have Prometheus and the Monitor running.

The Monitor

AutoTrader serves trading metrics via Prometheus from the `Monitor` *class*. This class can be instantiated from a Python script, or via the *Command Line Interface*. The following metrics will be exposed.

Metric	Description
`nav_gauge`	The net asset value.
`drawdown_gauge`	The account drawdown.
`max_pos_gauge`	The notional value of the largest position.
`max_pos_frac_gauge`	The fractional size of the largest position.
`abs_pnl_gauge`	The absolute PnL of the account.
`rel_PnL_gauge`	The relative PnL of the account.
`pos_gauge`	The number of positions held.
`total_exposure_gauge`	The total exposure of the account.
`net_exposure_gauge`	The net exposure of the account.
`leverage_gauge`	The active leverage of the account.

Configuration

Add the following monitor configuration yaml file to your `config` *directory*.

```yaml
# monitor.yaml
port: 8009
broker: "ccxt:binanceusdm"
environment: "paper"  # paper or live
initial_nav: 1000  # Reference NAV for PnL calculations
max_nav: 1000  # Maximum NAV for drawdown calculations
sleep_time: 30  # Monitor update time (s)
```

```yaml
# monitor.yaml
port: 8009
picklefile: "broker_picklefile"
environment: "paper"  # paper or live
initial_nav: 1000  # Reference NAV for PnL calculations
max_nav: 1000  # Maximum NAV for drawdown calculations
sleep_time: 30  # Monitor update time (s)
```

Start Prometheus

Before launching the monitor, make sure that Prometheus is running. For Linux machines, use `./prometheus` to run the executable in install directory. For Mac machines, run `brew services start prometheus`.

Launch Monitor from script

The snippet below provides a sample for launching the monitor from a Python script file.

```python
from autotrader.utilities import Monitor

monitor = Monitor(config_filepath="config/monitor.yaml")
monitor.run()
```

Launch Monitor from CLI

Launching the monitor from the command line is as simple as specifying the monitor config filepath to the `monitor` function:

```
autotrader monitor -c config/local_monitor.yaml
```

You will see an output similar to that shown below.

```






Server started on port 8009.
Monitoring with 30 second updates.
Connecting to ccxt:binanceusdm (paper environment)...
  Done.
```

Note that you can also launch the monitor without a config file, by providing some additional information directly. See the *docs for the command line interface* for more information.

Papertrade Monitoring

When papertrading, the broker instances used can be pickled, to save their state at that time. This allows us to glance into the simulated trading evironment and query the broker.

3.8 User Inputs

To use AutoTrader, you must provide three things:

1. A *trading strategy*,
2. The *strategy configuration*,
3. The *account configuration* (trading account keys).

The first of these, the trading strategy, is written as a Python class, and contains the logic of your strategy - given some data, what trading signals can be extracted?

The strategy configuration tells AutoTrader the required information about how your strategy should be run. This included things like the minimum number of data points required to run your strategy, and which assets to trade with. It also contains strategy parameters which get passed to your strategy, which can be tuned to improve the performance of your strategy.

Finally, the account configuration is required to connect to your broker and place orders. This is only required when live trading.

Each of these items must be formatted and structured according to a few rules, so read on to learn the details.

3.8.1 Trading Strategy

`strategies/*.py`

Trading strategies are built as class objects, and must follow a few simple rules to function properly with AutoTrader. At a minimum, a strategy is required to have two methods. The first is the `__init__` method, which is used to instantiate the strategy. The second is the `generate_signal` method, which contains the strategy logic and ouputs any trading signals which may present themselves. Read more about these methods below and take a look at the sample strategies in the demo repository. It might also be helpful to review the tutorial, Building a Strategy.

Strategy Overview

Tip: A strategy template can be found in the templates folder of the Github repository. You can also use the command line interface to download strategies from the repo.

The code block below provides some boilerplate for a strategy in AutoTrader. The methods shown are the only things **required** of a strategy. Note the differences in the arguments to the `generate_signal` method between periodic and continuous update mode. Read about these modes in the *AutoTrader docs*.

```python
from autotrader import Order

class Strategy:
    def __init__(self, parameters, data, instrument, **kwargs):
        """Define all attributes of the strategy.
        """
```

(continues on next page)

(continued from previous page)

```
        self.name = "Template Strategy"
        self.data = data
        self.params = params
        self.instrument = instrument

        # Define any indicators used in the strategy
        ...

        # Construct indicators dict for plotting
        self.indicators = {'Indicator Name': {'type': 'indicatortype',
                                              'data': 'indicatordata'},}

    def generate_signal(self, data):
        """Define strategy logic to determine entry signals.

        Parameters
        ----------
        data : pd.DataFrame|dict
            The strategy data.
        """
        # Example long market order
        order = Order(direction=1)
        return order
```

```
from autotrader import Order

class Strategy:
    def __init__(self, parameters, data, instrument, **kwargs):
        """Define all attributes of the strategy.
        """
        self.name = "Template Strategy"
        self.data = data
        self.params = params
        self.instrument = instrument

        # Define any indicators used in the strategy
        ...

        # Construct indicators dict for plotting
        self.indicators = {'Indicator Name': {'type': 'indicatortype',
                                              'data': 'indicatordata'},}

    def generate_signal(self, i):
        """Define strategy logic to determine entry signals.

        Parameters
        ----------
        i : int
            An integer index used to specify the location in the
            dataset.
        """
        # Example long market order
```

(continues on next page)

(continued from previous page)

```
        order = Order(direction=1)
        return order
```

Initialisation

The `__init__` method always initialises a strategy with the following named arguments:

1. `parameters`: a dictionary containing the strategy parameters from your strategy configuration file.
2. `data`: the strategy data, which may be a DataFrame, or a dictionary of different datasets.
3. `instrument`: a string with the trading instrument's name (as it appears in the *watchlist*), or a list of instruments if the strategy has `PORTFOLIO: True` in its configuration.

Broker Access

In some cases, you may like to directly connect with the broker from your strategy module. In this case, you must include `INCLUDE_BROKER: True` in your *strategy configuration*. This will tell AutoTrader to instantiate your strategy with the broker API and broker utilities. You will therefore need to include these as named arguments to your `__init__` method, as shown below. Now you can access the methods of the *broker* directly from your strategy!

```
def __init__(self, parameters, data, instrument, broker, broker_utils, **kwargs):
    """Define all attributes of the strategy.
    """
    self.data = data
    self.parameters = parameters
    self.instrument = instrument
    self.broker = broker
    self.utils = broker_utils
```

Data Stream Access

It may also be of interest to include the *data stream* object when your strategy is instantiated, particularly if you are using a custom data stream. As above, this can be achieved by specifying `INCLUDE_STREAM: True` in your *strategy configuration*.

```
def __init__(self, parameters, data, instrument, data_stream, **kwargs):
    """Define all attributes of the strategy.
    """
    self.data = data
    self.parameters = parameters
    self.instrument = instrument
    self.data_stream = data_stream
```

Indicators Dictionary

If you wish to include any indicators your strategy uses when visualising backtest results, you must define an `indicators` attribute. This attribute takes the form of a dictionary with the indicators which you would like to include. This dictionary then gets passed to *AutoPlot*. The general form of this dictionary is shown below, but read more in the *docs* for more information.

```
self.indicators = {'indicator 1 name': {'type': 'indicator 1 type',
                                        'data': self.indicator1_data},
                   'indicator 2 name': {'type': 'indicator 2 type',
                                        'data': self.indicator2_data},
                    }
```

Signal Generation

Signals are generated using the `generate_signal` method. This method contains the logic behind your strategy and returns trading signals in the form of *`Orders`*. This method must always return something, so either:

1. Return an `Order`,
2. Return multiple `Orders` grouped in a `list`,
3. Return a blank `Order` using `Order()`, or an empty list `[]`.

The details you provide in each order will depend on the *order type*. The most basic (and default) order is a `market` order, which only requires you to specify the `direction`: 1 for a long trade, and -1 for a short trade. Note that you do not have to provide the `instrument` to the order; AutoTrader will do that for you based on the instruments defined in the `WATCHLIST` of the strategy's configuration. In some cases, however, you may want to provide it directly (when trading multiple instruments from the same strategy, for example).

The input arguments to this method will depend on the *run mode* of AutoTrader. In *continuous update mode* (the default mode), the most-recent strategy data is passed in. When backtesting, this dataset will evolve over the backtest period. When livetrading, the latest data available will be passed in. This mode is less prone to lookahead. In *periodic update mode*, the data indexing parameter `i` is passed in as the first argument. Since the entire dataset is passed to the strategy in this mode, the indexing parameter `i` must be used to act on the correct datapoints.

Shutdown Routine

If you have a process you would like to exectue *after* your strategy has finished running, you may use the shutdown routine functionality to do so. This involves creating a method involving your shutdown routine and specifying it to AutoTrader via the *`add_strategy`* method.

To explain this functionality, consider you are livetrading with a strategy which maintains many open trades at once. If you have this bot deployed in *continuous update mode* and would like to terminate it by deleting its *instance file*, you likely would like it to safely close all open trades before terminating. Else, you may have unmanaged trades left open on your account. To prevent this, you may create a shutdown routine as shown below, which cancels any pending orders and closes all remaining open positions. Of course, this method should be more strategic than simply closing all positions and cancelling any orders.

```
def safe_exit_strategy(self):
    # Cancel all pending orders
    pending_orders = self.broker.get_orders(self.instrument, 'pending')
    for order_id in pending_orders:
        self.broker.cancel_order(order_id)
```

(continues on next page)

(continued from previous page)

```
    # Close all open trades
    close_order = Order(instrument=self.instrument, order_type='close')
    self.broker.place_order(close_order)
```

If you provide the name of your shutdown routine - in the example above this is 'safe_exit_strategy' - to AutoTrader via the `shutdown_method` argument of the *`add_strategy`* method, it will be called when the bot is terminated.

3.8.2 Strategy Configuration

`config/*.yaml`

The strategy configuration contains information related specifically to a strategy. Each *strategy* requires its own configuration to be able to run. It is written either as a .yaml file, or passed directly to AutoTrader as a dictionary via the *`add_strategy`* method (`.yaml` files are read into Python as dictionaries). Note that spacing is important in `.yaml` files, with each level defined by two subsequent spaces.

Tip: A template strategy configuration file can be generated using the command line interface! Simply run `autotrader init -s config` in your home directory, and a template file will be created in the `config/` directory. You can also find this template in the Github repository.

Configuration Options

The keys of the strategy configuration file are described in the table below. Note that all `PARAMETERS` defined will be accessible in the *strategy*, via the `parameters` argument.

Key	Description	Required	Default value
`NAME`	A string defining the strategy name.	Yes	
`MODUL`	A string containing the prefix name of the strategy module, without the *.py* suffix.	Yes, unless the strategy is passed directly via `add_strategy`	
`CLASS`	A string containing the class name within the strategy module.	Yes	
`INTER`	The granularity of the data used by the strategy.	Yes	
`PERIO`	The number of candles to fetch when live trading (eg. a value of 300 will fetch the latest 300 candles), or a timedelta string (eg. ‘30d’).	Yes	
`PARAM`	A dictionary containing custom strategy parameters (see below).	Yes	
`WATCH`	A list containing the instruments to be traded in the strategy, in the *format required* by the *data feed*.	Yes	
`SIZIN`	The method to use when calculating position size. Can be either ‘risk’ or an integer value corresponding to the number of units to trade. If using the ‘risk’ option, position size will be calculated based on trading account balance and the value assigned to `RISK_PC`.	No	None
`RISK_`	The percentage of the account balance to risk when determining position risk-based size.	No	None
`PORTF`	A boolean flag for if the strategy is a portfolio-based strategy, requiring fata for all instrumenets in the watchlists to run.	No	False
`INCLU`	A boolean flag to indicate if the broker interface and broker utilities should be passed to the strategy’s `__init__` method. Read more *here*.	No	False
`INCLU`	A boolean flag to indicate if the *data stream* object should be passed to the strategy’s `__init__` method.	No	False

Data Interval

The `INTERVAL` key is a string used to define the granularity of the data used by your strategy. For example, ‘1m’ for minutely data, or ‘4h’ for 4-hourly data. This is used to infer the timestep to take when backtesting (and livetrading), but also to convert the `PERIOD` to an integer value if necessary.

If you would like to build a strategy which uses multiple timeframes, simply specify the timeframes with comma separation in the `INTERVAL` key. For example, to have access to 15-minute and 4-hour data, you would specify something like `INTERVAL: 'M15,H1'`. In this case, the `data` object passed into the strategy will be a dictionary, with keys defined by each granularity specified in `INTERVAL` and the associated data.

Data Period

The PERIOD key is used to determine how many rows of data (OHLC) is required by your strategy. This could refer to the lookback period, either as an integer value of the number of rows, or as a string such as '30d', indicating 30 days worth of data is required. If an integer value is provided, for example 300, then the latest 300 rows of data will be passed to your strategy.

Strategy Parameters

The parameters key contains any information you would like to be able to access from your *strategy* module. This might include things like indicator configuration parameters, such as periods, and exit parameters, such as a risk-to-reward ratio.

Example Configuration

An example strategy configuration is provided below. Each file will look very similar to this, with the exception of the parameters key, which will be tailored to your own strategy. Feel free to look at the configuration files for the example strategies provided in the AutoTrader demo repository.

```
NAME: 'Strategy Name'
MODULE: 'modulename'
CLASS: 'StrategyClassName'
INTERVAL: 'M15'
PERIOD: 300
RISK_PC: 1      # 1%
SIZING: 'risk'
PARAMETERS:
  ema_period: 200
  rsi_period: 14

  # Exit level parameters
  RR: 2
  stop_buffer: 10 # pips

# Define instruments to monitor
WATCHLIST: ['EUR_USD']
```

```
strategy_config = {'NAME': 'Strategy Name',
                   'MODULE': 'modulename',
                   'CLASS': 'StrategyClassName',
                   'INTERVAL': 'M15',
                   'PERIOD': 300,
                   'RISK_PC': 1,
                   'SIZING': 'risk',
                   'PARAMETERS': {'ema_period': 200,
                                  'rsi_period': 14,
                                  'RR': 2,
                                  'stop_buffer': 10},
                    'WATCHLIST': ['EUR_USD',]}
```

3.8.3 Account Configuration

`config/keys.yaml`

In order to connect with your broker/exchange of choice, you must provide the API keys. Keys can be kept in a `keys.yaml` file in your `config` directory, or passed directly to AutoTrader as a dictionary. To do the latter, use the *configure* method of AutoTrader. If you are keeping your keys in the `keys.yaml` file, AutoTrader will automatically find them.

Example

Example account configuration structures are shown below. For exchange-specific configuration keys, refer to the appropriate *docs*.

Tip: A template `keys.yaml` file can be generated using the command line interface! Simply run `autotrader init` in your home directory, and the template file will be created in the `config/` directory. You can also find this template in the Github repository.

```yaml
OANDA:
  LIVE_API: "api-fxtrade.oanda.com"
  LIVE_ACCESS_TOKEN: "12345678900987654321-abc34135acde13f13530"
  PRACTICE_API: "api-fxpractice.oanda.com"
  PRACTICE_ACCESS_TOKEN: "12345678900987654321-abc34135acde13f13530"
  DEFAULT_ACCOUNT_ID: "xxx-xxx-xxxxxxxx-001"
  PORT: 443

CCXT:EXCHANGE:
  api_key: "xxxx"
  secret: "xxxx"
  base_currency: "USDT"
```

```python
keys_config = {
    "OANDA": {
        "LIVE_API": "api-fxtrade.oanda.com",
        "LIVE_ACCESS_TOKEN": "12345678900987654321-abc34135acde13f13530",
        "PRACTICE_API": "api-fxpractice.oanda.com",
        "PRACTICE_ACCESS_TOKEN": "12345678900987654321-abc34135acde13f13530",
        "DEFAULT_ACCOUNT_ID": "xxx-xxx-xxxxxxxx-001",
        "PORT": 443,
    },
    "CCXT:EXCHANGE": {"api_key": "xxxx", "secret": "xxxx", "base_currency": "USDT"},
}
```

To pass the keys dictionary to AutoTrader, use `at.configure(global_config=keys_config)`.

3.9 AutoTrader API

3.9.1 AutoTrader

```
from autotrader import AutoTrader
```

The `AutoTrader` class is the orchestrator of your trading system. It will sort out all the boring stuff so that you can spend more time where it matters - researching good trading strategies.

Configuration Methods

The following methods are used to configure the active instance of AutoTrader.

Run Configuration

To configure the settings of AutoTrader, use the `configure` method. Here you can specifiy the verbosity of AutoTrader, set your data feed, set which broker to trade with and more. If you are going to provide local data, you should always call the `configure` method prior to specifying the local data via the `add_data` *method*, to ensure that the working directory is correctly configured. In general, it is always good to call `configure` first up.

`AutoTrader.`**`configure`**(*verbosity: int = 1*, *broker: str | None = None*, *execution_method: Callable | None = None*, *feed: str | None = None*, *req_liveprice: bool = False*, *notify: int = 0*, *notification_provider: str | None = None*, *home_dir: str | None = None*, *allow_dancing_bears: bool = False*, *account_id: str | None = None*, *environment: str = 'paper'*, *show_plot: bool = False*, *jupyter_notebook: bool = False*, *mode: str = 'continuous'*, *update_interval: str | None = None*, *data_index_time: str = 'open'*, *global_config: dict | None = None*, *instance_str: str | None = None*, *broker_verbosity: int = 0*, *home_currency: str | None = None*, *allow_duplicate_bars: bool = False*, *deploy_time: datetime | None = None*, *max_workers: int | None = None*) → None

Configures run settings for AutoTrader.

Parameters

- **verbosity** (*int, optional*) – The verbosity of AutoTrader (0, 1, 2). The default is 1.
- **broker** (*str, optional*) – The broker(s) to connect to for trade execution. Multiple exchanges can be provided using comma separattion. The default is 'virtual'.
- **execution_method** (*Callable, optional*) – The execution model to call when submitting orders to the broker. This method must accept the broker instance, the order object, order_time and any *args, **kwargs.
- **feed** (*str, optional*) – The data feed to be used. This can be the same as the broker being used, or another data source. Options include 'yahoo', 'oanda', 'ib', 'dydx', 'ccxt', 'local' or 'none'. When data is provided via the add_data method, the feed is automatically set to 'local'. The default is None.
- **req_liveprice** (*bool, optional*) – Request live market price from broker before placing trade, rather than using the data already provided. The default is False.
- **notify** (*int, optional*) – The level of notifications (0, 1, 2). The default is 0.
- **notification_provider** (*str, optional*) – The notifications provider to use (currently only Telegram supported). The default is None.
- **home_dir** (*str, optional*) – The project home directory. The default is the current working directory.

- **allow_dancing_bears** (*bool, optional*) – Allow incomplete candles to be passed to the strategy. The default is False.
- **account_id** (*str, optional*) – The brokerage account ID to be used. The default is None.
- **environment** (*str, optional*) – The trading environment of this instance ('paper', 'live'). The default is 'paper'.
- **show_plot** (*bool, optional*) – Automatically generate trade chart. The default is False.
- **jupyter_notebook** (*bool, optional*) – Set to True when running in Jupyter notebook environment. The default is False.
- **mode** (*str, optional*) – The run mode (either 'periodic' or 'continuous'). The default is 'periodic'.
- **update_interval** (*str, optional*) – The update interval to use when running in 'continuous' mode. This should align with the highest resolution bar granularity in your strategy to allow adequate updates. The string inputted will be converted to a timedelta object. If None is passed, the update interval will be inferred from the strategy INTERVAL. The default is None.
- **data_index_time** (*str, optional*) – The time by which the data is indexed. Either 'open', if the data is indexed by the bar open time, or 'close', if the data is indexed by the bar close time. The default is 'open'.
- **global_config** (*dict, optional*) – Optionally provide your global configuration directly as a dictionary, rather than it being read in from a yaml file. The default is None.
- **instance_str** (*str, optional*) – The name of the active AutoTrader instance, used to control bots deployed when livetrading in continuous mode. When not specified, the instance string will be of the form 'autotrader_instance_n'. The default is None.
- **broker_verbosity** (*int, optional*) – The verbosity of the broker. The default is 0.
- **home_currency** (*str, optional*) – The home currency of trading accounts used (intended for FX conversions). The default is None.
- **allow_duplicate_bars** (*bool, optional*) – Allow duplicate bars to be passed on to the strategy. The default is False.
- **deploy_time** (*datetime, optional*) – The time to deploy the bots. If this is a future time, AutoTrader will wait until it is reached before deploying. It will also be used as an anchor to synchronise future bot updates. If not specified, bots will be deployed as soon as possible, with successive updates synchronised to the deployment time.
- **max_workers** (*int, optional*) – The maximum number of workers to use when spawning threads. The default is None.

Returns
Calling this method configures the internal settings of the active AutoTrader instance.

Return type
None

Backtest Configuration

To configure a backtest, use the `backtest` method.

`AutoTrader.`**`backtest`**`(`*start: str | None = None*, *end: str | None = None*, *start_dt: datetime | None = None*, *end_dt: datetime | None = None*, *warmup_period: str = '0s'*`)` → None

Configures settings for backtesting.

