TradingBot Documentation

Release 1.0.0

Alberto Cardellini

Jul 29, 2020

Contents

CHAPTER 1

Introduction

TradingBot is an autonomous trading system that uses customised strategies to trade in the London Stock Exchange market. This documentation provides an overview of the system, explaining how to create new trading strategies and how to integrate them with TradingBot. Explore the next sections for a detailed documentation of each module too.

1.1 System Overview

TradingBot is a python program with the goal to automate the trading of stocks in the London Stock Exchange market. It is designed around the idea that to trade in the stock market you need a **strategy**: a strategy is a set of rules that define the conditions where to buy, sell or hold a certain market. TradingBot design lets the user implement a custom strategy without the trouble of developing all the boring stuff to make it work.

The following sections give an overview of the main components that compose TradingBot.

1.1.1 TradingBot

TradingBot is the main entiy used to initialised all the components that will be used during the main routine. It reads the configuration file and the credentials file, it creates the configured strategy instance, the broker interface and it handle the processing of the markets with the active strategy.

1.1.2 Broker interface

TradingBot requires an interface with an executive broker in order to open and close trades in the market. The broker interface is initialised in the `TradingBot` module and it should be independent from its underlying implementation.

At the current status, the only supported broker is IGIndex. This broker provides a very good set of API to analyse the market and manage the account. TradingBot makes also use of other 3rd party services to fetch market data such as price snapshot or technical indicators.

1.1.3 Strategy

The `Strategy` is the core of the TradingBot system. It is a generic template class that can be extended with custom functions to execute trades according to the personalised strategy.

How to use your own strategy

Anyone can create a new strategy from scratch in a few simple steps. With your own strategy you can define your own set of rules to decide whether to buy, sell or hold a specific market.

1. Setup your development environment (see `README.md`)
2. Create a new python module inside the Strategy folder :

```
cd Strategies
touch my_strategy.py
```

3. Edit the file and add a basic strategy template like the following:

```
import os
import inspect
import sys
import logging

# Required for correct import path
currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.
↪currentframe())))
parentdir = os.path.dirname(currentdir)
sys.path.insert(0,parentdir)

from Components.Utils import Utils, Interval, TradeDirection
from .Strategy import Strategy
# Import any other required module

class my_strategy(Strategy): # Extends Strategy module
    """
    Description of the strategy
    """
    def read_configuration(self, config):
        # Read from the config json and store config parameters
        pass

    def initialise(self):
        # Initialise the strategy
        pass

    def fetch_datapoints(self, market):
        """
        Fetch any required datapoints (historic prices, indicators, etc.).
        The object returned by this function is passed to the 'find_trade_signal()
↪'
        function 'datapoints' argument
        """
        # As an example, this means the strategy needs 50 data point of
        # of past prices from the 1-hour chart of the market
        return self.broker.get_prices(market.epic, Interval.HOUR, 50)
```

(continues on next page)

(continued from previous page)

```
    def find_trade_signal(self, market, prices):
        # Here is where you want to implement your own code!
        # The market instance provide information of the market to analyse while
        # the prices dictionary contains the required price datapoints
        # Returns the trade direction, stop level and limit level
        # As an examle:
        return TradeDirection.BUY, 90, 150

    def backtest(self, market, start_date, end_date):
        # This is still a work in progress
        # The idea here is to perform a backtest of the strategy for the given␣
↪market
        return {"balance": balance, "trades": trades}
```

4. Add the implementation for these functions:
   - *read_configuration*: `config` is the json object loaded from the `config.json` file
   - *initialise*: initialise the strategy or any internal members
   - *fetch_datapoints*: fetch the required past price datapoints
   - *find_trade_signal*: it is the core of your custom strategy, here you can use the broker interface to decide if trade the given epic
   - *backtest*: backtest the strategy for a market within the date range
5. `Strategy` parent class provides a `Broker` type internal member that can be accessed with `self.broker`. This member is the TradingBot broker interface and provide functions to fetch market data, historic prices and technical indicators. See the *Modules* section for more details.
6. `Strategy` parent class provides access to another internal member that list the current open position for the configured account. Access it with `self.positions`.
7. Edit the `StrategyFactory` module inporting the new strategy and adding its name to the `StrategyNames` enum. Then add it to the *make* function
8. Edit the `config.json` adding a new section for your strategy parameters
9. Create a unit test for your strategy
10. Share your strategy creating a Pull Request :)

1.2 Modules

TradingBot is composed by different modules organised by their nature. Each section of this document provide a description of the module meaning along with the documentation of its internal members.