Parameters

- **`start`** (*`str, optional`*) – Start date for backtesting, in format dd/mm/yyyy. The default is None.
- **`end`** (*`str, optional`*) – End date for backtesting, in format dd/mm/yyyy. The default is None.
- **`start_dt`** (*`datetime, optional`*) – Datetime object corresponding to start time. The default is None.
- **`end_dt`** (*`datetime, optional`*) – Datetime object corresponding to end time. The default is None.
- **`warmup_period`** (*`str, optional`*) – A string describing the warmup period to be used. This is equivalent to the minimum period of time required to collect sufficient data for the strategy. The default is '0s'.

Notes

Start and end times must be specified as the same type. For example, both start and end arguments must be provided together, or alternatively, start_dt and end_dt must both be provided.

Virtual Account Configuration

Any time you are simulating trading, either in backtest or papertrading, you should configure the simulated trading account. This is done using the `virtual_account_config` method. Here, you can set your initial balance, the bid/ask spread, the trading fees and more. If you do not call this method when simulating trading, the default settings will be used.

If you plan on simultaneously trading across multiple venues, you will need to configure a virtual account for each exchange being simulated. When doing so, use the `exchange` argument to specify which account it is that you are configuring. This should align with the brokers/exchanges specified to the `broker` argument in *`AutoTrader.configure`*.

`AutoTrader.`**`virtual_account_config`**`(`*verbosity: int = 0*, *initial_balance: float = 1000*, *spread: float = 0*, *commission: float = 0*, *spread_units: str = 'price'*, *commission_scheme: str = 'percentage'*, *maker_commission: float | None = None*, *taker_commission: float | None = None*, *leverage: int = 1*, *hedging: bool = False*, *margin_call_fraction: float = 0*, *default_slippage_model: Callable | None = None*, *slippage_models: dict | None = None*, *picklefile: str | None = None*, *exchange: str | None = None*, *tradeable_instruments: list | None = None*, *refresh_freq: str = '1s'*, *home_currency: str | None = None*, *papertrade: bool = True*`)` → None

Configures the virtual broker's initial state to allow livetrading on the virtual broker. If you wish to create multiple virtual broker instances, call this method for each virtual account.

Parameters

- **verbosity** (*int, optional*) – The verbosity of the broker. The default is 0.
- **initial_balance** (*float, optional*) – The initial balance of the account. The default is 1000.
- **spread** (*float, optional*) – The bid/ask spread to use in backtest (specified in units defined by the spread_units argument). The default is 0.
- **spread_units** (*str, optional*) – The unit of the spread specified. Options are 'price', meaning that the spread is quoted in price units, or 'percentage', meaning that the spread is quoted as a percentage of the market price. The default is 'price'.
- **commission** (*float, optional*) – Trading commission as percentage per trade. The default is 0.
- **commission_scheme** (*str, optional*) – The method with which to apply commissions to trades made. The options are (1) 'percentage', where the percentage specified by the commission argument is applied to the notional trade value, (2) 'fixed_per_unit', where the monetary value specified by the commission argument is multiplied by the number of units in the trade, and (3) 'flat', where a flat monetary value specified by the commission argument is charged per trade made, regardless of size. The default is 'percentage'.
- **maker_commission** (*float, optional*) – The commission to charge on liquidity-making orders. The default is None, in which case the nominal commission argument will be used.
- **taker_commission** (*float, optional*) – The commission to charge on liquidity-taking orders. The default is None, in which case the nominal commission argument will be used.
- **leverage** (*int, optional*) – Account leverage. The default is 1.
- **hedging** (*bool, optional*) – Allow hedging in the virtual broker (opening simultaneous trades in oposing directions). The default is False.
- **margin_call_fraction** (*float, optional*) – The fraction of margin usage at which a margin call will occur. The default is 0.
- **default_slippage_model** (*Callable, optional*) – The default model to use when calculating the percentage slippage on the fill price, for a given order size. The default functon returns zero.
- **slippage_models** (*dict, optional*) – A dictionary of callable slippage models, keyed by instrument.
- **picklefile** (*str, optional*) – The filename of the picklefile to load state from. If you do not wish to load from state, leave this as None. The default is None.
- **exchange** (*str, optional*) – The name of the exchange to use for execution. This gets passed to an instance of AutoData to update prices and use the realtime orderbook for virtual order execution. The default is None.
- **tradeable_instruments** (*list, optional*) – A list containing strings of the instruments tradeable through the exchange specified. This is used to determine which exchange orders should be submitted to when trading across multiple exchanges. This should account for all instruments provided in the watchlist. The default is None.
- **refresh_freq** (*str, optional*) – The timeperiod to sleep for in between updates of the virtual broker data feed when manually papertrading. The default is '1s'.
- **home_currency** (*str, optional*) – The home currency of the account. The default is None.
- **papertrade** (*bool, optional*) – A boolean to flag when the account is to be used for papertrading (real-time trading on paper). The default is True.

Configure AutoPlot Settings

To configure the settings used by *AutoPlot* when creating charts, use the `plot_settings` method.

`AutoTrader.`**`plot_settings`**(*max_indis_over: int = 3*, *max_indis_below: int = 2*, *fig_tools: str = 'pan,wheel_zoom,box_zoom,undo,redo,reset,save,crosshair'*, *ohlc_height: int = 400*, *ohlc_width: int = 800*, *top_fig_height: int = 150*, *bottom_fig_height: int = 150*, *jupyter_notebook: bool = False*, *show_cancelled: bool = True*, *chart_timeframe: str = 'default'*, *chart_theme: str = 'caliber'*, *use_strat_plot_data: bool = False*, *portfolio_chart: bool = False*) → None

Configure the plot settings.

Parameters

- **`max_indis_over`** (*`int, optional`*) – Maximum number of indicators overlaid on the main chart. The default is 3.
- **`max_indis_below`** (*`int, optional`*) – Maximum number of indicators below the main chart. The default is 2.
- **`fig_tools`** (*`str, optional`*) – The figure tools. The default is "pan,wheel_zoom,box_zoom,undo, redo,reset,save,crosshair".
- **`ohlc_height`** (*`int, optional`*) – The height (px) of the main chart. The default is 400.
- **`ohlc_width`** (*`int, optional`*) – The width (px) of the main chart. The default is 800.
- **`top_fig_height`** (*`int, optional`*) – The height (px) of the figure above the main chart. The default is 150.
- **`bottom_fig_height`** (*`int, optional`*) – The height (px) of the figure(s) below the main chart. The default is 150.
- **`jupyter_notebook`** (*`bool, optional`*) – Boolean flag when running in Jupyter Notebooks, to allow inline plotting. The default is False.
- **`show_cancelled`** (*`bool, optional`*) – Show/hide cancelled trades. The default is True.
- **`chart_timeframe`** (*`str, optional`*) – The bar timeframe to use when gerating the chart. The timeframe provided must be a part of the strategy dataset. The default is 'default'.
- **`chart_theme`** (*`bool, optional`*) – The theme of the Bokeh chart generated. The default is "caliber".
- **`use_strat_plot_data`** (*`bool, optional`*) – Boolean flag to use data from the strategy instead of candlestick data for the chart. If True, ensure your strategy has a timeseries data attribute named 'plot_data'. The default is False.
- **`portfolio_chart`** (*`bool, optional`*) – Override the default plot settings to plot the portfolio chart even when running a single instrument backtest.

Returns

The plot settings will be saved to the active AutoTrader instance.

Return type

None

Other Methods

Below are the methods used to add your own components to AutoTrader.

Add New Strategy

Trading strategies can be added using the `add_strategy` method of AutoTrader. If you would like to add multiple strategies to the same instance of AutoTrader, simply call this method for each strategy being added. Note that this method accepts both `strategy_filename` and `strategy_dict` arguments. The first of these is used to provide the prefix of a *strategy configuration* file, while the second can be used to directly pass in a strategy configuration dictionary.

`AutoTrader.`**`add_strategy`**(*config_filename: str | None = None*, *config_dict: dict | None = None*, *strategy=None*, *shutdown_method: str | None = None*) → None

Adds a strategy to AutoTrader.

Parameters

- **`config_filename`** (*`str, optional`*) – The prefix of the yaml strategy configuration file, located in home_dir/config. The default is None.
- **`config_dict`** (*`dict, optional`*) – Alternative to config_filename, a strategy configuration dictionary can be passed directly. The default is None.
- **`strategy`** (*`AutoTrader Strategy, optional`*) – The strategy class object. The default is None.
- **`shutdown_method`** (*`str, optional`*) – The name of the shutdown method in the strategy (if any). This method will be called when AutoTrader is livetrading in continuous mode, and the instance has recieved the shutdown signal. The default is None.

Returns
The strategy will be added to the active AutoTrader instance.

Return type
None

Add Data

To trade using a local data source, use the `add_data` method to tell AutoTrader where to look for the data. You can use this method for both backtesting and livetrading. Of course, if you are livetrading, you will need to make sure that the locally stored data is being updated at an appropriate interval.

Note that you do not have to call this method if you are directly connecting to one of the supported exchanges for a data *feed*. In this case, AutoTrader will automatically download data using the information provided in your *strategy configuration* and supply it to your strategy.

Important: The `configure` *method* should be called before calling `add_data`, as it will set the `home_dir` of your project.

`AutoTrader.`**`add_data`**(*data_dict: dict | None = None*, *mapper_func: callable | None = None*, *quote_data: dict | None = None*, *data_directory: str = 'price_data'*, *abs_dir_path: str | None = None*, *auxdata: dict | None = None*, *stream_object=None*, *dynamic_data: bool = False*) → None

Specify local data to run AutoTrader on.

Parameters

- **data_dict** (*dict, optional*) – A dictionary containing the filenames of the datasets to be used. The default is None.
- **mapper_func** (*callable, optional*) – A callable used to provide the absolute filepath to the data given the instrument name (as it appears in the watchlist) as an input argument. The default is None.
- **quote_data** (*dict, optional*) – A dictionary containing the quote data filenames of the datasets provided in data_dict. The default is None.
- **data_directory** (*str, optional*) – The name of the sub-directory containing price data files. This directory should be located in the project home directory (at.home_dir). The default is 'price_data'.
- **abs_dir_path** (*str, optional*) – The absolute path to the data_directory. This parameter may be used when the datafiles are stored outside of the project directory. The default is None.
- **auxdata** (*dict, optional*) – A dictionary containing the data paths to supplement the data passed to the strategy module. For strategies involving multiple products, the keys of this dictionary must correspond to the products, with the auxdata in nested dictionaries or otherwise. The default is None.
- **stream_object** (*DataStream, optional*) – A custom data stream object, allowing custom data pipelines. The default is DataStream (from autotrader.utilities).
- **dynamic_data** (*bool, optional*) – A boolean flag to signal that the stream object provided should be refreshed each timestep of a backtest. This can be useful when backtesting strategies with futures contracts, which expire and must be rolled. The default is False.

Raises

Exception – When multiple quote-data files are provided per instrument traded.

Returns

Data will be assigned to the active AutoTrader instance for later use.

Return type

None

Notes

To ensure proper directory configuration, this method should only be called after calling autotrader.configure().

The data provided to the strategy will either contain a single timeframe OHLC dataframe, a dictionary of MTF dataframes, or a dict with 'base' and 'aux' keys, for aux and base strategy data (which could be single of MTF).

Examples

An example data_dict is shown below.

```
>>> data_dict = {'product1': 'filename1.csv',
                 'product2': 'filename2.csv'}
```

For MTF data, data_dict should take the form shown below. In the case of MTF data, quote data should only be provided for the base timeframe (ie. the data which will be iterated on when backtesting). Therefore, the quote_data dict will look the same for single timeframe and MTF backtests.

```
>>> data_dict = {'product1': {'H1': 'product1_H1.csv',
                             'D': 'product1_D.csv'},
                 'product2': {'H1': 'product2_H1.csv',
                             'D': 'product2_D.csv'}
                 }
```

An example for the quate_data dictionary is shown below.

```
>>> quote_data = {'product1': 'product1_quote.csv',
                  'product2': 'product2_quote.csv'}
```

The auxdata dictionary can take the form shown below. This data will be passed on to your strategy.

```
>>> auxdata = {'product1': 'aux_price_data.csv',
               'product2': {'extra_data1': 'dataset1.csv',
                            'extra_data2': 'dataset2.csv'}
               }
```

Get Bots Deployed

After running AutoTrader, it may be of interest to access the *trading bots* that were deployed. To do so, use the `get_bots_deployed` method.

`AutoTrader.`**`get_bots_deployed`**`(`*instrument: str | None = None*`)` → dict

Returns a dictionary of AutoTrader trading bots, organised by instrument traded.

Parameters
: **`instrument`** (*`str, optional`*) – The instrument of the bot to retrieve. The default is None.

Returns
: A dictionary of deployed AutoTrader bot instances.

Return type
: dict

Notes

If there is only one trading bot deployed, this will be returned directly, rather than in a dict.

Plot Backtest

After running a backtest, you can call `plot_backtest` to create a chart of the trade results. You can also pass in a specific *trading bot* to view the trades taken by that specific bot.

`AutoTrader.`**`plot_backtest`**`(`*bot=None*`)` → None

Plots trade results of an AutoTrader Bot.

Parameters
: **`bot`** (*`AutoTrader bot instance, optional`*) – AutoTrader bot class containing trade results. The default is None.

Returns
: A chart will be generated and shown.

Return type
None

Print Trade Results

`AutoTrader.`**`print_trade_results`**`(`*trade_results:* TradeAnalysis | *None* = *None*`)` → None

Prints trade results.

Parameters
`trade_results` (`TradeAnalysis`) – The trade analysis results class object.

Returns
Trade results will be printed.

Return type
None

Run AutoTrader

After *configuring* AutoTrader, *adding a strategy* and specifying how you would like to trade, you are ready to run AutoTrader. To do so, simply call the `run` method.

`AutoTrader.`**`run`**`()` → *AbstractBroker*

Performs essential checks and runs AutoTrader.

Example Runfiles

Shown below are example scripts for running AutoTrader.

See also:

More examples can be found in the demo repository.

```
from autotrader import AutoTrader

at = AutoTrader()
at.configure(feed='oanda', broker='oanda', verbosity=1)
at.add_strategy('macd_crossover')
at.backtest(start='1/1/2015', end='1/3/2022')
at.virtual_account_config(leverage=30, exchange='oanda')
at.run()
```

```
from autotrader import AutoTrader

at = AutoTrader()
at.configure(feed='oanda', broker='oanda', verbosity=1)
at.add_strategy('macd_crossover')
at.virtual_account_config(leverage=30, exchange='oanda')
at.run()
```

```
from autotrader import AutoTrader

at = AutoTrader()
```

(continues on next page)

(continued from previous page)

```
at.configure(feed='oanda', broker='oanda', verbosity=1)
at.add_strategy('macd_crossover')
at.run()
```

3.9.2 AutoData

`from autotrader import AutoData`

AutoData is the unified data retrieval module of AutoTrader. It enables fetching historical and live price data from the feeds of the *supported brokers* (where available). You can also use `AutoData` to load locally stored data. To configure AutoData, you must provide a `data_config` dictionary, specifying at least the data source. If you do not provide this dictionary, AutoData will assume you will be using local data.

AutoData has three main public methods:

1. `fetch`: to fetch historical OHLC data
2. `L1`: to get a snapshot of level 1 data
3. `L2`: to get a snapshot of level 2 data

AutoData Class

The `AutoData` class is optionally initialised with a configuration dictionary (either manually constructed, or the output of the utility function *`get_data_config`*. This dictionary contains any authentication information related to the data feed being used. Some samples are provided below.

```
data_config = {
    'data_source': 'ccxt',
    'exchange': 'binance'
}
ad = AutoData(data_config)
```

```
data_config = {
    'data_source': 'dydx',
}
ad = AutoData(data_config)
```

```
data_config = {
    'data_source': "oanda",
    'API': "api-fxtrade.oanda.com",
    'ACCESS_TOKEN': "xxx-yyy",
    'PORT': 443,
    'ACCOUNT_ID': "xxx-xxx-xxxxxxxx-xxx",
}
ad = AutoData(data_config)
```

Fetch historical OHLC data

To fetch OHLC price data, use the `fetch` method.

`AutoData.`**`fetch`**`(`*instrument: str*, *granularity: str | None = None*, *count: int | None = None*, *start_time: datetime | None = None*, *end_time: datetime | None = None*, **args*, ***kwargs*`)` → DataFrame

Unified OHLC data retrieval api.

Parameters

- **`instrument`** (*`str, list`*) – The instrument to fetch data for, or a list of instruments.
- **`granularity`** (*`str`*) – The granularity of the data to fetch.
- **`count`** (*`int, optional`*) – The number of OHLC bars to fetch. The default is None.
- **`start_time`** (*`datetime, optional`*) – The start date of the data to fetch. The default is None.
- **`end_time`** (*`datetime, optional`*) – The end date of the data to fetch. The default is None.

Returns
data – The OHLC data.

Return type
pd.DataFrame, dict[pd.DataFrame]

Get level 1 data

To retrieve a snapshot of the current level 1 data, use the `L1` method.

`AutoData.`**`L1`**`(`*instrument=None*, **args*, ***kwargs*`)`

Unified level 1 data retrieval api.

Get level 2 data

To retrieve a snapshot of the current level 2 data, use the `L2` method.

`AutoData.`**`L2`**`(`*instrument=None*, **args*, ***kwargs*`)`

Unified level 2 data retrieval api.

Accessing exchange endpoints

If you wish to access an endpoint of the feed-specific API, you can do so through the `AutoData.api` attribute. This attribute gets created when you instantiate AutoData with the `data_config` dictionary.

3.9.3 AutoPlot

```
from autotrader import AutoPlot
```

AutoPlot is the automated plotting module of AutoTrader. It is automatically used when visualising backtest results, but can also be interacted with to create charts and visualise indicators. Refer to this blog post for an example.

class `autotrader.autoplot.`**`AutoPlot`**(*data: DataFrame | Series | None = None*)

Automated trading chart generator.

`configure`()

Configure the plot settings.

`add_tool`(*tool_name*)

Add bokeh tool to plot. This adds the tool_name string to the fig_tools attribute.

`plot`()

Creates a trading chart of OHLC price data and indicators.

Methods

Configure

`AutoPlot.`**`configure`**(*max_indis_over: int | None = None*, *max_indis_below: int | None = None*, *fig_tools: str | None = None*, *ohlc_height: int | None = None*, *ohlc_width: int | None = None*, *top_fig_height: int | None = None*, *bottom_fig_height: int | None = None*, *jupyter_notebook: bool | None = None*, *show_cancelled: bool | None = None*, *chart_theme: str | None = None*, *use_strat_plot_data: bool = False*) → None

Configures the plot settings.

Parameters

- **max_indis_over** (*int, optional*) – Maximum number of indicators overlaid on the main chart. The default is 3.
- **max_indis_below** (*int, optional*) – Maximum number of indicators below the main chart. The default is 2.
- **fig_tools** (*str, optional*) – The figure tools. The default is "pan,wheel_zoom,box_zoom,undo, redo,reset,save,crosshair".
- **ohlc_height** (*int, optional*) – The height (px) of the main chart. The default is 400.
- **ohlc_width** (*int, optional*) – The width (px) of the main chart. The default is 800.
- **top_fig_height** (*int, optional*) – The height (px) of the figure above the main chart. The default is 150.
- **bottom_fig_height** (*int, optional*) – The height (px) of the figure(s) below the main chart. The default is 150.
- **jupyter_notebook** (*bool, optional*) – Boolean flag when running in Jupyter Notebooks, to allow inline plotting. The default is False.
- **show_cancelled** (*bool, optional*) – Show/hide cancelled trades. The default is True.
- **chart_theme** (*bool, optional*) – The theme of the Bokeh chart generated. The default is "caliber".

Returns
The plot settings will be saved to the active AutoTrader instance.

Return type
None

See also:

Refer to the Bokeh documentation for more details of the different themes.

Add Plot Tools

To customise the tools of the figures produced with AutoPlot, the `add_tool` should be used. This method simply appends the `tool_name` to the `fig_tools` attribute of the class instance. Refer to the Bokeh documentation for details on the tools available.

`AutoPlot.`**`add_tool`**`(`*tool_name: str*`)` → None

Adds a tool to the plot.

Parameters
`tool_name` (*`str`*) – The name of tool to add (see Bokeh documentation).

Returns
The tool will be added to the chart produced.

Return type
None

Create Plot

The `plot` method is used to generate a chart. It is used in both backtest plotting and indicator viewing.

`AutoPlot.`**`plot`**`(`*instrument: str* | *None = None*, *indicators: dict* | *None = None*, *trade_results:* TradeAnalysis | *None = None*, *show_fig: bool = True*`)` → None

Creates a trading chart of OHLC price data and indicators.