1.2.1 TradingBot

class TradingBot.**TradingBot**(*time_provider: Optional[tradingbot.Components.TimeProvider.TimeProvider] = None*, *config_filepath: Optional[pathlib.Path] = None*)

Class that initialise and hold references of main components like the broker interface, the strategy or the epic_ids list

backtest(*market_id: str*, *start_date: str*, *end_date: str*, *epic_id: Optional[str] = None*) → None

Backtest a market using the configured strategy

close_open_positions() → None
Closes all the open positions in the account

process_market(*market: tradingbot.Interfaces.Market.Market, open_positions: List[tradingbot.Interfaces.Position.Position]*) → None
Spin the strategy on all the markets

process_market_source() → None
Process markets from the configured market source

process_open_positions() → None
Fetch open positions markets and run the strategy against them closing the trades if required

process_trade(*market: tradingbot.Interfaces.Market.Market, direction: tradingbot.Components.Utils.TradeDirection, limit: Optional[float], stop: Optional[float], open_positions: List[tradingbot.Interfaces.Position.Position]*) → None
Process a trade checking if it is a "close position" trade or a new trade

safety_checks() → None
Perform some safety checks before running the strategy against the next market

Raise exceptions if not safe to trade

setup_logging() → None
Setup the global logging settings

start() → None
Starts the TradingBot main loop - process open positions - process markets from market source - wait for configured wait time - start over

1.2.2 Components

The `Components` module contains the components that provides services used by TradingBot.

Broker

The `Broker` class is the wrapper of all the trading services and provides the main interface for the `strategies` to access market data and perform trades.

AbstractInterfaces

class Components.Broker.AbstractInterfaces.**AbstractInterface**(*config: tradingbot.Components.Configuration.Configuratio*

class Components.Broker.AbstractInterfaces.**AccountInterface**(*config: tradingbot.Components.Configuration.Configuration*

class Components.Broker.AbstractInterfaces.**StocksInterface**(*config: tradingbot.Components.Configuration.Configuration*)

IGInterface

class Components.Broker.IGInterface.**IGInterface**(*config: tradingbot.Components.Configuration.Configuration*)
IG broker interface class, provides functions to use the IG REST API

authenticate() → bool
Authenticate the IGInterface instance with the configured credentials

close_all_positions() → bool
Try to close all the account open positions.

- Returns **False** if an error occurs otherwise True

close_position(*position: tradingbot.Interfaces.Position.Position*) → bool
Close the given market position

- **position**: position json object obtained from IG API
- Returns **False** if an error occurs otherwise True

confirm_order(*dealRef: str*) → bool
Confirm an order from a dealing reference

- **dealRef**: dealing reference to confirm
- Returns **False** if an error occurs otherwise True

get_account_balances() → Tuple[Optional[float], Optional[float]]
Returns a tuple (balance, deposit) for the account in use

- Returns **(None,None)** if an error occurs otherwise (balance, deposit)

get_account_used_perc() → Optional[float]
Fetch the percentage of available balance is currently used

- Returns the percentage of account used over total available amount

get_market_info(*epic_id: str*) → tradingbot.Interfaces.Market.Market
Returns info for the given market including a price snapshot

- **epic_id**: market epic as string
- Returns **None** if an error occurs otherwise the json returned by IG API

get_markets_from_watchlist(*name: str*) → List[tradingbot.Interfaces.Market.Market]
Get the list of markets included in the watchlist

- **name**: name of the watchlist

get_open_positions() → List[tradingbot.Interfaces.Position.Position]
Returns the account open positions in an json object

- Returns the json object returned by the IG API

get_positions_map() → Dict[str, int]
Returns a *dict* containing the account open positions in the form {string: int} where the string is defined as 'marketId-tradeDirection' and the int is the trade size

- Returns **None** if an error occurs otherwise a dict(string:int)

navigate_market_node(*node_id: str*) → Dict[str, Any]
Navigate the market node id