Extended Summary

The following lists the keys corresponding to various indicators, which can be included in the chart using the indicators argument.

over
[over] Generic line indicator over chart with key: data

below
[below] Generic line indicator below chart with key: data

MACD
[below] MACD indicator with keys: macd, signal, histogram

MA
[over] Moving average overlay indicator with key: data

RSI
[below] RSI indicator with key: data

Heikin-Ashi
[below] Heikin Ashi candlesticks with key: data

Supertrend
[over] Supertrend indicator with key: data

Swings
[over] Swing levels indicator with key: data

Engulfing
[below] Engulfing candlestick pattern with key: data

Crossover
[below] Crossover indicator with key: data

Grid
[over] Grid levels with key: data

Pivot
[over] Pivot points with keys: data

HalfTrend
[over] Halftrend indicator with key: data

multi
[below] Multiple indicator type

signals
[over] Trading signals plot with key: data

bands
[over] Shaded bands indicator type

threshold
[below] Threshold indicator type

trading-session
[over] Highlighted trading session times with key: data

bricks
[below] Price-based bricks with keys: data (DataFrame), timescale (bool)

param instrument
The traded instrument name. The default is None.

type instrument
str, optional

param trade_results
The TradeAnalysis results object. The default is None.

type trade_results
TradeAnalysis, optional

param indicators
Indicators dictionary. The default is None.

type indicators
dict, optional

param show_fig
Flag to show the chart. The default is True.

type show_fig
bool, optional

returns
Calling this method will automatically generate and open a chart.

rtype
None

See also:

`autotrader.autotrader.AutoTrader.analyse_backtest`

Usage

Indicator Specification

To plot indicators with AutoPlot, you must provide a dictionary containing each indicator. This dictionary must be structured according to the example provided below.

```
self.indicators = {'indicator 1 name': {'type': 'indicator 2 type',
                                        'data': self.indicator_data},
                   'indicator 2 name': {'type': 'indicator 1 type',
                                        'data': self.indicator_data},
                  ...
                  }
```

In this dictionary, each key 'indicator name' is used to create the legend entry corresponding to the indicator. The sub-dictionary assigned to each indicator contains the specific information and associated data. The `type` key should be a string corresponding to the type of indicator, as defined in the table below. It is used to determine whether the indicator should be plotted overlayed on the OHLC chart, or below it on a separate plot. Finally, the data associated with the indicator must also be provided. For indicators with one set of data, such as a moving average, simply provide the data with the `data` key. For indicators with multiple sets of data, such as MACD, provide a key for each set named according to the keys specified in the table below.

Indicator	Type	Keys
Generic overlay indicator	`over`	`data`
Generic below-figure indicator	`below`	`data`
MACD	`MACD`	`macd`, `signal`, `histogram`
EMA	`MA`	`data`
SMA	`MA`	`data`
RSI	`RSI`	`data`
Stochastics	`STOCHASTIC`	`K`, `D`
Heikin-Ashi	`Heikin-Ashi`	`data`
Supertrend	`Supertrend`	`data`
HalfTrend	`HalfTrend`	`data`
Swings	`Swings`	`data`
Engulfing	`Engulfing`	`data`
Crossover	`Crossover`	`data`
Pivot Points	`Pivot`	`data`, optionally `levels`
Multiple line plot	`multi`	See below
Shaded bands	`bands`	See below
Shaded threshold	`threshold`	See below
Trade signals	`signals`	`data`

Example Indicator Dictionary

In this dictionary, each key is used to create a legend entry corresponding to the indicator. The sub-dictionary assigned to each key contains the specific information and associated data. The `type` key is a string corresponding to the type of indicator, for example:

- `'type': 'MA'` for an exponential moving average (or any type of moving average)
- `'type': 'STOCH'` for stochastics Finally, the data associated with the indicator must also be provided. For indicators with one set of data, such as a moving average, simply provide the data with the `data` key. For indicators with multiple sets of data, such as MACD, provide a key for each set named according to the *indicator specification*. See the example below for a strategy with MACD, two RSI's and two EMA's.

```
self.indicators = {'MACD (12/26/9)': {'type': 'MACD',
                                      'macd': self.MACD,
                                      'signal': self.MACDsignal,
                                      'histogram': self.MACDhist},
                   'EMA (200)': {'type': 'MA',
                                 'data': self.ema200},
                   'RSI (14)': {'type': 'RSI',
                                'data': self.rsi14},
                   'RSI (7)': {'type': 'RSI',
                               'data': self.rsi7},
                   'EMA (21)': {'type': 'MA',
                                'data': self.ema21}
                  }
```

Multiple line plot

To plot multiple lines on the same figure, the `multi` indicator type can be used. In the example below, a figure with title 'Figure title' will be created below the OHLC chart. On this figure, two lines will be plotted, with legend names of 'Line 1 name' and 'Line 2 name'. Line 1 will be blue and line 2 will be red, as set using the 'color' key specifier.

```
indicator_dict = {'Figure title': {'type': 'multi',
                                   'Line 1 name': {'data': line1_data,
                                                   'color': 'blue'},
                                   'Line 2 name': {'data': line2_data,
                                                   'color': 'red'},
                                   }
                  }
```

Shaded bands plot

To plot shaded bands, such as Bollinger Bands®, the `bands` indicator type can be used. An example of using this indicator type is provided below.

```
indicator_dict = {'Bollinger Bands': {'type': 'bands',
                                      'lower': bb.lower,
                                      'upper': bb.upper,
                                      'mid': bb.mid,
                                      'mid_name': 'Bollinger Mid Line'},
                  }
```

The full list of keys which can be provided with the indicator type is shown in the table below.

Key	Required/Optional	Description	Default value
`band_name`	Optional	legend name for bands	String provided for indicator (eg. 'Bollinger Bands')
`fill_color`	Optional	color filling upper and lower bands	'blue'
`fill_alpha`	Optional	transparency of fill (0 - 1)	0.6
`mid`	Optional	data for a mid line	None
`mid_name`	Optional	legend name for mid line	'Band Mid Line'
`line_color`	Optional	line color for mid line	'black'

Shaded threshold plot

The `threshold` plot indicator type is the standalone-figure version of the `bands` indicator type. That is, instead of overlaying shaded bands on the OHLC chart, a new figure is created below. The same keys apply to this method as the keys of the `bands` indicator type, as documented in the table above. An example of using this indicator is provided below.

```
'RSI threshold': {'type': 'threshold',
                  'lower': 30,
                  'upper': 70,
                  'mid': rsi,
                  'mid_name': 'RSI'},
```

Trade signals plot

The `signals` plot indicator type can be used to overlay buy and sell signals onto the OHLC chart. To do so, pass a DataFrame with columns named "buy" and "sell" with the `data` key. Note that the values in these coloumns are the prices at which the signal occurs. This means that if you have a DataFrame with Booleans corresponding to the buy and sell points, you will need to multiply them by the price data to shift them.

Unrecognised indicator type

If an indicator type isn't recognised, AutoPlot will attempt to plot it as a line plot on a new chart below the OHLC chart using the `data` key of the indicator. A warning message stating that the indicator is not recognised will also be printed. Also note that a `type` key can be used for an indicator that isn't specified above if it has similar plotting behaviour. See the *indicators* for details on the indicators listed above.

Minimum Working Example

As a plotting class, each instance of AutoPlot must be provided with price data in the form of OHLC data. A minimal working example is provided below to visualise the price data of Apple from the Yahoo finance feed of *AutoData*.

```python
from autotrader import AutoPlot, AutoData

instrument = 'AAPL'
get_data = AutoData({'data_source': 'yahoo'})
data = get_data.fetch(instrument, '1d',
                      start_time='2020-01-01',
                      end_time='2021-01-01')

ap = AutoPlot(data)
ap.plot(instrument=instrument)
```

3.9.4 AutoBot

Every time you run a strategy in AutoTrader, a trading bot (from the class `AutoTraderBot`), will be deployed for each instrument in your strategy's watchlist. Each trading bot is therefore responsible for trading a single instrument using the rules of a single strategy, until it is terminated. All methods of the `AutoTraderBot` class are private, as it is unlikely you will ever need to call them as a user. However, you may want to use the bot instance for other things, such as *backtest plotting*. For this purpose, you can use the *`get_bots_deployed`* method of AutoTrader.

class `autotrader.autobot.`**`AutoTraderBot`**(*instrument: str*, *strategy_dict: dict*, *broker*, *deploy_dt: datetime*, *data_dict: dict*, *quote_data_path: str*, *auxdata: dict*, *autotrader_instance*)

AutoTrader Trading Bot.

`instrument`

The trading instrument assigned to the bot.

Type
str

`data`

The OHLC price data used by the bot.

Type
pd.DataFrame

`quote_data`

The OHLC quote data used by the bot.

Type
pd.DataFrame

`MTF_data`

The multiple timeframe data used by the bot.

Type
dict

`backtest_results`

A class containing results from the bot in backtest. This is available only after a backtest.

Type
TradeAnalysis

`_check_auxdata`(*auxdata: dict*, *timestamp: datetime*, *indexing: str = 'open'*, *tail_bars: int | None = None*, *check_for_future_data: bool = True*) → dict

Function to check the strategy auxiliary data.

Parameters

- **auxdata** (*dict*) – The strategy's auxiliary data.
- **timestamp** (*datetime*) – The current timestamp.
- **indexing** (*str, optional*) – How the OHLC data has been indexed (either by bar 'open' time, or bar 'close' time). The default is 'open'.
- **tail_bars** (*int, optional*) – If provided, the data will be truncated to provide the number of bars specified. The default is None.
- **check_for_future_data** (*bool, optional*) – A flag to check for future entries in the data. The default is True.

Returns
The checked auxiliary data.

Return type
dict

`_check_data`(*timestamp: datetime*, *indexing: str = 'open'*) → dict

Function to return trading data based on the current timestamp. If dynamc_data updates are required (eg. when livetrading), the datastream will be refreshed each update to retrieve new data. The data will then be checked to ensure that there is no future data included.

Parameters

- **timestamp** (*datetime*) – DESCRIPTION.
- **indexing** (*str, optional*) – DESCRIPTION. The default is 'open'.

Returns

- **strat_data** (*dict*) – The checked strategy data.
- **current_bars** (*dict(pd.core.series.Series)*) – The current bars for each product.
- **quote_bars** (*dict(pd.core.series.Series)*) – The current quote data bars for each product.
- **sufficient_data** (*bool*) – Boolean flag whether sufficient data is available.

`_check_last_bar`(*current_bars: dict*) → bool

Checks for new data to prevent duplicate signals.

`static _check_ohlc_data`(*ohlc_data: DataFrame*, *timestamp: datetime*, *indexing: str = 'open'*, *tail_bars: int | None = None*, *check_for_future_data: bool = True*) → DataFrame

Checks the index of inputted data to ensure it contains no future data.

Parameters

- `ohlc_data` (*pd.DataFrame*) – DESCRIPTION.
- `timestamp` (*datetime*) – The current timestamp.
- `indexing` (*str, optional*) – How the OHLC data has been indexed (either by bar 'open' time, or bar 'close' time). The default is 'open'.
- `tail_bars` (*int, optional*) – If provided, the data will be truncated to provide the number of bars specified. The default is None.
- `check_for_future_data` (*bool, optional*) – A flag to check for future entries in the data. The default is True.

Raises
`Exception` – When an unrecognised data indexing type is specified.

Returns
past_data – The checked data.

Return type
pd.DataFrame

`_check_orders`(*orders*) → list

Checks that orders returned from strategy are in the correct format.

Return type
List of Orders

Notes

An order must have (at the very least) an order type specified. Usually, the direction will also be required, except in the case of close order types. If an order with no order type is provided, it will be ignored.

`_check_strategy_for_plot_data`(*use_strat_plot_data: bool = False*)

Checks the bot's strategy to see if it has the plot_data attribute.

Returns
plot_data – The data to plot.

Return type
pd.DataFrame

Notes

This method is a placeholder for a future feature, allowing customisation of what is plotted by setting plot_data and plot_type attributes from within a strategy.

`_create_trade_results`(*broker_histories: dict*) → dict

Constructs bot-specific trade summary for post-processing.

`_get_iteration_range`() → int

Checks mode of operation and returns data iteration range. For backtesting, the entire dataset is iterated over. For livetrading, only the latest candle is used. ONLY USED IN BACKTESTING NOW.

`_qualify_orders`(*orders: list*, *current_bars: dict*, *quote_bars: dict*) → None

Passes price data to order to populate missing fields.

`_refresh_data`(*timestamp: datetime | None = None*, ***kwargs*)

Refreshes the active Bot's data attributes for trading.

When backtesting without dynamic data updates, the data attributes of the bot will be constant. When using dynamic data, or when livetrading in continuous mode, the data attributes will change as time passes, reflecting more up-to-date data. This method refreshes the data attributes for a given timestamp by calling the datastream object.

Parameters

- `timestamp` (*datetime, optional*) – The current timestamp. If None, datetime.now() will be called. The default is None.
- `**kwargs` (*dict*) – Any other named arguments.

Raises

`Exception` – When there is an error retrieving the data.

Returns

The up-to-date data will be assigned to the Bot instance.

Return type

None

`_replace_data`(*data: DataFrame*) → None

Function to replace the data assigned locally and to the strategy. Called when there is a mismatch in data lengths during multi-instrument backtests in periodic update mode.

`static _submit_order`(*broker*, *order*, **args*, ***kwargs*)

The default order execution method.

`_update`(*i: int | None = None*, *timestamp: datetime | None = None*) → None

Update strategy with the latest data and generate a trade signal.

Parameters

- `i` (*int, optional*) – The indexing parameter used when running in periodic update mode. The default is None.
- `timestamp` (*datetime, optional*) – The timestamp parameter used when running in continuous update mode. The default is None.

Returns

Trade signals generated will be submitted to the broker.

Return type

None

`_update_virtual_broker`(*current_bars: dict*) → None

Updates virtual broker with latest price data.

3.9.5 AutoTrader Utilities Module

```
from autotrader import utilities
```

Configuration Utilities

Get Broker Configuration

autotrader.utilities.**get_broker_config**(*broker: str*, *global_config: dict | None = None*, *environment: str = 'paper'*) → dict

Returns a broker configuration dictionary.

Parameters

- **broker** (*str*) – The name(s) of the broker/exchange. Specify multiple exchanges using comma separation.
- **global_config** (*dict*) – The global configuration dictionary.
- **environment** (*str, optional*) – The trading evironment ('demo' or 'real').

Get Data Configuration

autotrader.utilities.**get_data_config**(*feed: str*, *global_config: dict | None = None*, ***kwargs*) → dict

Returns a data configuration dictionary for AutoData. :param feed: The name of the data feed. :type feed: str :param global_config: The global configuration dictionary. :type global_config: dict

YAML Utilities

Read YAML

autotrader.utilities.**read_yaml**(*file_path: str*) → dict

Function to read and extract contents from .yaml file.

Parameters
file_path (*str*) – The absolute filepath to the yaml file.

Returns
The loaded yaml file in dictionary form.

Return type
dict

Write YAML

autotrader.utilities.**write_yaml**(*data: dict*, *filepath: str*) → None

Writes a dictionary to a yaml file.

Parameters

- **data** (*dict*) – The dictionary to write to yaml.
- **filepath** (*str*) – The filepath to save the yaml file.

Returns
The data will be written to the filepath provided.

Return type
None

Unpickle Broker Instance

`autotrader.utilities.`**`unpickle_broker`**(*picklefile: str = '.virtual_broker'*)

Unpickles a virtual broker instance for post-processing.

TradeAnalysis Class

`class autotrader.utilities.`**`TradeAnalysis`**(*broker*, *broker_histories: dict*, *instrument: str | None = None*, *price_history: DataFrame | None = None*)

AutoTrader trade analysis class.

`instruments_traded`

The instruments traded during the trading period.

Type
list

`account_history`

A timeseries history of the account during the trading period.

Type
pd.DataFrame

`holding_history`

A timeseries summary of holdings during the trading period, by portfolio allocation fraction.

Type
pd.DataFrame

`order_history`

A timeseries history of orders placed during the trading period.

Type
pd.DataFrame

`cancelled_orders`

Orders which were cancelled during the trading period.

Type
pd.DataFrame

`trade_history`

A history of all trades (fills) made during the trading period.

Type
pd.DataFrame

`analyse_account`(*broker*, *broker_histories: dict*, *instrument: str | None = None*) → None

Analyses trade account and creates summary of key details.

`static create_fill_summary`(*fills: list*, *broker_name: str | None = None*)

Creates a dataframe of fill history.

`static create_position_history`(*trade_history: DataFrame*, *account_history: DataFrame*) → DataFrame

Creates a history of positions held, recording number of units held at each timestamp.

`static create_trade_summary`(*trades: dict | None = None*, *orders: dict | None = None*, *instrument: str | None = None*, *broker_name: str | None = None*) → DataFrame

Creates a summary dataframe for trades and orders.

`summary`() → dict

Constructs a trading summary for printing.

DataStream Class

`class autotrader.utilities.DataStream`(***kwargs*)

Data stream class.

This class is intended to provide a means of custom data pipelines.

`instrument`

The instrument being traded.

Type
str

`feed`

The data feed.

Type
str

`data_filepaths`

The filepaths to locally stored data.

Type
str|dict

`quote_data_file`

The filepaths to locally stored quote data.

Type
str

`auxdata_files`

The auxiliary data files.

Type
dict

`strategy_params`

The strategy parameters.

Type
dict

`get_data`
The AutoData instance.

Type
AutoData

`data_start`
The backtest start date.

Type
datetime

`data_end`
The backtest end date.

Type
datetime

`portfolio`
The instruments being traded in a portfolio, if any.

Type
bool|list

`data_path_mapper`
A callable to map an instrument to an absolute filepath of data for that instrument.

Type
callable

Notes

A 'dynamic' dataset is one where the specific products being traded change over time. For example, trading contracts on an underlying product. In this case, dynamic_data should be set to True in AutoTrader.add_data method. When True, the datastream will be refreshed each update interval to ensure that data for the relevant contracts are being provided.

When the data is 'static', the instrument being traded does not change over time. This is the more common scenario. In this case, the datastream is only refreshed during livetrading, to accomodate for new data coming in. In backtesting however, the entire dataset can be provided after the initial call, as it will not evolve during the backtest. Note that future data will not be provided to the strategy; instead, the data returned from the datastream will be filtered by each AutoTraderBot before being passed to the strategy.

`get_trading_bars`(*data: DataFrame*, *quote_bars: bool*, *timestamp: datetime | None = None*, *processed_strategy_data: dict | None = None*) → dict

Returns a dictionary of the current bars of the products being traded, based on the up-to-date data passed from autobot.

Parameters

- **data** (*pd.DataFrame*) – The strategy base OHLC data.
- **quote_bars** (*bool*) – Boolean flag to signal that quote data bars are being requested.
- **processed_strategy_data** (*dict*) – A dictionary containing all of the processed strategy data, allowing flexibility in what bars are returned.

Returns
A dictionary of OHLC bars, keyed by the product name.

Return type
dict

Notes

The quote data bars dictionary must have the exact same keys as the trading bars dictionary. The quote_bars boolean flag is provided in case a distinction must be made when this method is called.

`match_quote_data`(*data: DataFrame*, *quote_data: DataFrame*) → DataFrame

Function to match index of trading data and quote data.

`refresh`(*timestamp: datetime | None = None*)

Returns up-to-date trading data for AutoBot to provide to the strategy.

Parameters
`timestamp` (*`datetime, optional`*) – The current timestamp, which can be used to fetch data if need. Note that look-ahead is checked for in AutoTrader.autobot, so the data returned from this method can include all available data. The default is None.

Returns

- **data** (*pd.DataFrame*) – The OHLC price data.
- **multi_data** (*dict*) – A dictionary of DataFrames.
- **quote_data** (*pd.DataFrame*) – The quote data.
- **auxdata** (*dict*) – Strategy auxiliary data.

Monitor

`class autotrader.utilities.`**`Monitor`**(*config_filepath: str | None = None*, *config: dict | None = None*, **args*, ***kwargs*)

`get_broker`(*broker*) → *AbstractBroker*

Returns the broker object.

`run`() → None

Runs the monitor indefinitely.

`static start_server`(*port*)

Starts the http server for Prometheus.

This page provides a top-level overview of AutoTrader, and how everything is tied together.

3.9.6 Module Overview

The table below provides a summary of the modules available.

Module	Description
AutoTrader	The primary API, used for all trading purposes.
AutoData	The data retrieval API, used by AutoTrader and for manual use.
AutoPlot	The automated plotting tool, used by AutoTrader and for manual use.
AutoBot	A trading bot, used to manage data and run strategies.
Utilities	A collection of tools and utilities to make everything work.

3.9.7 Code Workflow

AutoTrader follows a logical procedure when running a trading strategy (or multiple strategies). This is summarised in the flowchart below. Note that the flowchart below exemplifies running two trading strategies with six instrument-strategy pairs (hence six trading bots). However, it is possible to run AutoTrader with as many or as few strategies and instruments as you would like.

User Input Files

To run AutoTrader, a *strategy module*, containing the trading strategy, is required. Each strategy module requires it's own *strategy configuration* file, containing the strategy parameters and strategy watchlist. There is a second configuration file, the *global configuration* file, which is used conditionally. If you are live trading, you will need to create a global configuration file to provide brokerage account details. You will also need to do this if you wish to use a broker to obtain price data. If you will are only backtesting, you do not need to provide a global configuration file. In this case, *AutoData* will revert to using the Yahoo Finance API for price data.

AutoTrader

AutoTrader provides the the skeleton to your trading framework - read the complete documentation for it *here*. In brief, AutoTrader loads each strategy, deploys trading bots, and monitors them for as long as they are trading.

The mechanism by which the bots are deployed depends on the selected *run mode* of AutoTrader. Bots can either be periodically updated with new data, or run continuously without stopping. When running periodically, the bots will be deployed each time the data is updated, and terminated after executing the strategy.

Broker API Connection

Each bot will also be connected to one or more *broker APIs*. When they recieve a signal from the trading strategy, they will place an order with the broker. This modular structure allows for a seamless transition from backtesting to livetrading.

Shutdown Routines

AutoTrader also supports the inclusion of strategy-specific shutdown routines. This includes any processes you would like to run *after* your strategy is finished trading. This may include writing data to file, pickling the strategy instance, or sending termination emails. Read more about this functionality *here*.

3.10 AutoTrader Broker Interface

3.10.1 Trading Objects

Orders

Orders can be created using the `Order` object, shown below. The different order types and requirements are documented following.

`class autotrader.brokers.trading.Order`(*instrument: str | None = None*, *direction: int | None = None*, *order_type: str = 'market'*, *size: float | None = None*, *order_limit_price: float | None = None*, *order_stop_price: float | None = None*, *stop_loss: float | None = None*, *stop_type: str | None = None*, *take_profit: float | None = None*, ***kwargs*)

AutoTrader Order object.

`instrument`

The trading instrument of the order.

Type
str

`direction`

The direction of the order (1 for long, -1 for short).

Type
int

`order_type`

The type of order. The default is 'market'.

Type
str

`size`

The number of units.

Type
float

`base_size`

The number of units, in the base currency (pre-HCF conversion).

Type
float

`target_value`

The target value of the resulting trade, specified in the home currency of the account.

Type
float

`order_limit_price`

The limit price of the order (for 'limit' and 'stop-limit' order types).

Type
float

`order_stop_price`

The stop price of the order (for 'stop-limit' order types).

Type
float

`order_price`

The price of the instrument when the order was placed.

Type
float

`order_time`

The time at which the order was placed.

Type
datetime

`stop_loss`

The price to set the stop-loss at.

Type
float

`stop_distance`

The pip distance between the order price and the stop-loss.

Type
float

`stop_type`

The type of stop-loss (limit or trailing). The default is 'limit'.

Type
str

`take_profit`

The price to set the take-profit at.

Type
float

`take_distance`

The pip distance between the order price and the take-profit.

Type
float

`related_orders`

A list of related order/trade ID's.

Type
list

`id`

The order ID.

Type
int

`pip_value`

The pip value of the product being traded. Specify this for non-FX products when using stop_distance/take_distance arguments. The default is None.

Type
float, optional

`currency`

The base currency of the order (IB only).

Type
str

`secType`

The security type (IB only).

Type
str

`contract_month`

The contract month string (IB only).

Type
str

`localSymbol`

The exchange-specific instrument symbol (IB only).

Type
str

`post_only`

Enforce that the order is placed as a maker order (dYdX only). The default is False.

Type
bool, optional

`limit_fee`

The maximum fee to accept as a percentage (dYdX only). The default is '0.015'.

Type
str, optional

`exchange`

The exchange to which the order should be submitted.

Type
str

`ccxt_params`

The CCXT parameters dictionary to pass when creating an order. The default is {}.

Type
dict, optional

`__init__`(*instrument: str | None = None*, *direction: int | None = None*, *order_type: str = 'market'*, *size: float | None = None*, *order_limit_price: float | None = None*, *order_stop_price: float | None = None*, *stop_loss: float | None = None*, *stop_type: str | None = None*, *take_profit: float | None = None*, ***kwargs*) → *Order*

`_calculate_exit_prices`(*broker=None*, *working_price: float | None = None*) → None

Calculates the prices of the exit targets from the pip distance values.

Parameters

- **broker** (*AutoTrader Broker Interface, optional*) – The autotrade-broker instance. The default is None.
- **working_price** (*float, optional*) – The working price used to calculate amount risked. The default is None.

Returns
The exit prices will be assigned to the order instance.

Return type
None

`_calculate_position_size`(*broker=None*, *working_price: float | None = None*, *HCF: float = 1*, *risk_pc: float = 0*, *sizing: str | float = 'risk'*, *amount_risked: float | None = None*) → None

Calculates trade size for order.

Parameters

- **broker** (*AutoTrader Broker Interface, optional*) – The autotrade-broker instance. The default is None.
- **working_price** (*float, optional*) – The working price used to calculate amount risked. The default is None.
- **HCF** (*float, optional*) – The home conversion factor. The default is 1.
- **risk_pc** (*float, optional*) – The percentage of the account NAV to risk on the trade. The default is 0.
- **sizing** (*str | float, optional*) – The sizing option. The default is 'risk'.
- **amount_risked** (*float, optional*) – The dollar amount risked on the trade. The default is None.

Returns
The trade size will be assigned to the order instance.

Return type
None

`_set_working_price`(*order_price: float | None = None*) → None

Sets the Orders' working price, for calculating exit targets.

Parameters
order_price (*float, optional*) – The order price.

Returns
The working price will be saved as a class attribute.

Return type
None

`as_dict`() → dict

Converts Order object to dictionary.

Returns
The order instance returned as a dict object.

Return type
dict

Notes

This method enables legacy code operation, returning order/trade objects as a dictionary.

Summary of Order Types

AutoTrader is intelligent when it comes to order types. If your strategy has no stop loss, you do not need to include it in the signal dictionary. If you prefer to set a stop loss in terms of distance in pips, you can do that instead. Same goes for take profit levels, specify price or distance in pips - whatever is more convenient. The following tables provides accepted values for the `order_type` of an order.

Order Type	Description	Additional keys required
`market`	A market order type	None
`limit`	A limit order type	`order_limit_price`
`stop-limit`	A stop-limit order type	`order_limit_price` and `order_stop_price`

Empty Order

Useful when no signal is present. AutoTrader will recognise this as an empty order and skip over it.

```
empty_order = Order()
```

Market Order

A market order triggers a trade immediately at the best available price, provided there is enough liquidity to accomodate the order. Read more here. Note that this is the default order type, so `order_type` does not need to be specified.

```
long_market_order = Order(direction=1)
short_market_order = Order(direction=-1)
```

Limit Order

A limit order is allows a trader to buy or sell an instrument at a specified price (or better). Limit orders can be used to avoid slippage. However, a limit order is not gauranteed to be filled. When using this order type, the limit price must be specified. Read more about limit orders here.

```
limit_order = Order(direction=1, order_type='limit', order_limit_price=1.2312)
```

Stop-Limit Order

A stop-limit order type provides a means to place a conditional limit order. This order type can be used to place an order, under the condition that price first moves to the stop price, at which point, a limit order is placed. In addition to a limit price, a stop price must also be specified. These prices are usually the same, although need not be. Read more about stop-limit orders here.

```
stop_limit_order = Order(direction=1, order_type='stop-limit',
                         order_limit_price=1.2312, order_stop_price=1.2300)
```

Stop Loss Types

AutoTrader supports both limit stop loss orders, and trailing stop loss orders. The keys required to specify these stop loss types are provided in the table below. Note that `limit` stop losses are the default type, so if a `stop_price` (or `stop_distance`) is provided with no `stop_type`, a limit stop-loss will be placed.

Stop Type	Description	Additional keys required
`limit`	A regular stop loss order	None
`trailing`	A trailing stop loss order	`stop_loss` or `stop_distance`

A note about setting stop losses: the `stop_price` takes precedence over `stop_distance`. That is, if `stop_price` is provided along with a `stop_distance`, the stop loss will be set at the price defined by `stop_price`. To avoid ambiguity, only specify one.

Note that for the purpose of calculating trading fees, stop loss orders are treated as `limit` order type (liquidity providing).

Limit Stop Loss

`stop_type='limit'`

A `'limit'` stop loss type will place a limit order at the price specified by the `stop_loss` key. If the price crosses a stop loss order, the trade associated with it will be closed. Read more about stop losses here.

Trailing Stop Loss

`stop_type='trailing'`

A trailing stop loss can be used to protect unrealised gains by moving with price in favour of the trade direction. When using the `trailing` stop loss type, the initial stop loss position can be set by the price provided to the `stop_loss` key, or by providing the stop loss distance (in pips) to the `stop_distance` key. Read more about trailing stop loss orders here.

Take Profit Orders

Take-profit orders can be attached to an order using the `take_profit` or `take_price` attribute. Note that take profit orders are treated as `market` order type (liquidity consuming) when calculating trading costs.

Trades

class `autotrader.brokers.trading.`**`Trade`**(*instrument: str*, *order_price: float*, *order_time: datetime*, *order_type: str*, *size: float*, *last_price: float*, *fill_time: datetime*, *fill_price: float*, *fill_direction: int*, *fee: float*, ***kwargs*)

Bases: `object`

AutoTrader Trade object. Represents an exchange of value.

`__init__`(*instrument: str*, *order_price: float*, *order_time: datetime*, *order_type: str*, *size: float*, *last_price: float*, *fill_time: datetime*, *fill_price: float*, *fill_direction: int*, *fee: float*, ***kwargs*) → *Trade*

Trade constructor.

Isolated Positions

class `autotrader.brokers.trading.`**`IsolatedPosition`**(*order: Order | None = None*, ***kwargs*)

AutoTrader IsolatedPosition. Use to connect SL and TP orders to individual trades.

`unrealised_PL`

The floating PnL of the trade.

Type
float

`margin_required`

The margin required to maintain the trade.

Type
float

`time_filled`

The time at which the trade was filled.

Type
datetime

`fill_price`

The price at which the trade was filled.

Type
float

`last_price`

The last price observed for the instrument associated with the trade.

Type
float

`last_time`

The last time observed for the instrument associated with the trade.

Type
datetime

`exit_price`
The price at which the trade was closed.

Type
float

`exit_time`
The time at which the trade was closed.

Type
datetime

`fees`
The fees associated with the trade.

Type
float

`parent_id`
The ID of the order which spawned the trade.

Type
int

`id`
The trade ID.

Type
int

`status`
The status of the trade (open or closed).

Type
str

`split`
If the trade has been split.

Type
bool

Notes

When a trade is created from an Order, the Order will be marked as filled.

Positions

class `autotrader.brokers.trading.`**`Position`**(***kwargs*)

AutoTrader Position object.

`instrument`

The trade instrument of the position.

Type
str

`pnl`

The pnl of the position.

Type
float

`long_units`

The number of long units in the position.

Type
float

`long_PL`

The PnL of the long units in the position.

Type
float

`long_margin`

The margin required to maintain the long units in the position.

Type
float

`short_units`

The number of short units in the position.

Type
float

`short_PL`

The PnL of the short units in the position.

Type
float

`short_margin`

The margin required to maintain the short units in the position.

Type
float

`total_margin`

The total margin required to maintain the position.

Type
float

`trade_IDs`
The trade ID's associated with the position.

Type
list[int]

`net_position`
The total number of units in the position.

Type
float

`net_exposure`
The net exposure (in $ value) of the position.

Type
float

`PL`
The floating PnL (IB only).

Type
float

`contracts`
The contracts associated with the position (IB only).

Type
list

`portfolio_items`
The portfolio items associated with the position (IB only).

Type
list

`as_dict()` → dict
Converts Position object to dictionary.

Returns
The Position instance returned as a dict object.

Return type
dict

Notes

This method enables legacy code operation, returning order/trade objects as a dictionary.

3.10.2 Virtual Broker

`broker='virtual'`

The virtual broker immitates the functionality of a real broker for the purpose of backtesting and live papertrading.

Virtual Account Configuration

Whenever the virtual broker is used (for example in backtesting or papertrading), the virtual trading account must be configured using the `virtual_account_config` method. If multiple brokers are being used simultaneously, this method must be called once for each broker.

When a real broker/exchange is specified in this method, the instance of AutoData created for data management will be connected to the broker specified.

Internal Position Management

Orders

Orders can take one of four statuses:

1. `pending` - a pending order is one which has been submitted, but is being held until new data arrives.
2. `open` - an open order is one which is valid to be filled. In the case of a market order, it will be filled as soon as new data is seen. In the case of limit orders, the order will remain open until its limit price has been triggered.
3. `filled` - after an order has been filled, its status is changed to 'filled'.
4. `cancelled` - if an order is invalid, or gets cancelled by the user, its status will be changed to 'cancelled'.

Positions

Positions in an instrument are the result of orders being filled and trades being made.

In order to keep track of stop-losses and take-profits associated with individual orders, a position is made up of multiple `IsolatedPositions`. These are positions resulting from a single trade, and are treated in isolation of the entire position in case there is a stop-loss or take-proft attached to them.

Trade Execution

The virtual broker maintains its own orderbook. The details of this depend on whether AutoTrader is in backtest or livetrade mode.

Backtest Mode

When backtesting, the top levels of the book are simulated to have infinite liquidity. The bid and ask prices are set using the OHLC data and the specified bid/ask spread model.

Livetrade Mode

When livetrading (including papertrading), the execution of orders can become more accurate by tapping into the real-time orderbook of your chosen exchange. To do so, the broker/exchange specified in `AutoTrader.configure` and `AutoTrader.virtual_account_config` is connected to when creating an instance of the virtual broker. When orders are recieved, the real-time order book will be queried and used to simulate execution.

API Reference

`class autotrader.brokers.virtual.broker.`**`Broker`**(*broker_config: dict | None = None*, *utils: BrokerUtils | None = None*)

Autotrader Virtual Broker for simulated trading.

`verbosity`

The verbosity of the broker.

Type
int

`pending_orders`

A dictionary containing pending orders.

Type
dict

`open_orders`

A dictionary containing open orders yet to be filled.

Type
dict

`filled_orders`

A dictionary containing filled orders.

Type
dict

`cancelled_orders`

A dictionary containing cancelled orders.

Type
dict

`open_trades`

A dictionary containing currently open trades (fills).

Type
dict

`closed_trades`

A dictionary containing closed trades.

Type
dict

`base_currency`

The base currency of the account. The default is ‘AUD’.

Type
str

`NAV`
The net asset value of the account.

Type
float

`equity`
The account equity balance.

Type
float

`floating_pnl`
The floating PnL.

Type
float

`margin_available`
The margin available on the account.

Type
float

`leverage`
The account leverage.

Type
int

`spread`
The average spread to use when opening and closing trades.

Type
float

`spread_units`
The units of the spread (eg. ‘price’ or ‘percentage’). The default is ‘price’.

Type
str

`hedging`
Flag whethere hedging is enabled on the account. The default is False.

Type
bool

`margin_closeout`
The fraction of margin available at margin call. The default is 0.

Type
float

`commission_scheme`
The commission scheme being used (‘percentage’, ‘fixed_per_unit’ or ‘flat’). The default is ‘percentage’.

Type
str

`commission`
The commission value associated with the commission scheme.

Type
float

`maker_commission`
The commission value associated with liquidity making orders.

Type
float

`taker_commission`
The commission value associated with liquidity taking orders.

Type
float

`_add_orders_to_book`(*instrument*, *orderbook*)
Adds local orders to the orderbook.

`_adjust_balance`(*amount: float*, *latest_time: datetime | None = None*) → None
Adjusts the balance of the account.

`_calculate_commissions`(*price: float*, *units: float | None = None*, *HCF: float = 1*, *order_type: str = 'market'*) → float
Calculates trade commissions.

`_calculate_margin`(*position_value: float*) → float
Calculates margin required to take a position with the available leverage of the account.

`_fill_order`(*last_price: float*, *fill_price: float*, *fill_time: datetime*, *order: Order | None = None*, *liquidation_order: bool = False*) → None
Marks an order as filled and records the trade as a Fill.

Parameters

- **fill_price** (*float*) – The fill price.
- **fill_time** (*datetime*) – The time at which the order is filled.
- **order** (*Order, optional*) – The order to fill. The default is None, in which case the arguments below must be specified.
- **liquidation_order** (*bool, optional*) – A flag whether this is a liquidation order from the broker.

`_load_state`()
Loads the state of the broker from a pickle.

`_make_deposit`(*deposit: float*) → None
Adds deposit to account balance and NAV.

`_margin_call`(*instrument: str*, *latest_time: datetime*, *latest_price: float*)
Closes the open position of the instrument.

`_modify_position`(*trade: Trade*, *reduce_only: bool*)
Modifies the position in a position.

_move_order(*order:* Order, *from_dict: str = '_open_orders'*, *to_dict: str = '_filled_orders'*, *new_status: str = 'filled'*) → None

Moves an order from the from_dict to the to_dict.

_process_order(*order:* Order, *fill_time: datetime | None = None*, *reference_price: float | None = None*, *trade_size: float | None = None*)

Processes an order, either filling or cancelling it.

Parameters

- **order** (Order) – The order being processed.
- **fill_time** (*datetime, optional*) – The time to fill the order.
- **reference_price** (*float, optional*) – The order reference price (either market price or order limit price).
- **trade_size** (*float, optional*) – The size of a public trade being used to fill orders (papertrade mode). The default is None.

_public_trade(*instrument: str*, *trade: dict*)

Uses a public trade to update virtual orders.

_save_state()

Pickles the current state of the broker.

_trade_through_book(*instrument: str*, *direction: int*, *size: float*, *reference_price: float | None = None*, *precision: int | None = None*) → float

Returns an average fill price by filling an order through the orderbook.

Parameters

- **instrument** (*str*) – The instrument to fetch the orderbook for.
- **direction** (*int*) – The direction of the trade (1 for long, -1 for short). Used to specify either bid or ask prices.
- **size** (*float*) – The size of the trade.
- **reference_price** (*float, optional*) – The reference price to use if artificially creating an orderbook.
- **precision** (*dict, optional*) – The precision to use for rounding prices. The default is None.

_update_all()

Convenience method to update all open positions when paper trading.

_update_instrument(*instrument*)

Convenience method to update a single instrument when paper trading.

_update_margin(*instrument: str | None = None*, *latest_time: datetime | None = None*) → None

Updates the margin available in the account.

_update_positions(*instrument: str*, *candle: Series | None = None*, *L1: dict | None = None*, *trade: dict | None = None*) → None

Updates orders and open positions based on the latest data.

Parameters

- **instrument** (*str*) – The name of the instrument being updated.
- **candle** (*pd.Series*) – An OHLC candle used to update orders and trades.

- **L1** (*dict, optional*) – A dictionary a containing level 1 price snapshot to update the positions with. This dictionary must have the keys 'bid', 'ask', 'bid_size' and 'ask_size'.
- **trade** (*dict, optional*) – A public trade, used to update virtual limit orders.

`static _zero_slippage_model`(**args*, ***kwargs*)

Returns zero slippage.

cancel_order(*order_id: int*, *reason: str | None = None*, *from_dict: str = '_open_orders'*, *timestamp: datetime | None = None*, ***kwargs*) → None

Cancels the order.

Parameters

- **order_id** (*int*) – The ID of the order to be cancelled.
- **reason** (*str, optional*) – The reason why the order is being cancelled. The default is None.
- **from_dict** (*str, optional*) – The dictionary currently holding the order. The default is 'open_orders'.
- **timestamp** (*datetime, optional*) – The datetime stamp of the order cancellation. The default is None.

configure(*verbosity: int | None = None*, *initial_balance: float | None = None*, *leverage: int | None = None*, *spread: float | None = None*, *spread_units: str | None = None*, *commission: float | None = None*, *commission_scheme: str | None = None*, *maker_commission: float | None = None*, *taker_commission: float | None = None*, *hedging: bool | None = None*, *base_currency: str | None = None*, *paper_mode: bool | None = None*, *public_trade_access: bool | None = None*, *margin_closeout: float | None = None*, *default_slippage_model: Callable | None = None*, *slippage_models: dict | None = None*, *charge_funding: bool | None = None*, *funding_history: DataFrame | None = None*, *autodata_config: dict | None = None*, *picklefile: str | None = None*, ***kwargs*)

Configures the broker and account settings.

Parameters

- **verbosity** (*int, optional*) – The verbosity of the broker. The default is 0.
- **initial_balance** (*float, optional*) – The initial balance of the account, specified in the base currency. The default is 0.
- **leverage** (*int, optional*) – The leverage available. The default is 1.
- **spread** (*float, optional*) – The bid/ask spread to use in backtest (specified in units defined by the spread_units argument). The default is 0.
- **spread_units** (*str, optional*) – The unit of the spread specified. Options are 'price', meaning that the spread is quoted in price units, or 'percentage', meaning that the spread is quoted as a percentage of the market price. The default is 'price'.
- **commission** (*float, optional*) – Trading commission as percentage per trade. The default is 0.
- **commission_scheme** (*str, optional*) – The method with which to apply commissions to trades made. The options are (1) 'percentage', where the percentage specified by the commission argument is applied to the notional trade value, (2) 'fixed_per_unit', where the monetary value specified by the commission argument is multiplied by the number of units in the trade, and (3) 'flat', where a flat monetary value specified by the commission argument is charged per trade made, regardless of size. The default is 'percentage'.