- Returns the json representing the market node

search_market(*search: str*) → List[tradingbot.Interfaces.Market.Market]
Returns a list of markets that matched the search string

set_default_account(*accountId: str*) → bool
Sets the IG account to use

- **accountId**: String representing the accound id to use

- Returns **False** if an error occurs otherwise True

trade(*epic_id: str*, *trade_direction: tradingbot.Components.Utils.TradeDirection*, *limit: float*, *stop: float*) → bool
Try to open a new trade for the given epic

- **epic_id**: market epic as string
- **trade_direction**: BUY or SELL
- **limit**: limit level
- **stop**: stop level
- Returns **False** if an error occurs otherwise True

Enums

class Components.Broker.IGInterface.**IG_API_URL**
IG REST API urls

AVInterface

class Components.Broker.AVInterface.**AVInterface**(*config: tradingbot.Components.Configuration.Configuration*)
AlphaVantage interface class, provides methods to call AlphaVantage API and return the result in useful format handling possible errors.

daily(*marketId: str*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the Daily time series for the given market

- **marketId**: string representing an AlphaVantage compatible market id
- Returns **None** if an error occurs otherwise the pandas dataframe

intraday(*marketId: str*, *interval: Components.Broker.AVInterface.AVInterval*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the Intraday time series for the given market

- **marketId**: string representing an AlphaVantage compatible market id
- **interval**: string representing an AlphaVantage interval type
- Returns **None** if an error occurs otherwise the pandas dataframe

macd(*marketId: str*, *interval: Components.Broker.AVInterface.AVInterval*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the MACDEXT tech indicator series for the given market

- **marketId**: string representing an AlphaVantage compatible market id
- **interval**: string representing an AlphaVantage interval type
- Returns **None** if an error occurs otherwise the pandas dataframe

macdext(*marketId: str*, *interval: Components.Broker.AVInterface.AVInterval*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the MACDEXT tech indicator series for the given market

- **marketId**: string representing an AlphaVantage compatible market id
- **interval**: string representing an AlphaVantage interval type
- Returns **None** if an error occurs otherwise the pandas dataframe

quote_endpoint (*market_id: str*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the Quote Endpoint data for the given market

- **market_id**: string representing the market id to fetch data of
- Returns **None** if an error occurs otherwise the pandas dataframe

weekly (*marketId: str*) → pandas.core.frame.DataFrame
Calls AlphaVantage API and return the Weekly time series for the given market

- **marketId**: string representing an AlphaVantage compatible market id
- Returns **None** if an error occurs otherwise the pandas dataframe

Enums

class Components.Broker.AVInterface.**AVInterval**
AlphaVantage interval types: '1min', '5min', '15min', '30min', '60min'

YFinanceInterface

class Components.Broker.YFinanceInterface.**YFInterval**
An enumeration.

Broker

class Components.Broker.Broker.**Broker** (*factory: tradingbot.Components.Broker.BrokerFactory.BrokerFactory*)
This class provides a template interface for all those broker related actions/tasks wrapping the actual implementation class internally

close_all_positions () → bool
Attempt to close all the current open positions

close_position (*position: tradingbot.Interfaces.Position.Position*) → bool
Attempt to close the requested open position

get_account_used_perc () → Optional[float]
Returns the account used value in percentage

get_macd (*market: tradingbot.Interfaces.Market.Market*, *interval: tradingbot.Components.Utils.Interval*, *datapoints_range: int*) → tradingbot.Interfaces.MarketMACD.MarketMACD
Return a pandas dataframe containing MACD technical indicator for the requested market with requested interval

get_market_info (*market_id: str*) → tradingbot.Interfaces.Market.Market
Return the last available snapshot of the requested market

get_markets_from_watchlist (*watchlist_name: str*) → List[tradingbot.Interfaces.Market.Market]
Return a name list of the markets in the required watchlist

get_open_positions () → List[tradingbot.Interfaces.Position.Position]
Returns the current open positions

get_prices (*market: tradingbot.Interfaces.Market.Market*, *interval: tradingbot.Components.Utils.Interval*, *data_range: int*) → tradingbot.Interfaces.MarketHistory.MarketHistory
Returns past prices for the given market