- **maker_commission** (*float, optional*) – The commission to charge on liquidity-making orders. The default is None, in which case the nominal commission argument will be used.
- **taker_commission** (*float, optional*) – The commission to charge on liquidity-taking orders. The default is None, in which case the nominal commission argument will be used.
- **hedging** (*bool, optional*) – Allow hedging in the virtual broker (opening simultaneous trades in oposing directions). The default is False.
- **base_currency** (*str, optional*) – The base currency of the account. The default is AUD.
- **paper_mode** (*bool, optional*) – A boolean flag to indicate if the broker is in paper trade mode. The default is False.
- **public_trade_access** (*bool, optional*) – A boolean flag to signal if public trades are being used to update limit orders. The default is False.
- **margin_closeout** (*float, optional*) – The fraction of margin usage at which a margin call will occur. The default is 0.
- **default_slippage_model** (*Callable, optional*) – The default model to use when calculating the percentage slippage on the fill price, for a given order size. The default functon returns zero.
- **slippage_models** (*dict, optional*) – A dictionary of callable slippage models, keyed by instrument.
- **charge_funding** (*bool, optional*) – A boolean flag to charge funding rates. The default is False.
- **funding_history** (*pd.DataFrame, optional*) – A DataFrame of funding rate histories for instruments being traded, to backtest trading perpetual futures. This is a single frame with as many columns as instruments being traded. If an instrument is not present, the funding rate will be zero.
- **picklefile** (*str, optional*) – The filename of the picklefile to load state from. If you do not wish to load from state, leave this as None. The default is None.

get_NAV() → float

Returns Net Asset Value of account.

get_balance() → float

Returns balance of account.

get_margin_available() → float

Returns the margin available on the account.

get_orderbook(*instrument: str*, *midprice: float | None = None*) → OrderBook

Returns the orderbook.

get_orders(*instrument: str | None = None*, *order_status: str = 'open'*) → dict

Returns orders of status order_status.

get_positions(*instrument: str | None = None*) → dict

Returns the positions held by the account, sorted by instrument.

Parameters

instrument (*str, optional*) – The trading instrument name (symbol). If 'None' is provided, all positions will be returned. The default is None.

Returns
open_positions – A dictionary containing details of the open positions.

Return type
dict

Notes

net_position: refers to the number of units held in the position.

get_trades(*instrument: str | None = None*, ***kwargs*) → dict

Returns fills for the specified instrument.

Parameters
instrument (*str, optional*) – The instrument to fetch trades under. The default is None.

place_order(*order:* Order, ***kwargs*) → None

Place order with broker.

3.10.3 Oanda Broker API

`broker='oanda'`

Supported Features

Feature	Supported?	Alternative
Stop loss	Yes	N/A
Take profit	Yes	N/A

Configuration

Trading through dYdX requires the following configuration details.

```
OANDA:
  LIVE_API: "api-fxtrade.oanda.com"
  PRACTICE_API: "api-fxpractice.oanda.com"
  ACCESS_TOKEN: "12345678900987654321-abc34135acde13f13530"
  DEFAULT_ACCOUNT_ID: "xxx-xxx-xxxxxxxx-001"
  PORT: 443
```

```
{"OANDA":
   {
      "LIVE_API": "api-fxtrade.oanda.com",
      "PRACTICE_API": "api-fxpractice.oanda.com",
      "ACCESS_TOKEN": "12345678900987654321-abc34135acde13f13530",
      "DEFAULT_ACCOUNT_ID": "xxx-xxx-xxxxxxxx-001",
      "PORT": 443
   }
}
```

API Reference

3.10.4 Interactive Brokers

`broker='ib'`

Supported Features

Feature	Supported?	Alternative
Stop loss	No	Implement manually in strategy with limit orders
Take profit	No	Implement manually in strategy with stop-limit orders

API Reference

3.10.5 dYdX Exchange Interface

`broker='dydx'`

dYdX is a decentralised cryptocurrency derivatives exchange.

Supported Features

Feature	Supported?	Alternative
Stop loss	No	Implement manually in strategy with limit orders
Take profit	No	Implement manually in strategy with stop-limit orders

Configuration

Trading through dYdX requires the following configuration details.

```yaml
dYdX:
  ETH_ADDRESS: "0xxxxx"
  ETH_PRIV_KEY: "xxxxx"
```

```json
{"dYdX":
   {
      "ETH_ADDRESS": "0xxxxx",
      "ETH_PRIV_KEY": "xxxxx",
   }
}
```

API Reference

3.10.6 CCXT Exchange Interface

`broker='ccxt:<exchange name>'`

The CryptoCurrency eXchange Trading (CCXT) library is an open-source Python library supporting over 100 cryptocurrency exchange markets and trading APIs.

Specifying the Exchange

CCXT serves as a unified API for many different cryptocurrency exchanges. To trade with one of the supported exchanges, you must specify the name of the exchange after providing the `ccxt` key. For example, to trade with Binance, you would specify:

```
broker=ccxt:binance
```

Supported Features

Feature	Supported?	Alternative
Stop loss	No	Implement manually in strategy with limit orders
Take profit	No	Implement manually in strategy with stop-limit orders

Configuration

Trading through CCXT requires the following configuration details. Note that the heading key given for each exchange must match both the format specified above (eg. `ccxt:binance`), and the exchange name format required by CCXT.

```yaml
CCXT:EXCHANGE:
  api_key: "xxxx"
  secret: "xxxx"
  password: "abcDEF"
  base_currency: "USDT"
  options:
    defaultType: "swap"
```

```json
{"CCXT:EXCHANGE":
   {
      "api_key": "xxxx",
      "secret": "xxxx",
      "password": "abcDEF"
      "base_currency": "USDT",
      "options": {
         "defaultType": "swap",
      }
   }
}
```

Mainnet and Testnet Configurations

If you have api keys for an exchanges mainnet and testnet, you can include both in your `keys.yaml` file using the format shown below. The `mainnet` configuration is used when the *`environment`* is set to `"live"`, while the `testnet` configuration is used when it is set to `"paper"`.

```yaml
CCXT:EXCHANGE:
  mainnet:
    api_key: "xxxx"
    secret: "xxxx"
    password: "abcDEF"
    base_currency: "USDT"
    options:
      defaultType: "swap"
  testnet:
    api_key: "yyyy"
    secret: "yyyy"
    password: "ABC123"
    base_currency: "USDT"
    options:
      defaultType: "swap"
```

API Reference

To make the transition from backtesting to live-trading seamless, each broker integrated into AutoTrader interfaces using a set of common methods. This means that the same strategy can be run with the virtual broker or any other broker without changing a single line of code in your strategy. This page provides a general overview of the methods contained within each broker interface module. Of course, the mechanics behind each method will change depending on the broker, however each method will behave in the same way, accepting the same input arguments and outputting the same objects.

See also:

A great way to learn the broker interface is to do some *click trading*.

3.10.7 Methods

The shared methods of the broker interfaces are described below. Note that each broker may have their own additional methods to extend their functionality based on the brokers API capabilities.

Tip: A template for integrating new brokers into AutoTrader is included in the 'templates/' directory of the GitHub repository.

Method	Function
`get_NAV`	Returns Net Asset Value of account.
`get_balance`	Returns balance of account.
`place_order`	Place order with broker.
`get_orders`	Returns orders.
`cancel_order`	Cancels an order by ID.
`get_trades`	Returns open trades for the specified instrument.
`get_positions`	Returns the open positions in the account.

Accessing Exchange API Methods

When trading with a real exchange/broker, you are also able to communicate directly with their API endpoints via the `api` attribute of the broker instance. This allows you to access all methods offered by an exchange, outside of the unified methods listed in the table above.

3.10.8 Accessing the Broker Instance

To access the broker instance from your strategy, set `INCLUDE_BROKER` to True in your *strategy configuration*. Doing so, the broker instance and broker utilities will be passed as named arguments `broker` and `broker_utils` to your strategy's `__init__` method.

3.10.9 Module Structure

Each new broker API is contained within its own submodule of the `autotrader.brokers` module. This submodule must contain two more submodules:

1. A core API module which communicates with the broker.
2. A utility module containing helper functions related to the core API module.

```
autotrader.brokers
├── broker_utils.py
└── broker_name
    ├── broker.py
    ├── __init__.py
    └── utils.py
```

3.10.10 Supported Brokers and Exchanges

Virtual Broker

`broker='virtual'`

At the heart of AutoTrader's backtesting algorithm is the virtual broker, a Python class intended to replicate the functionality of a real broker. See the documentation of the *Virtual Broker* for more information about how this functions.

Oanda v20 REST API

`broker='oanda'`

AutoTrader supports Oanda's v20 REST API. See the documentation of the *Oanda Broker* module for more information.

Interactive Brokers

`broker='ib'`

As of AutoTrader `v0.6.0`, *Interactive Brokers* is also supported.

DYDX Cryto Exchange

`broker='dydx'`

dYdX is a decentralised cryptocurrency derivatives exchange.

CCXT

`broker='ccxt:<exchange name>'`

The CryptoCurrency eXchange Trading (CCXT) library is an open-source Python library supporting over 100 cryptocurrency exchange markets and trading APIs.

3.10.11 API Reference

class `autotrader.brokers.broker.`**`AbstractBroker`**(*config: dict*, *utils: BrokerUtils | None = None*)

abstract **`cancel_order`**(*order_id: int*, **args*, ***kwargs*) → None

Cancels order by order ID.

abstract **`get_NAV`**(**args*, ***kwargs*) → float

Returns the net asset/liquidation value of the account.

abstract **`get_balance`**(**args*, ***kwargs*) → float

Returns account balance.

abstract **`get_orders`**(*instrument: str | None = None*, **args*, ***kwargs*) → dict

Returns all pending orders (have not been filled) in the account.

abstract **`get_positions`**(*instrument: str | None = None*, **args*, ***kwargs*) → dict

Gets the current positions open on the account.

Parameters
instrument (*str, optional*) – The trading instrument name (symbol). The default is None.

Returns
open_positions – A dictionary containing details of the open positions.

Return type
dict

abstract get_trades(*instrument: str | None = None*, **args*, ***kwargs*) → dict

Returns the trades (fills) made by the account.

abstract place_order(*order: Order*, **args*, ***kwargs*) → None

Translate order and place via exchange API.

3.11 AutoTrader Communications Module

3.11.1 Telegram

AutoTrader supports using a Telegram Bot for notifications. To make use of this, there are a few simple steps you must complete:

1. Create a new Telegram bot using the BotFather. Once you have gone through the prompts, you will be given a HTTP access token. Copy this token.
2. Run the code snippet below from within your *project directory*, passing in the token copied above from the BotFather as `telegram_token`. This will start running the bot on your computer.
3. If you haven't already, start a conversation with your newly created bot on Telegram: type and send the `/start` command. Your chat ID will be printed to the console running the bot. This chat ID will be used for trading notifications.
4. Send the following message to the bot: "write id". This will automatically write your Telegram details to your `keys.yaml` file.
5. Press `ctrl+c` to kill the bot. Now you are ready to use Telegram for trading notifications.

```python
from autotrader.comms import Telegram

# Instantiate the bot and run it to get chat ID
tb = Telegram(telegram_token)
tb.run_bot()
```

Keys

Activating Telegram notifications requires the following keys in your `keys.yaml` file. You can add them manually, or use the code snippet above to let the bot do it!

```yaml
TELEGRAM:
  api_key: < telegram api key >
  chat_id: < telegram chat ID >
```

Note that once you have added your details as shown below, the `Telegram` class can be instantiated without the API token; it will locate it in your `keys.yaml` file.

3.12 AutoTrader Custom Indicators

This page showcases the indicators available in AutoTraders' indicator library. All images shown here were created with *AutoPlot*, using the `view_indicators` function. This function can be called using the code snipped provided below, where `indicator_dict` is constructed for the indicator being plotted. This dictionary is shown for each indicator below. Note that the *indicators dictionary* passed to the `view_indicators` method must be formatted according to the correct *specification*.

```python
from autotrader import indicators
from autotrader.autoplot import AutoPlot

indicator_dict = {}

ap = autoplot.AutoPlot()
ap.data = data
ap.view_indicators(indicator_dict)
```

For each indicator below, the function definition is provided, along with a sample code snippet of how to plot the indicator with *AutoPlot*.

3.12.1 Indicators

Supertrend Indicator

`autotrader.indicators.`**`supertrend`**(*data: DataFrame*, *period: int = 10*, *ATR_multiplier: float = 3.0*, *source: Series | None = None*) → DataFrame

SuperTrend indicator, ported from the SuperTrend indicator by KivancOzbilgic on TradingView.

Parameters

- **data** (*pd.DataFrame*) – The OHLC data.
- **period** (*int, optional*) – The lookback period. The default is 10.
- **ATR_multiplier** (*int, optional*) – The ATR multiplier. The default is 3.0.
- **source** (*pd.Series, optional*) – The source series to use in calculations. If None, hl/2 will be used. The default is None.

Returns

supertrend_df – A Pandas DataFrame of containing the SuperTrend indicator, with columns of 'uptrend' and 'downtrend' containing uptrend/downtrend support/resistance levels, and 'trend', containing -1/1 to indicate the current implied trend.

Return type

pd.DataFrame

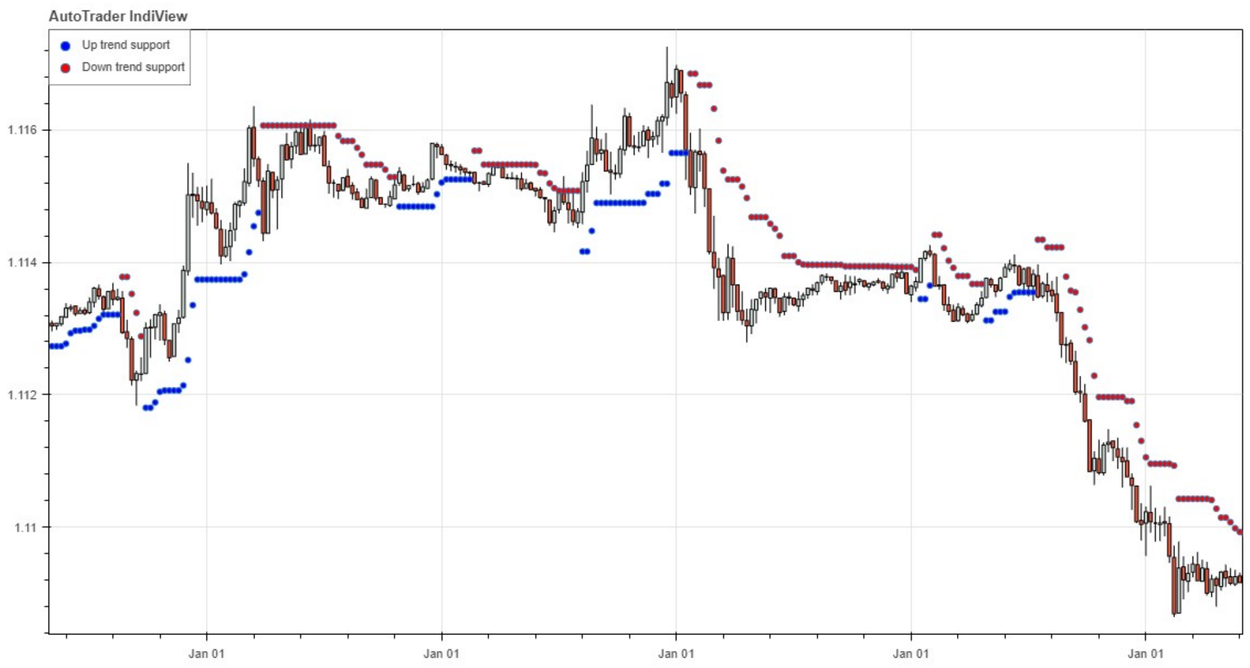
References

https://www.tradingview.com/script/r6dAP7yi/

```
st_df = indicators.supertrend(data, ATR_multiplier=2)

indicator_dict = {'Supertrend': {'type': 'Supertrend',
                                 'data': st_df}
                 }
```

Note that the supertrend dataframe also contains a trend column, indicating the current trend.

Column	Description
uptrend	Uptrend price support level
downtrend	Downtrend price support level
trend	Current trend (1 for uptrend, -1 for downtrend)



HalfTrend Indicator

The HalfTrend indicator is based on the indicator by *everget* on TradingView.

`autotrader.indicators.`**`halftrend`**(*data: DataFrame*, *amplitude: int = 2*, *channel_deviation: float = 2*) → DataFrame

HalfTrend indicator, ported from the HalfTrend indicator by Alex Orekhov (everget) on TradingView.

Parameters

- **data** (*pd.DataFrame*) – OHLC price data.
- **amplitude** (*int, optional*) – The lookback window. The default is 2.

- **channel_deviation** (*float, optional*) – The ATR channel deviation factor. The default is 2.

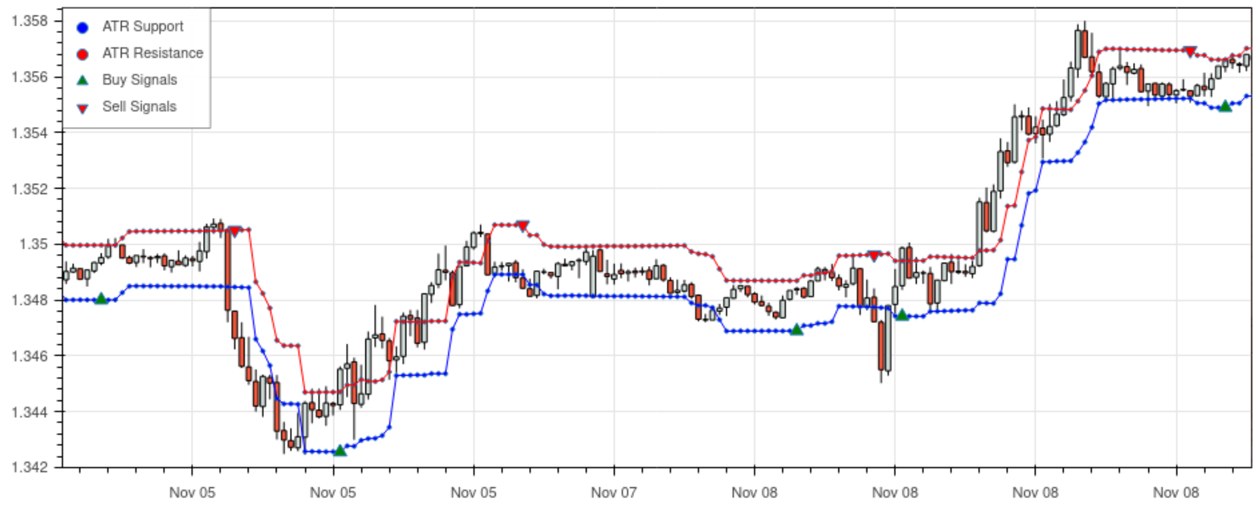
Returns
htdf – DESCRIPTION.

Return type
TYPE

References

https://www.tradingview.com/script/U1SJ8ubc-HalfTrend/

```
halftrend_df = indicators.halftrend(data)
indicator_dict = {'HalfTrend': {'type': 'HalfTrend',
                                'data': halftrend_df}}
```

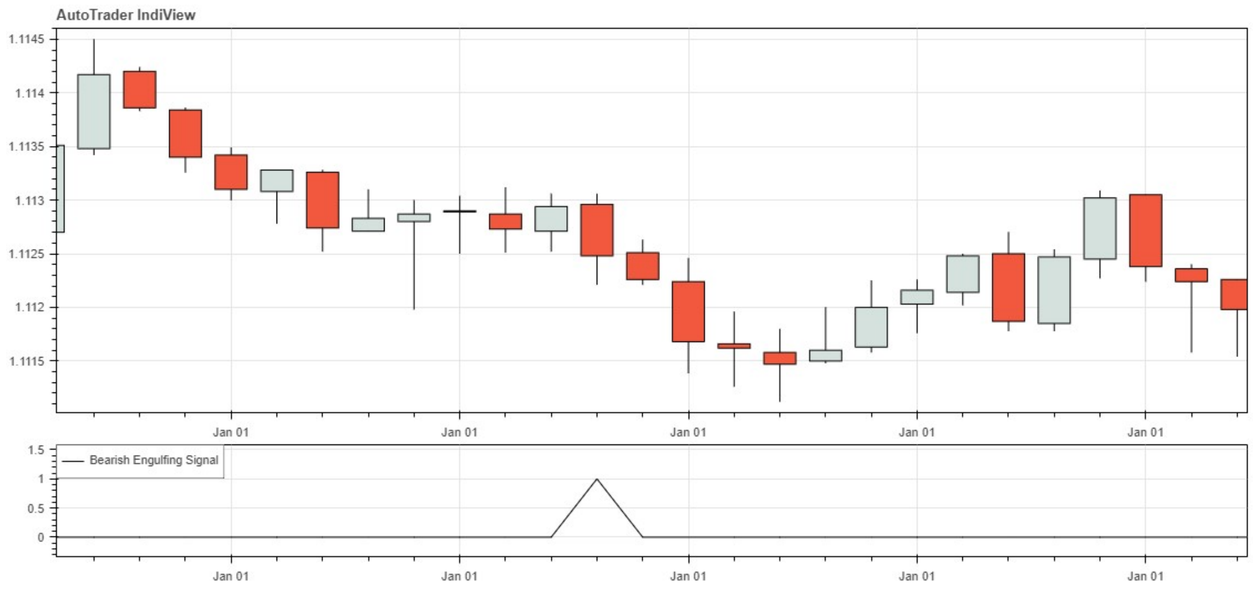


Bearish Engulfing Pattern

Returns a list with values of 1 when the bearish engulfing pattern appears and a value of 0 elsewhere.

autotrader.indicators.**bearish_engulfing**(*data: DataFrame*, *detection: str | None = None*)
Bearish engulfing pattern detection.

```
engulfing_bearish = indicators.bearish_engulfing(data, detection = None)
indicator_dict = {'Bearish Engulfing Signal': {'type': 'Engulfing',
                                               'data': engulfing_bearish}
                 }
```



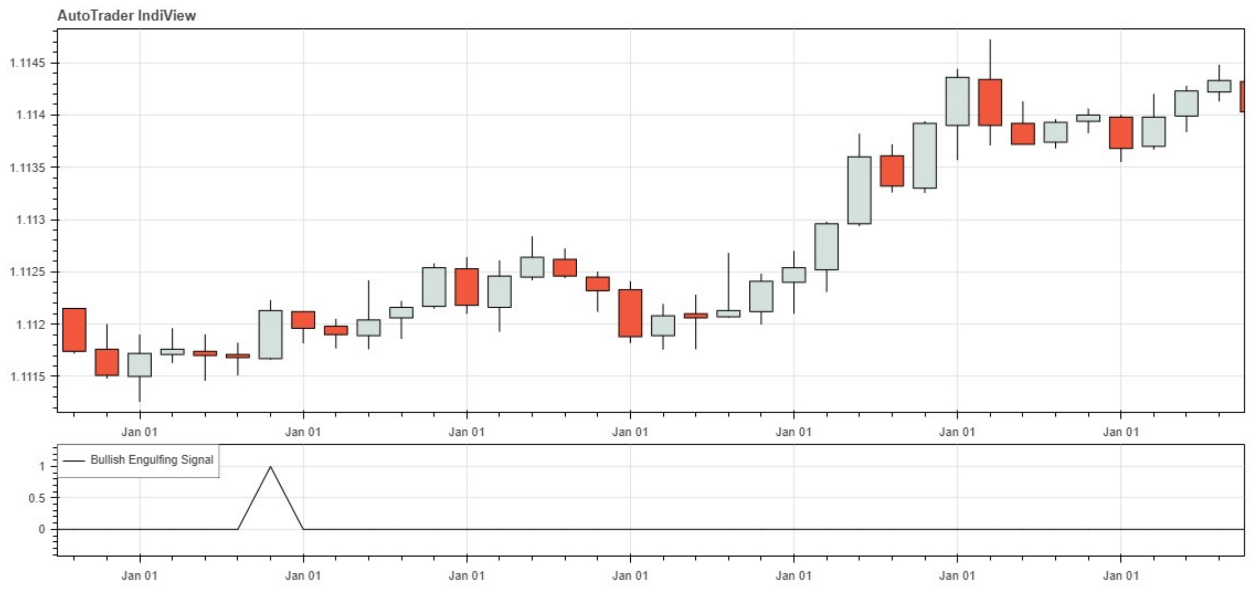
Bullish Engulfing Pattern

Returns a list with values of 1 when the bullish engulfing pattern appears and a value of `0` elsewhere.

autotrader.indicators.**bullish_engulfing**(*data: DataFrame*, *detection: str | None = None*)

Bullish engulfing pattern detection.

```
engulfing_bullish = indicators.bullish_engulfing(data, detection = None)
indicator_dict = {'Bullish Engulfing Signal': {'type': 'Engulfing',
                                               'data': engulfing_bullish}
                 }
```



Heikin-Ashi Candlesticks

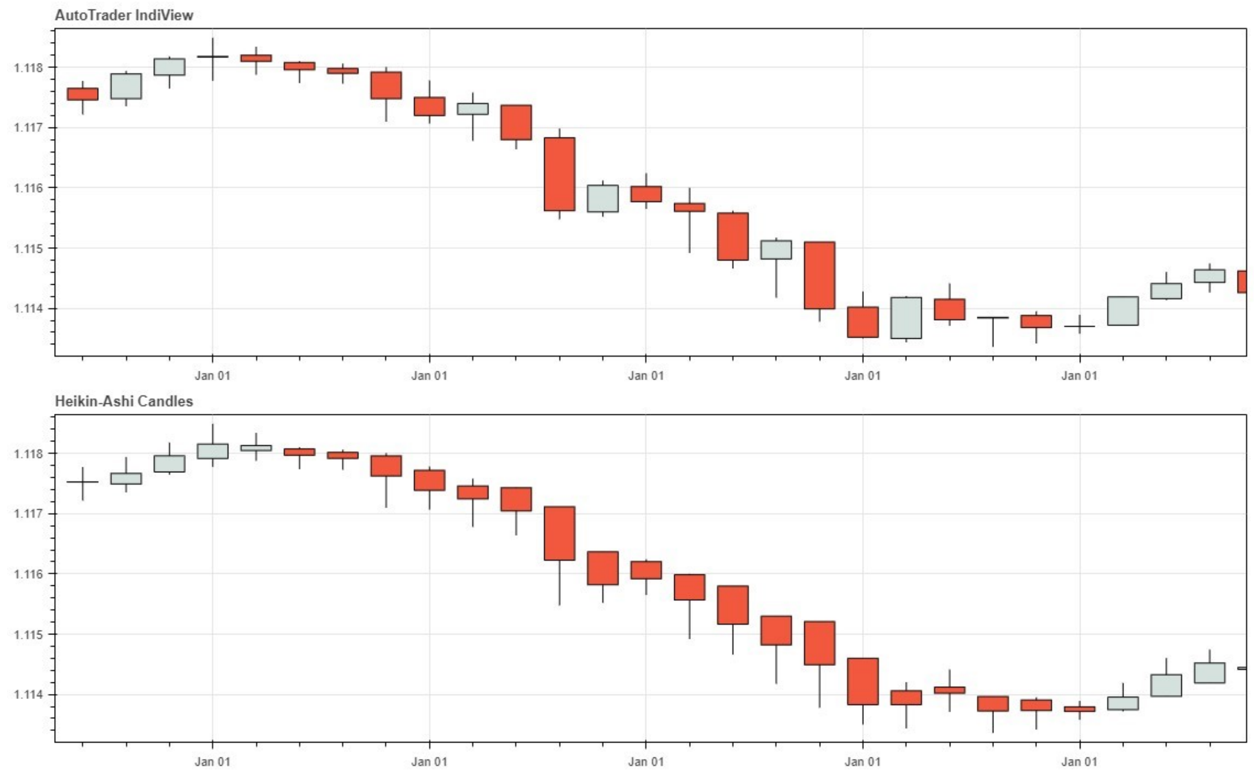
Returns a dataframe of Heikin-Ashi candlesticks.

`autotrader.indicators.`**`heikin_ashi`**`(`*data: DataFrame*`)`

Calculates the Heikin-Ashi candlesticks from Japanese candlestick data.

```
ha_data = indicators.heikin_ashi(OHLC_data)

indicator_dict = {'Heikin-Ashi Candles': {'type': 'Heikin-Ashi',
                                          'data': ha_data}
                 }
```

Note that a copy of the data must be passed into the `heikin-ashi` function by using the `.copy()` function, to prevent overwriting the original `data`. See here for more information.



Divergence

The `autodetect_divergence` indicator can be used to detect divergence between price movements and and an indicator.

`autotrader.indicators.`**`autodetect_divergence`**`(`*ohlc: DataFrame*, *indicator_data: DataFrame*, *tolerance: int = 1*, *method: int = 0*`)` → DataFrame

A wrapper method to automatically detect divergence from inputted OHLC price data and indicator data.

Parameters

- **`ohlc`** (*`pd.DataFrame`*) – A dataframe of OHLC price data.
- **`indicator_data`** (*`pd.DataFrame`*) – dataframe of indicator data.

- **tolerance** (*int, optional*) – A parameter to control the lookback when detecting divergence. The default is 1.
- **method** (*int, optional*) – The divergence detection method. Set to 0 to use both price and indicator swings to detect divergence. Set to 1 to use only indicator swings to detect divergence. The default is 0.

Returns
divergence – A DataFrame containing columns 'regularBull', 'regularBear', 'hiddenBull' and 'hiddenBear'.

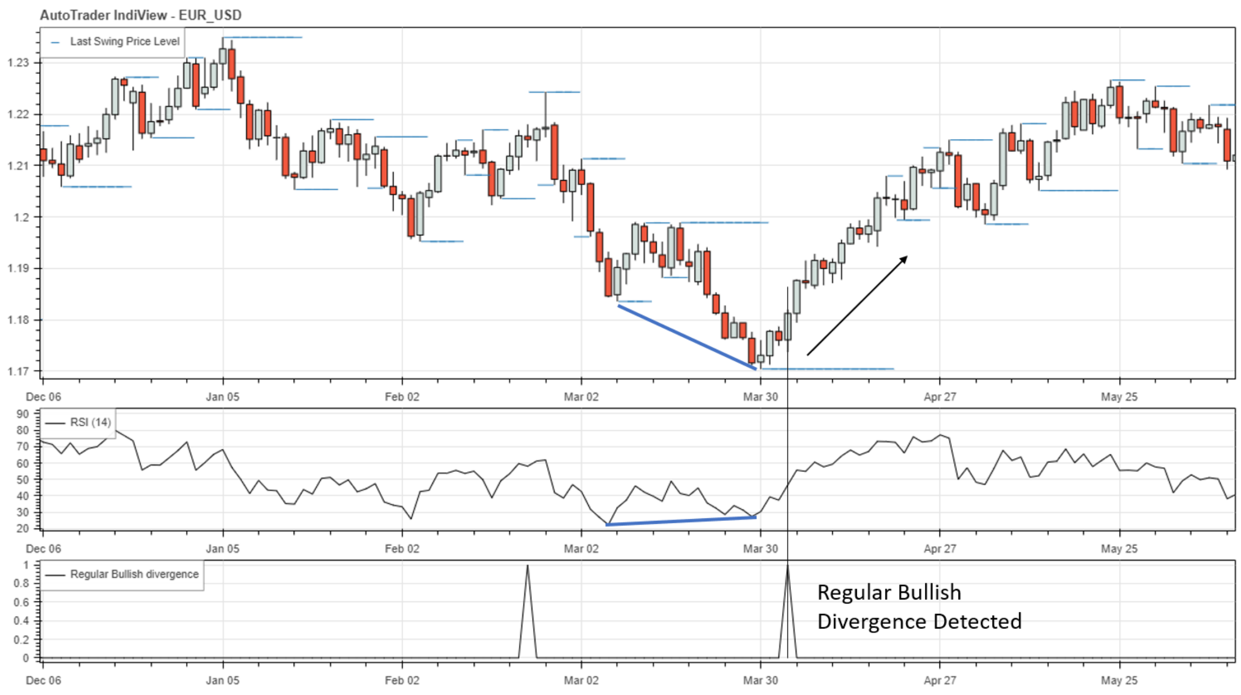
Return type
pd.DataFrame

See also:

autotrader.indicators.find_swings, *autotrader.indicators.classify_swings*, *autotrader.indicators.detect_divergence*

```
rsi_divergence = indicators.autodetect_divergence(data, rsi)

indicator_dict = {'RSI (14)': {'type': 'RSI',
                               'data': rsi},
                  'Bullish divergence': {'type': 'below',
                                         'data': rsi_divergence['regularBull']},
                  }
```

Below is an example of this indicator, as detailed in the Detecting Divergence blog post. Note that this indicator is actually a wrapper for other indicators, to make detecting divergence even simpler.



See Also

find_swings, *classify_swings* and *detect_divergence*

3.12.2 Utility Indicators

The following is a collection of utility indicators which assist in building effective strategies.

Swing Detection

A common challenge of algo-trading is the ability to accurately pick recent swings in price to use as stop loss levels. This indicator attempts to solve that problem by locating the recent swings in price. This indicator returns a dataframe with three columns: Highs, Lows and Last, described below.

`autotrader.indicators.`**`find_swings`**`(`*data: DataFrame*, *n: int = 2*`)` → DataFrame

Locates swings in the inputted data using a moving average gradient method.

Parameters

- **`data`** (*`pd.DataFrame | pd.Series | list | np.array`*) – An OHLC dataframe of price, or an array/list/Series of data from an indicator (eg. RSI).
- **`n`** (*`int, optional`*) – The moving average period. The default is 2.

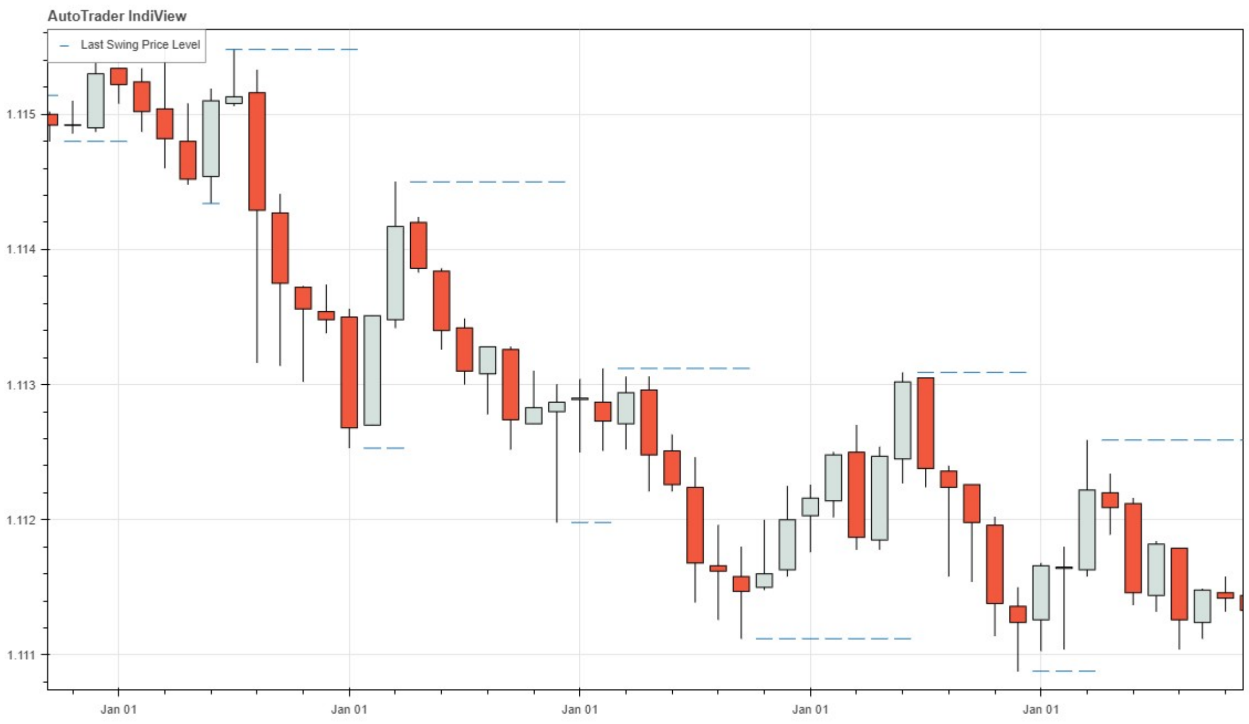
Returns

- **swing_df** (*pd.DataFrame*) – A dataframe containing the swing levels detected.
- *pd.Series(hl2, name="hl2"),*

Column	Description
Highs	Most recent swing high
Lows	Most recent swing low
Last	Most recent swing

```
swings         = indicators.find_swings(data)
indicator_dict = {'Swing Detection': {'type': 'Swings',
                                      'data': swings}
                  }
```

To detect swings, an exponential moving average is fitted to the inputted data. The slope of this line is then used determine when a swing has been formed. Naturally, this is a lagging indicator. The lag can be controlled by the input parameter `n`, which corresponds to the period of the EMA.



This indicator was used in the MACD strategy developed in the *tutorials* to set stop-losses.

Classifying Swings

The `classify_swings` indicator may be used to detect 'higher highs' and 'lower lows'. This indicator of course also detects 'lower highs' and 'higher lows'. It relies upon the output of the `find_swings` indicator. It returns a dataframe with the following columns appended to the swing dataframe.

Column Name	Description	Values
`HH`	Higher High	`True` or `False`
`HL`	Higher Low	`True` or `False`
`LL`	Lower Low	`True` or `False`
`LH`	Lower High	`True` or `False`

`autotrader.indicators.`**`classify_swings`**(*swing_df: DataFrame*, *tol: int = 0*) → DataFrame

Classifies a dataframe of swings (from find_swings) into higher-highs, lower-highs, higher-lows and lower-lows.

Parameters

- **`swing_df`** (*`pd.DataFrame`*) – The dataframe returned by find_swings.
- **`tol`** (*`int, optional`*) – The classification tolerance. The default is 0.

Returns

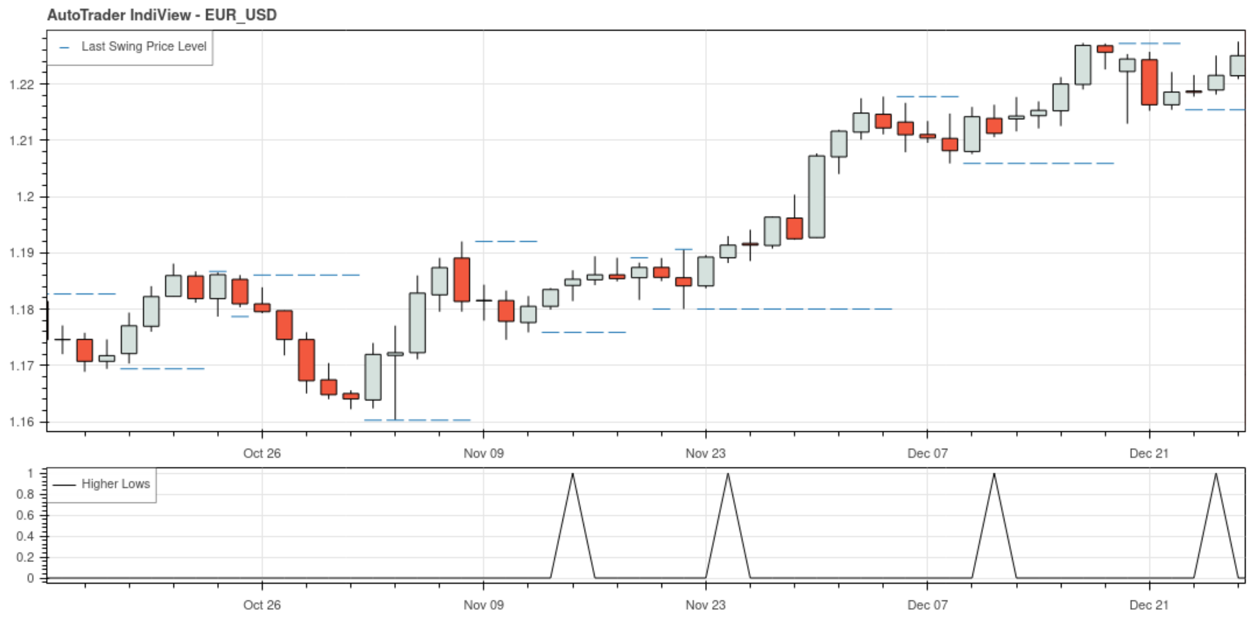
swing_df – A dataframe containing the classified swings.

Return type

pd.DataFrame

```
price_swings = indicators.find_swings(price_data)
price_swings_classified = indicators.classify_swings(price_swings)

indicator_dict = {'Price Swings': {'type': 'Swings',
                                   'data': price_swings},
                  'Higher Lows': {'type': 'below',
                                  'data': price_swings_classified['HL']},
                  }
```

The plot below gives an example of this indicator detecting higher lows in price.



See Also

find_swings

Detecting Divergence

To detect divergence between price and an indicator, the `detect_divergence` indicator may be used. This indicator relies on both `find_swings` and `classify_swings`. It detects regular and hidden divergence.

`autotrader.indicators.`**`detect_divergence`**(*classified_price_swings: DataFrame*, *classified_indicator_swings: DataFrame*, *tol: int = 2*, *method: int = 0*) → DataFrame

Detects divergence between price swings and swings in an indicator.

Parameters

- **`classified_price_swings`** (*pd.DataFrame*) – The output from classify_swings using OHLC data.
- **`classified_indicator_swings`** (*pd.DataFrame*) – The output from classify_swings using indicator data.

- **tol** (*int, optional*) – The number of candles which conditions must be met within. The default is 2.
- **method** (*int, optional*) – The method to use when detecting divergence (0 or 1). The default is 0.

Raises
`Exception` – When an unrecognised method of divergence detection is requested.

Returns
divergence – A dataframe containing divergence signals.

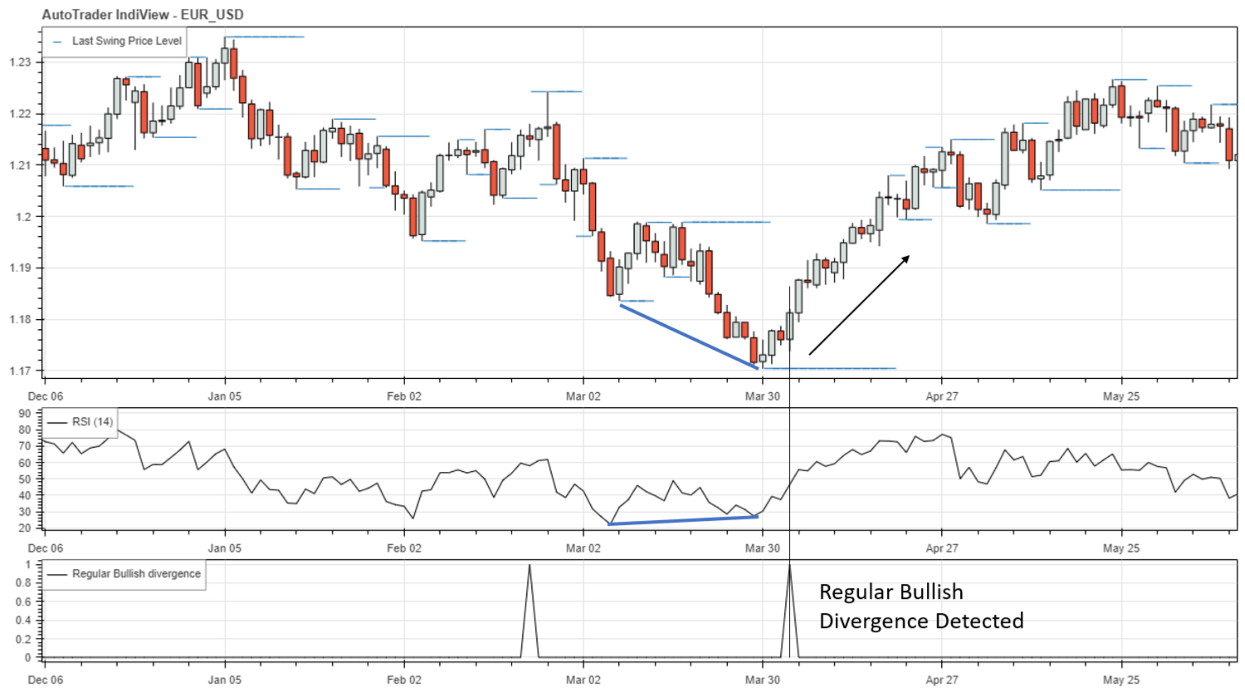
Return type
pd.DataFrame

Notes

Options for the method include:
0: use both price and indicator swings to detect divergence (default)

1: use only indicator swings to detect divergence (more responsive)

The example below shows the indicator detecting regular bullish divergence in price using the RSI as the indicator.



See Also

find_swings, *classify_swings* and *autodetect_divergence*

Crossover

Returns a list with values of 1 when input `list_1` crosses **above** input `list_2`, values of -1 when input `list_1` crosses **below** input `list_2`, and values of `0` elsewhere.

`autotrader.indicators.`**`crossover`**(*ts1: Series*, *ts2: Series*) → Series

Locates where two timeseries crossover each other, returning 1 when list_1 crosses above list_2, and -1 for when list_1 crosses below list_2.

Parameters

- **ts1** (*pd.Series*) – The first timeseries.
- **ts2** (*pd.Series*) – The second timeseries.

Returns

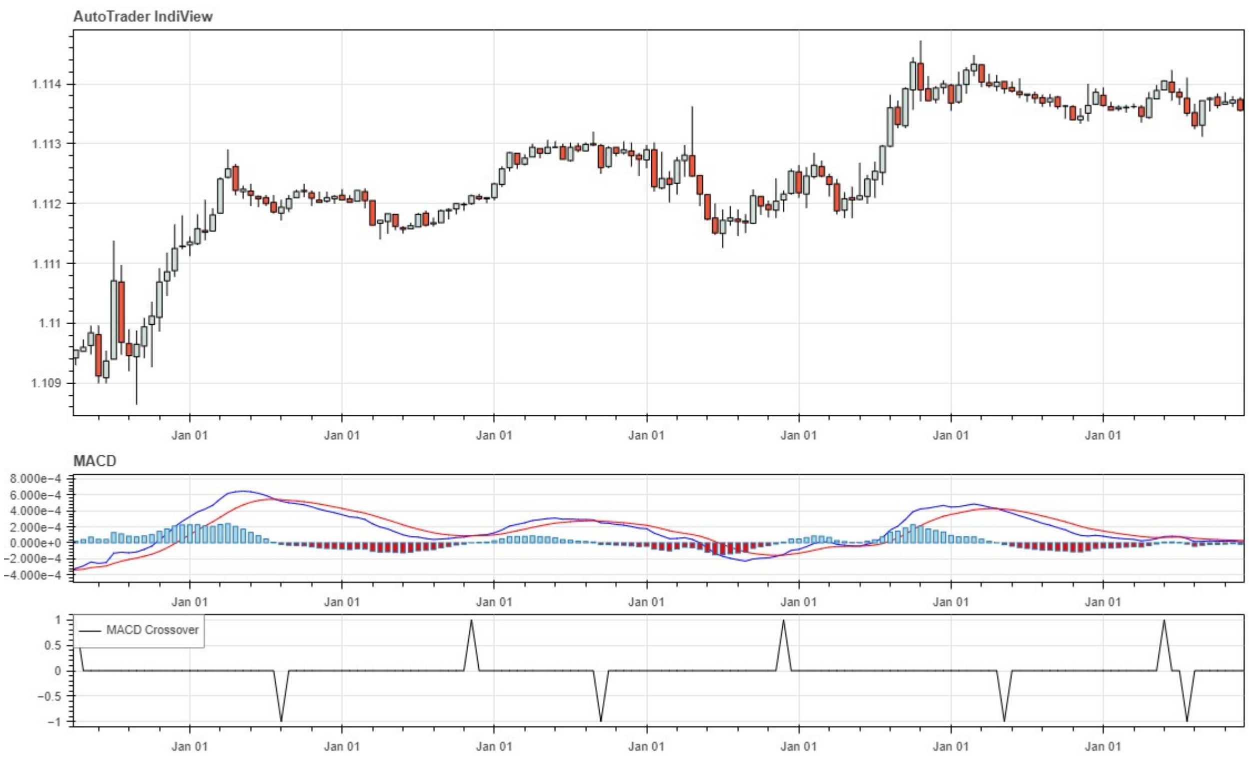
crossovers – The crossover series.

Return type

pd.Series

The example below illustrates the functionality of this indicator with the MACD indicator. Note that the MACD line is passed into `indicators.crossover` as `list_1`, and the MACD signal line as `list_2`. This ensures that a value of 1 will correspond to points when the MACD line crosses above the signal line, and a value of -1 when it crosses below the signal line.

```
macd, macd_signal, macd_hist = ta.MACD(data.Close.values)
macd_crossover = indicators.crossover(macd, macd_signal)

indicator_dict = {'MACD': {'type': 'MACD',
                           'macd': macd,
                           'signal': macd_signal,
                           'histogram': macd_hist},
                  'MACD Crossover': {'type': 'Crossover',
                                     'data': macd_crossover}
                  }
```



Cross Value

Returns the value at which a crossover occurs using linear interpolation. Requires three inputs: two lists and a third list corresponding to the points in time which the two lists crossover. Consider the example described below.

`autotrader.indicators.`**`cross_values`**(*ts1: list* | *Series*, *ts2: list* | *Series*, *ts_crossover: list* | *Series* | *None* = *None*) → list | Series

Returns the approximate value of the point where the two series cross.

Parameters

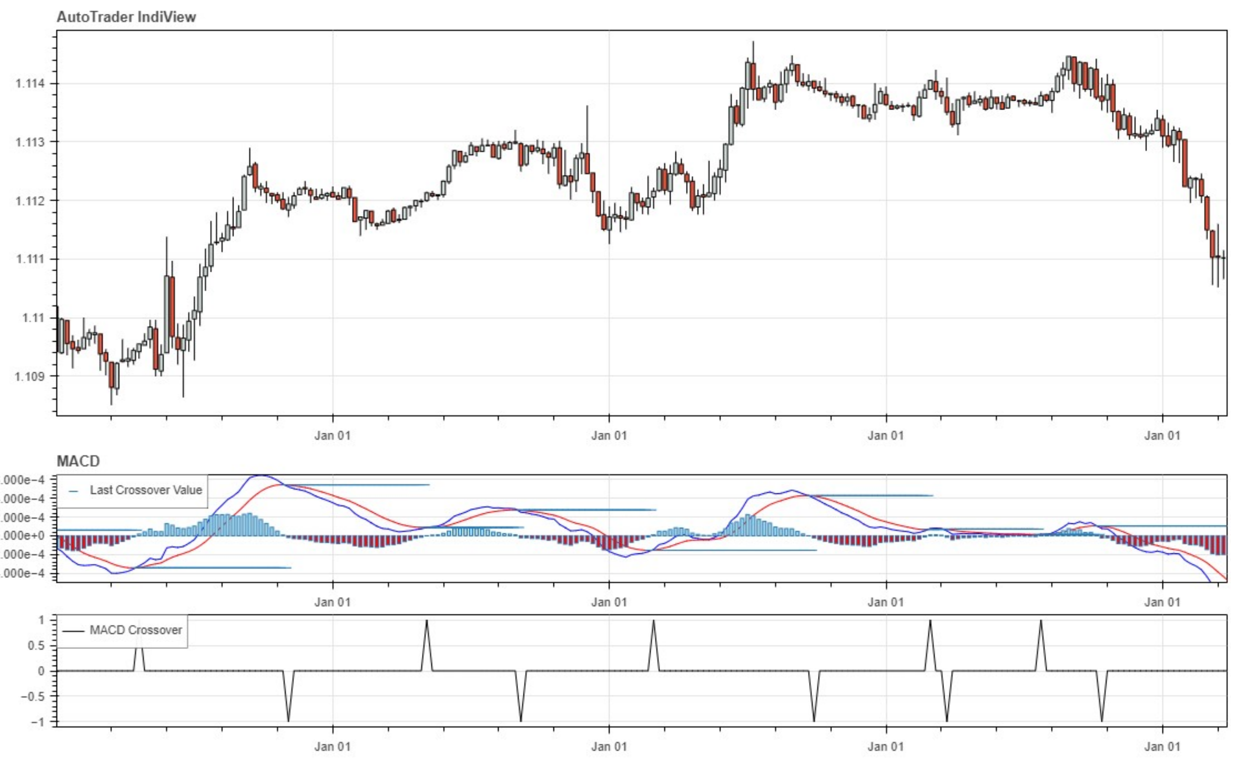
- **`ts1`** (*`list | pd.Series`*) – The first timeseries..
- **`ts2`** (*`list | pd.Series`*) – The second timeseries..
- **`ts_crossover`** (*`list | pd.Series, optional`*) – The crossovers between timeseries 1 and timeseries 2.

Returns
cross_points – The values at which crossovers occur.

Return type
list | pd.Series

The example provided below builds upon the example described for the *crossover* indicator. Again, the MACD indicator is used, and MACD/signal line crossovers are found using `indicators.crossover`. The specific values at which this crossover occurs can then be calculated using `indicators.cross_values(macd, macd_signal, macd_crossover)`. This will return a list containing the values (in MACD y-axis units) where the crossover occured. This is shown in the image below, labelled 'Last Crossover Value'. This indicator is useful in strategies where a crossover must occur above or below a specified threshold.

```
macd, macd_signal, macd_hist = ta.MACD(data.Close.values)
macd_crossover = indicators.crossover(macd, macd_signal)
macd_covals = indicators.cross_values(macd, macd_signal, macd_crossover)

indicator_dict = {'MACD': {'type': 'MACD',
                           'macd': macd,
                           'signal': macd_signal,
                           'histogram': macd_hist,
                           'crossvals': macd_covals},
                  'MACD Crossover': {'type': 'Crossover',
                                     'data': macd_crossover}
                  }
```



Candles Between Crosses

Returns a list with a count of how many candles have passed since the last crossover (that is, how many elements in a list since the last non-zero value).

`autotrader.indicators.`**`candles_between_crosses`**(*crosses: list* | *Series*, *initial_count: int = 0*) → list | Series

Returns a rolling sum of candles since the last cross/signal occurred.

Parameters

crosses (*list | pd.Series*) – The list or Series containing crossover signals.

Returns

counts – The rolling count of bars since the last crossover signal.

Return type

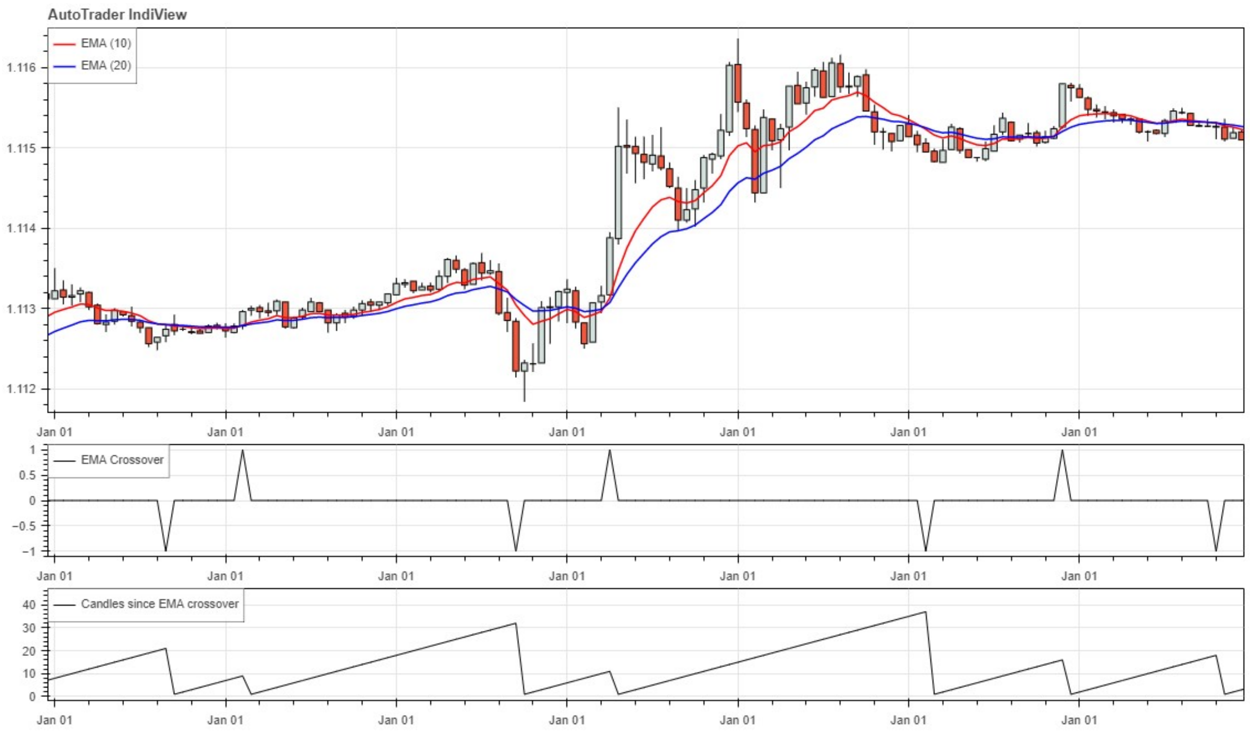
TYPE

See also:

`indicators.crossover`

The example provided below demonstrates this indicator with EMA crossovers.

```python
ema10           = ta.EMA(data.Close.values, 10)
ema20           = ta.EMA(data.Close.values, 20)
ema_crossovers  = indicators.crossover(ema10, ema20)
ema_crosscount = indicators.candles_between_crosses(ema_crossovers)

indicator_dict = {'EMA (10)': {'type': 'MA',
                             'data': ema10},
                  'EMA (20)': {'type': 'MA',
                             'data': ema20},
                  'EMA Crossover': {'type': 'Crossover',
                                  'data': ema_crossovers},
                  'Candles since EMA crossover': {'type': 'Crosscount',
                                                'data': ema_crosscount}
                  }
```



Heikin-Ashi Candlestick Run

`autotrader.indicators.`**`ha_candle_run`**(*ha_data: DataFrame*)

Returns a list for the number of consecutive green and red Heikin-Ashi candles.

Parameters

`ha_data` (*`pd.DataFrame`*) – The Heikin Ashi OHLC data.

See also:

`heikin_ashi`

This indicator returns two lists; one each for the number of consecutive green and red Heikin-Ashi candles. Since Heikin-Ashi trends usually last for approximately 5-8 candles, it is useful to know how many consecutive red or green candles there have been so far, to avoid getting into a trend too late. This indicator allows you to prevent that by telling you how many candles into a trend the price action is.

Merge signals

Returns a single signal list which has merged two signal lists.

`autotrader.indicators.`**`merge_signals`**(*signal_1: list*, *signal_2: list*) → list

Returns a single signal list which has merged two signal lists.

Parameters

- **`signal_1`** (*`list`*) – The first signal list.
- **`signal_2`** (*`list`*) – The second signal list.

Returns

merged_signal_list – The merged result of the two inputted signal series.

Return type

list

Examples

```
>>> s1 = [1,0,0,0,1,0]
>>> s2 = [0,0,-1,0,0,-1]
>>> merge_signals(s1, s2)
    [1, 0, -1, 0, 1, -1]
```

Rolling Signal

Returns a list which maintains the previous signal, until a new signal is given.

`autotrader.indicators.`**`rolling_signal_list`**(*signals: list* | *Series*) → list

Returns a list which repeats the previous signal, until a new signal is given.

Parameters

`signals` (*`list | pd.Series`*) – A series of signals. Zero values are treated as ‘no signal’.

Returns

A list of rolled signals.

Return type

list

Examples

```
>>> rolling_signal_list([0,1,0,0,0,-1,0,0,1,0,0])
    [0, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1]
```

Unroll Signal List

Performs the reverse function of *rolling*.

autotrader.indicators.**unroll_signal_list**(*signals: list | Series*) → array

Unrolls a rolled signal list.

Parameters
signals (*Union[list, pd.Series]*) – DESCRIPTION.

Returns
unrolled_signals – The unrolled signal series.

Return type
np.array

See also:

`This`

Examples

```
>>> unroll_signal_list([0, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1])
    array([ 0.,  1.,  0.,  0.,  0., -1.,  0.,  0.,  1.,  0.,  0.])
```

3.13 Command Line Interface

As of `v0.7.0`, AutoTrader features a Command Line Interface (CLI) for various tasks.

```
Usage: autotrader [OPTIONS] COMMAND [ARGS]...

  AutoTrader command line interface.

Options:
  --help  Show this message and exit.

Commands:
  demo      Runs a demo backtest in AutoTrader.
  init      Initialises the directory NAME for trading with AutoTrader.
  monitor   Monitors a broker and serves the information to a prometheus...
  snapshot  Prints a snapshot of the trading account of a broker PICKLE...
  version   Shows the installed version number of AutoTrader.
```

3.13.1 Installed Version Number

To quickly check what version of AutoTrader you have installed, you can use:

```
autotrader version
```

Note that you can also get this information from Python using the snippet below.

```
import autotrader

print(autotrader.__version__)
```

3.13.2 Demo Backtest

```
Usage: autotrader demo [OPTIONS]

  Runs a demo backtest in AutoTrader.
```

3.13.3 Directory Initialisation

To quickly initialise your directory to the *recommended structure*, you can use `autotrader init`.

```
Usage: autotrader init [OPTIONS] [NAME]

  Initialises the directory NAME for trading with AutoTrader. If no directory
  NAME is provided, the current directory will be initialised.

  To include ready-to-go strategies in the initialised directory, specify them
  using the strategies option. You can provide the following arguments:

  - template: a strategy template module
  - config: a strategy configuration template
  - strategy_name: the name of a strategy to load

  Strategies are loaded from the AutoTrader demo repository here:
  https://github.com/kieran-mackle/autotrader-demo

Options:
  -s, --strategies TEXT  The name of strategies to include in the initialised
                         directory.
  --help                 Show this message and exit.
```

3.13.4 Trading Monitor

Tip: See the tutorial on *setting up your own trade dashboard* for more information.

```
Usage: autotrader monitor [OPTIONS]

  Monitors a broker/exchange and serves the information to a prometheus
  database.

Options:
  -p, --port INTEGER      The port to serve data to.
  -i, --initial-nav TEXT  The initial NAV to use for relative PnL
                          calculations.
  -m, --max-nav TEXT      The maximum NAV to use for drawdown calculations.
  -f, --picklefile TEXT   The pickle file containing a virtual broker
                          instance.
  -c, --config TEXT       The monitor yaml configuration filepath.
  -b, --broker TEXT       The name of the broker to connect to.
  -e, --environment TEXT  The trading environment.
  --help                  Show this message and exit.
```

3.14 AutoTrader Changelog

3.14.1 v0.12.1 (2023-09-26)

Fix

- **autoplot.py**: fix plot_data error for indicator scaling

3.14.2 v0.12.0 (2022-11-17)

Feat

- **indicators**: added signal column to chandelier indicator
- **indicators**: added chandelier exit indicator
- **cli**: added option to initialise directory from demo repository

Fix

- **indicators.py**: fill na values in indicator data for finding swings
- **autoplot.py**: skip autoscaling when there is no data variation
- **indicators.py**: fixed outdated argument for autodecting divergence

3.14.3 v0.11.2 (2022-10-27)

Fix

- **AutoData**: do not truncate yahoo data if start and end arguments are not None

3.14.4 v0.11.1 (2022-10-27)

Refactor

- **AutoData**: optionally provide workers as kwarg to fetch

3.14.5 v0.11.0 (2022-10-27)

Feat

- **AutoData**: batch fetch instruments

3.14.6 v0.10.1 (2022-10-26)

Perf

- data fetch of yahoo finance with count extends range to account for business days

3.14.7 v0.10.0 (2022-10-25)

Feat

- improved monitor and added dashboard template
- **cli.py**: added backtest demo function to cli

Fix

- **autotrader.py**: write broker hist when click paper trading
- **autodata.py**: raise exception when invalid granularity is provided to yahoo finance

Refactor

- added exception handling for click papertrade
- **autodata.py**: added exception handling for ccxt orderbook method
- moved print_banner function to utilities
- **cli.py**: init method more robust
- **macd_strategy.py**: load data from yahoo finance
- **macd_strategy.py**: changed data directory path to cwd
- deprecated support for email notifications

3.14.8 v0.9.1 (2022-10-23)

Fix

- **indicators.py**: fixed handling of different data types in find_swings indicator

3.14.9 v0.9.0 (2022-10-21)

Feat

- reimplemented scan mode
- integrated Telegram for trade notifications (#12)
- **tg.py**: telegram bot can write chat id to keys.yaml
- **telegram.py**: telegram bot returns chat id for initialisation
- added initialisation method

Fix

- fixed circular import errors
- **tg.py**: order side determination logic

Refactor

- **tg.py**: renamed telegram.py to tg.py to avoid name conflict
- **notifier.py**: added abstract communications class Notifier

3.14.10 v0.8.2 (2022-10-19)

Refactor

- **Broker**: all broker class inherit from AbstractBroker
- **brokers.broker.py**: renamed Broker to AbstractBroker
- **broker.py**: implemented initial broker abstraction

3.14.11 v0.8.1 (2022-10-19)

Refactor

- **ccxt.broker.py**: added network exception handling with single retries

3.14.12 v0.8.0 (2022-10-17)

Feat

- **autoplot.py**: portfolio plot includes equity and nav hovertool

3.14.13 v0.7.11 (2022-10-17)

Fix

- email_manager import (#46)
- datetime.timezone import
- CCXT get_trades uses kwargs in fetchMyTrades call

3.14.14 Version 0.7.10

Changes

- Improved verbosity for exception handling.
- Improved verbosity in `autotrader.py` for bot updates.
- Added utility to CCXT interface (`get_min_notional` and `get_ticksize` methods).
- Improved CCXT `get_orders` capability.

Fixes

- CCXT interface `get_trades` method updated for `Trade` object arguments.

3.14.15 Version 0.7.9

Fixes

- Plotting bug when option to show cancelled orders is True.

3.14.16 Version 0.7.8

Features

- Upgraded virtual broker: backtest speedup for large portfolio's
- Ability to specify `deploy_time` in `AutoTrader.configure()`, a datetime object for when to release trading bots.
- Improved verbosity from main module when running.

3.14.17 Version 0.7.7

Fixes

- Decimal error when placing market orders with `dydx` module.

3.14.18 Version 0.7.6

Fixes

- Import error of `AutoData` in `dydx` module.

3.14.19 Version 0.7.5

Features

- AutoBot submits orders using `ThreadPoolExecutor` to speedup submission of multiple orders.
- Ability to provide custom execution methods via `AutoTrader.configure(execution_method=)`.
- Improved verbosity from `autobots`.

Fixes

- Handling of testnet/mainnet keys when paper/virtual/live trading.
- Inclusion of `__init__.py` file in `autotrader/brokers/ccxt/`.
- Timezone handling.
- Virtual broker does not use lambda functions to allow pickling.
- Unified `broker._utils` attribute naming.

3.14.20 Version 0.7.4

Features

- Better exception handling in CCXT broker interface.
- Ability to specify `mainnet` and `testnet` API keys in your `keys.yaml` file.
- Ability to provide slippage models for backtests (via `at.configure()`).

Fixes

- Inifite `while` loop bug in virtual broker `_reduce_position` method due to machine precision.
- Backtest portfolio plotting of more than 18 instruments is possible now due to an increased color pallete.

3.14.21 Version 0.7.3

Fixes

- Unification of `get_orderbook` in supporting `broker` modules.
- Expected behaviour of `get_positions` method in CCXT broker module.

Features

- Trading object `Position` includes attribute `ccxt` to include the output from `CCXT` methods.
- Improved configuration options for CCXT exchanges in `keys.yaml` file.

3.14.22 Version 0.7.2

Fixes

- Oanda live trade functionality restored (after `keys.yaml` rename).

Features

- AutoData is more intelligent when creating a new instance; kwargs can be used in place of data_config dictionary, simplifying instantiation.
- Utility methods get_broker_config and get_data_config have been simplified, allowing calling without global_config argument (keys.yaml will be read in from config/ directory).

3.14.23 Version 0.7.1

Changes

- Oanda configuration keys in keys.yaml have changed for clarification

Fixes

- Oanda data_config includes account id, restoring automated data retrieval

Features

- Improved portfolio plot type
- Improved printouts

3.14.24 Version 0.7.0

AUGUST 2022

Breaking Changes

- Backtest spread is now specified in absolute price units (rather than pips as previously)
- Environment specification: paper trading can be activated by setting environment to paper (default) and live trading can be activated by setting environment to live
- To further remove the distinction between backtesting and livetrading, various methods and attributes have been renamed to reflect their generality and indifference to mode of trading. Important changes include AutoTrader.backtest_results to AutoTrader.trade_results (and similar for AutoBot), AutoTrader.print_backtest_results to AutoTrader.print_trade_results and BacktestResults class to TradeAnalysis. Renaming generally followed the pattern of renaming *backtest* to *trade*.
- For consistency in naming conventions, GetData class of autodata.py has been renamed to AutoData.
- Broker interface method get_positions will directly
- Rename virtual_livetrade_config to virtual_account_config.
- Strategy configuration key INCLUDE_POSITIONS has been deprecated in favour of using INCLUDE_BROKER, then directly fetching positions from broker using get_positions method.
- Renamed GLOBAL.yaml to keys.yaml for clarification.
- Run mode ‘continuous’ has become the default run mode. To continue running strategies in periodic update mode, you will now need to specify mode='periodic' in configure.

- The behaviour of broker method `get_trades` has changed: now returns a list of fills (executed trades based on the `Trade` object), rather than a dictionary of `IsolatedPositions` objects as before. This falls in line with the more common definition of a trade, but diverges from Oanda. As such, a new method `get_isolated_positions` has been added to the virtual broker and Oanda API interface to maintain the previous functionality of `get_trades`.

Features

- Major backtest speed improvements: over 50% reduction in backtest time for large, multi-asset backtests
- Live paper-trading via the virtual broker: use `AutoTrader.virtual_livetrade_config` to configure virtual broker.
- To check-in on paper trading status, there is a new convenience method `papertrade_snapshot`, which will print up-to-date trade results from the virtual broker pickled instance.
- Support for decentralised crypto exchange dYdX
- Support for many more crypto exchanges via CCXT
- Introduction of 'portfolio' strategies: passing data of multiple assets to a single strategy. Simply include `PORTFOLIO: True` in your strategy configuration.
- Data feeds have been unified to make data retrieval simpler than ever. Now there are methods `fetch` and `quote`, which can be used to fetch OHLC price data from various feeds, depending on the `data_source` specified in the data configuration dictionary. Retrieval of level 1 and level 2 data is also available (where possible), accessible via the `L1` and `L2` methods.
- Improved backtest accuracy, with orderbook simulation and order type dependent commissions.
- Additional commission schemes for backtesting.
- Option to specify bid/ask spread as a percentage value.
- Manual trading (paper and live) via command line. Simply configure an instance of AutoTrader without adding a strategy, and the broker specified will be instantiated ready for trading. Papertrading via the virtual broker supported.
- Ability to trade across multiple venues from a single strategy. Simply provide the broker names with comma separation via the `configure` method,
- Exchange-specific precision checking for Orders. Even in backtest mode, AutoTrader will communicate with your chosen exchange to precision-check your orders.
- Code is now formatted using Black.
- Ability to specify a time range for `PERIOD` in strategy configuration. This value will be converted to an integer using the `INTERVAL` key.

Deprecation Notices

- Broker method `get_trade_details` has been deprecated in favour of `get_trades` method.
- Strategy configuration key `INCLUDE_POSITIONS` has been deprecated in favour of using `INCLUDE_BROKER`, then directly fetching positions from broker using `get_positions` method.

Fixes

- Minor improvements to margin requirement calculations in backtest

3.14.25 Version 0.6.6

Features

- Support of local data with portfolio strategies
- Backtest spread is now specified in price units for disambiguation
- Skip data warmup period to speed up backtests (specify `warmup_period` in autotrader.backtest) in continuous update mode
- Improved backtest printout
- All instruments will be passed to a portfolio strategy as a list using the `instrument` argument
- Instrument specific pip values can be provided when creating an order
- Improved Trade and Position `__repr__` methods

Fixes

- Bug with floating pnl calculation when running multi-instrument backtests
- Pagination of Oanda data retrieval

3.14.26 Version 0.6.5

Features

- General exception handling of bot updates in continuous mode

Fixes

- Link to documentation and website

3.14.27 Version 0.6.4

Fixes

- Autodetect divergence order of operations, timeseries indexing
- Specification and handling of ‘home_currency’ (provided through `configure`)
- Calculation of home conversion factors, and handling of oanda quote data

3.14.28 Version 0.6.3

Features

- Portfolio strategies: include `"PORTFOLIO": True` in your strategy configuration to signal that the strategy is a portfolio-based strategy. Doing so, data for each instrument in the watchlist will be passed to the strategy, allowing a single strategy to control multiple instruments at once, as in a portfolio. Currently supported for continuous mode only.
- Strategy configuration key `PARAMETERS` now optional.
- Autodetection of multiple instrument backtests for plotting.
- Option to select chart type (standard or portfolio) for single instrument backtests, via `AutoTrader.plot_settings()`.
- Option to specify `base_size` when creating an `Order`. This refers to the trade size calculated using the base currency, pre-conversion using the account's home currency (particularly useful for Forex traders).
- `modify` order types are now supported by the Oanda broker API, allowing a trader to change the take profit or stop loss attached to an open trade.

Fixes

- generalised `get_size` method of broker utilities to give correct results for non-FX instruments (in this case, SL price must be provided rather than SL distance).

3.14.29 Version 0.6.2

Features

- Named arguments for strategy initialisation: strategies must be constructed from named arguments "parameters", "data" and "instrument". Additionally, "broker" and "broker_utils", when including broker access, and "data_stream" when including data stream access. This change was made for disambiguation of input arguments.
- Improvements to `AutoPlot`, including autoscaling of indicator figures and backtest account history
- Addition of `BacktestResults` class, improving readability and accessibility of backtest results.

3.14.30 Version 0.6.1

Features

- Simpler imports: for example, `AutoTrader` can be imported using `from autotrader import AutoTrader`, instead of `from autotrader.autotrader import AutoTrader`. Likewise for `AutoPlot`, `GetData`, and trade objects (`Order`, `Trade`, `Position`).

Fixes

- Handling of close and reduce order types in `autobot`
- Assign UTC timezone to data after downloading from yfinance
- Fetch current positions from virtual broker after updating with latest data.
- Duplicate bar checking method in `autobot`

3.14.31 Version 0.6.0

- Interactive Brokers is now supported.
- Improvements to public broker methods for clarity.
- Comprehensive docstrings and type hints added.
- Distinction of broker and feed, allowing specification of broker and feed separately.
- New broker template directory added.
- All AutoTrader attributes have been made private to avoid confusion - the configuration methods should be used exclusively to set the attributes. This also clarifies and promotes visibility of public methods.
- New method `get_bots_deployed` added to AutoTrader.
- Project heirarchy: note changes in location of `autodata`, `indicators` and other modules previously in the `lib/` directory.
- Deprecated `help` and `usage` methods of AutoTrader (replaced by in-code docstrings).
- AutoTrader method `add_strategy` now accepts strategy classes as input argument, to directly provide strategy class objects.
- Broker public method name changes: `cancel_pending_order` to `cancel_order`, `get_pending_orders` to `get_orders`, `get_open_trades` to `get_trades`, `get_open_positions` to `get_positions`.
- Broker public method deleted: `get_cancelled_orders` - functionality available using `get_orders` method with `order_status = 'cancelled'`.
- To facilitate strategies built with prior autotrader versions, the previous format of signal dictionaries from strategy modules is still supported. Support for this format will be phased out in favour of the new `Order` and `Trade` objects (found in `autotrader.brokers.trading` module).
- For new Order, Trade and Position objects, support for legacy code is included via `as_dict` methods, to convert class objects to dictionaries.
- AutoTrader demo repository has been updated to reflect the changes above.
- Option to include/exclude positions from broker when updating strategy.
- Distinction of order/trade size and direction; size is now an absolute value representing the number of units to be traded, while direction specifies if the trade is long or short.
- Strategy module: method `generate_signal` is passed named arguments `i` and `current_position`.
- Continuous mode: single instantiation, iteration by timestamp
- AutoStream deprecated: if using streamed data, local data file paths should be provided using the `add_data` method.
- Abstracted data update method into `DataStream` class (within `autotrader.utilities` module) to allow custom data pipelines

- Ability to trade multiple contracts on an underlying asset (continuous mode only)
- Ability to use virtual broker in livetrade mode

3.14.32 Older Versions

For a changelog of versions prior to `v0.6.0`, please refer to the *older versions* changelog.

AutTrader Changelog (pre v0.6.0)

Version 0.5.0

Breaking change:

- virtual broker method 'get_open_positions' will now behave more as expected, returning the culmination of open trades for the specified instrument(s). Instead of returning a dictionary of open trades, a nested dictionary will be returned, containing the total position size held (long and short units), associated trade ID's and other information.

Fixes:

- margin calculations for multi-instrument backtests
- fix: MTF None type handling when optimising
- fix: MTF assignment error when providing custom data file
- fix: Heikin Ashi overwriting inputted price data
- fix: trailing stop behaviour in virtual broker, when specifying stop loss as a price.
- fix: added v20 dependency
- fix: stop loss filter will only be applied when there is a stop loss
- fix: overwrite of keys in strategy parameters

Features:

- Multi-instrument backtest data checking: datasets with mis-matched lengths are automatically corrected to improve backtest reliability.
- improved docstrings
- feat: added pivot point plot method to AutoPlot
- feat: added resampling method to AutoPlot to allow for MTF plotting
- feat: MTF support for local files
- feat: new indicators: divergence detection
- feat: improved divergence indicators
- feat: new indicator: halftrend
- feat: improved robustness of generic indicator line plotting
- feat: added capability to plot multiple indicator lines on same figure

0.5.32

- fix: trailing stops bug in virtual broker
- fix: pending order method in Oanda module

0.5.31

- fix: virtual broker is now more robust to bad data
- docs: added commission method to virtual broker, eventually to allow more complex commission schemes
- fix: autodetect_divergence now accepts `tolerance` argument

0.5.30

- docs: virtual broker `cancel_pending_order` method closer reflects Oanda method equivalent method
- feat: `add_data` allows specifying local `quote_data` for home conversion factor.

0.5.29

- feat: added trading session plot type, to show times of trading sessions in AutoPlot (indicator type `trading-session`)
- feat: added get_trade_details method to oanda module to match virtual broker

0.5.28

- feat: `instrument` key added to `signal_dict` to allow optionally trading other products from a strategy.

0.5.27

- fix: backtest dates will be adhered to (as close as possible) when providing local data
- feat: generalised `plot_backtest` method
- feat: (beta) ability to specify chart candle timeframe via `plot_settings` to plot MTF strategies on timeframes other than the base timeframe. This feature is useful to reduce the chart filesize by plotting on higher timeframe candles.

0.5.26

- feat: AutoTrader analyse backtest methods are now simpler to use, requiring only the bot as an input.
- feat: `bot.backtest_summary` now includes an `account_history` key, containing a DataFrame with the time-history of the trading account balance, NAV, margin available and drawdown.

0.5.25

- fix: order submission time error when backtesting with verbosity
- feat: empty signal dicts are now accepted when no order is to be submitted

0.5.24

- fix: order SL and TP filter for limit and stop-limit order types
- feat: added option to show/hide cancelled orders in AutoPlot

0.5.23

- feat: added shaded bands plotting to AutoPlot

0.5.22

- feat: added total trading fees to trade summary

0.5.21

- feat: added `add_data` method to conveniently provide local price data files.

0.5.20

- feat: added `order_type: modify` to virtual broker, to allow dynamically updating stop losses and take profits (Issue 11). **This order type is not yet supported in the Oanda module.**

0.5.19

- fix: AutoPlot attribute error

0.5.18

- feat: added Jupyter notebook config option to AutoTrader

0.5.17

- feat: added Jupyter notebook flag to AutoPlot to allow inline plotting
- fix: duplicate data will be deleted when downloading

0.5.16

- fix: Oanda `get_open_positions()` more reflective of virtual broker, added (incomplete) method for `get_open_trades()`

0.5.15

- fix: default setting of limit stop loss type

0.5.14

- fix: overwrite of keys in strategy parameters: risk_pc, granularity, sizing
  and period. If these keys exist already, they will no longer be overwritten

0.5.13

- fix: data alignment verification method when using MTF data

0.5.12

- feat: added signal plotting method to IndiView ('type': 'signals')
- feat: improved multibot backtest axis labelling
- docs: changed virtual broker update order in backtest to improve order executions

0.5.11

- fix: stop loss filter will only be applied when there is a stop loss

0.5.10

- feat: improved robustness of generic indicator line plotting
- feat: added capability to plot multiple indicator lines on same figure

0.5.9

- fix: added v20 dependency

0.5.8

- fix: trailing stop behaviour in virtual broker, when specifying stop loss as a price.
- feat: new indicator: halftrend

0.5.7

- fix: Heikin Ashi overwriting inputted price data

0.5.6

- feat: improved divergence indicators

0.5.5

- feat: new indicators: divergence detection

0.5.4

- feat: MTF support for local files

0.5.3

- fix: MTF assignment error when providing custom data file
- feat: added pivot point plot method to AutoPlot
- feat: added resampling method to AutoPlot to allow for MTF plotting

0.5.2

- fix: MTF None type handling when optimising

0.5.1

- fix: margin available will update upon initial deposit
- improved docstrings

Version 0.4.0

- Livetrade mode now supports bot detachment, so that bots will trade until a termination signal is received. This is achieved through the bot manager.
- Data time alginment can optionally be disabled
- Various plotting improvements

0.4.27

- Feature: added trade unit precision method to oanda

0.4.26

- Changed links on pypi

0.4.25

- Added website/github link on pypi

0.4.24

- Added pip location method to Oanda API module

0.4.23

- Fix: rounding of position sizing in broker utils

0.4.22

- Fix: added small pause between opening new plot from scan results to

0.4.21

- Added generic emailing method 'send_message' to easily send custom emails

0.4.20

- Added position retrieval from Oanda

0.4.19

- Plotting enhancements
- Improvements to scan mode

0.4.18

- various stream fixes

0.4.17

- Stream will check instrument of tick data to attempt to fix price bug

0.4.16

- Stream will run indefinitely until manually stopped, to ensure bots using stream will not be prematurely terminated

0.4.15

- Improved exception handling

0.4.14

- Ability to suspend bots and stream when livetrading. This is useful for weekends / closed trading period, where trading is not possible, but you do not wish to kill an active bot.
- Various stream connection improvements
- Docstring improvements

0.4.13

- Added 3 second sleep when reconnecting to Oanda API

0.4.12

- fix typo in connection check

0.4.11

- Added connection check to Oanda module

0.4.10

- Major improvements to AutoStream
- Livetrade with data updates directly from stream
- If running in detached bot mode, must include initialise_strategy(data) method in strategy module so that it can recieve data updates from the bot
- Must also have exit_strategy(i) method in strategy module, to allow safe strategy termination from bot manager

3.15 License

which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

```
a) The work must carry prominent notices stating that you modified
it, and giving a relevant date.

b) The work must carry prominent notices stating that it is
released under this License and any conditions added under section
7.  This requirement modifies the requirement in section 4 to
"keep intact all notices".
```

(continues on next page)

(continued from previous page)

```
c) You must license the entire work, as a whole, under this
License to anyone who comes into possession of a copy.  This
License will therefore apply, along with any applicable section 7
additional terms, to the whole of the work, and all its parts,
regardless of how they are packaged.  This License gives no
permission to license the work in any other way, but it does not
invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display
Appropriate Legal Notices; however, if the Program has interactive
interfaces that do not display Appropriate Legal Notices, your
work need not make them do so.
```

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

```
a) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by the
Corresponding Source fixed on a durable physical medium
customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by a
written offer, valid for at least three years and valid for as
long as you offer spare parts or customer support for that product
model, to give anyone who possesses the object code either (1) a
copy of the Corresponding Source for all the software in the
product that is covered by this License, on a durable physical
medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this
conveying of source, or (2) access to copy the
Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the
written offer to provide the Corresponding Source.  This
alternative is allowed only occasionally and noncommercially, and
only if you received the object code with such an offer, in accord
with subsection 6b.

d) Convey the object code by offering access from a designated
place (gratis or for a charge), and offer equivalent access to the
Corresponding Source in the same way through the same place at no
further charge.  You need not require recipients to copy the
Corresponding Source along with the object code.  If the place to
```

(continues on next page)

(continued from previous page)

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

```
a) Disclaiming warranty or limiting liability differently from the
terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or
author attributions in that material or in the Appropriate Legal
Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or
requiring that modified versions of such material be marked in
reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or
authors of the material; or

e) Declining to grant rights under trademark law for use of some
trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that
material by anyone who conveys the material (or modified versions of
it) with contractual assumptions of liability to the recipient, for
any liability that these contractual assumptions directly impose on
those licensors and authors.
```

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRO-

INDEX

Symbols