- market: market to query prices for
- interval: resolution of the time series: minute, hours, etc.
- data_range: amount of past datapoint to fetch
- Returns the MarketHistory instance

navigate_market_node (*node_id: str*) → Dict[str, Any]
Return the children nodes of the requested node

search_market (*search: str*) → List[tradingbot.Interfaces.Market.Market]
Search for a market from a search string

trade (*market_id: str*, *trade_direction: tradingbot.Components.Utils.TradeDirection*, *limit: float*, *stop: float*)
Request a trade of the given market

BrokerFactory

class Components.Broker.BrokerFactory.**BrokerFactory** (*config: tradingbot.Components.Configuration.Configuration*)

class Components.Broker.BrokerFactory.**InterfaceNames**
An enumeration.

MarketProvider

class Components.MarketProvider.**MarketProvider** (*config: tradingbot.Components.Configuration.Configuration*, *broker: tradingbot.Components.Broker.Broker.Broker*)
Provide markets from different sources based on configuration. Supports market lists, dynamic market exploration or watchlists

get_market_from_epic (*epic: str*) → tradingbot.Interfaces.Market.Market
Given a market epic id returns the related market snapshot

next () → tradingbot.Interfaces.Market.Market
Return the next market from the configured source

reset () → None
Reset internal market pointer to the beginning

search_market (*search: str*) → tradingbot.Interfaces.Market.Market
Tries to find the market which id matches the given search string. If successful return the market snapshot. Raise an exception when multiple markets match the search string

Enums

class Components.MarketProvider.**MarketSource**
Available market sources: local file list, watch list, market navigation through API, etc.

TimeProvider

class Components.TimeProvider.**TimeProvider**
Class that handle functions dependents on actual time such as wait, sleep or compute date/time operations

get_seconds_to_market_opening (*from_time: datetime.datetime*) → float
Return the amount of seconds from now to the next market opening, taking into account UK bank holidays and weekends

is_market_open (*timezone: str*) → bool
Return True if the market is open, false otherwise

- **timezone**: string representing the timezone

wait_for (*time_amount_type: Components.TimeProvider.TimeAmount*, *amount: float = -1.0*) → None
Wait for the specified amount of time. An TimeAmount type can be specified

Enums

class Components.TimeProvider.**TimeAmount**
Types of amount of time to wait for

Backtester

class Components.Backtester.**Backtester** (*broker: tradingbot.Components.Broker.Broker.Broker, strategy: Union[tradingbot.Strategies.SimpleMACD.SimpleMACD, tradingbot.Strategies.WeightedAvgPeak.WeightedAvgPeak]*)
Provides capability to backtest markets on a defined range of time

print_results () → None
Print backtest result in log file

start (*market: tradingbot.Interfaces.Market.Market*, *start_dt: datetime.datetime*, *end_dt: datetime.datetime*) → None
Backtest the given market within the specified range

Configuration

class Components.Configuration.**Configuration** (*dictionary: Dict[str, Any]*)

Utils

class Components.Utils.**Utils**
Utility class containing static methods to perform simple general actions

static humanize_time (*secs: Union[int, float]*) → str
Convert the given time (in seconds) into a readable format hh:mm:ss

static is_between (*time: str, time_range: Tuple[str, str]*)
Return True if time is between the time_range. time must be a string. time_range must be a tuple (a,b) where a and b are strings in format 'HH:MM'

static macd_df_from_list (*price_list: List[float]*) → pandas.core.frame.DataFrame
Return a MACD pandas dataframe with columns "MACD", "Signal" and "Hist

static midpoint (*p1: Union[int, float], p2: Union[int, float]*) → Union[int, float]
Return the midpoint

static percentage (*part: Union[int, float], whole: Union[int, float]*) → Union[int, float]
Return the percentage value of the part on the whole

static percentage_of(*percent: Union[int, float], whole: Union[int, float]*) → Union[int, float]
Return the value of the percentage on the whole

Enums

class Components.Utils.**TradeDirection**
Enumeration that represents the trade direction in the market: NONE means no action to take.

class Components.Utils.**Interval**
Time intervals for price and technical indicators requests

Exceptions

class Components.Utils.**MarketClosedException**
Error to notify that the market is currently closed

class Components.Utils.**NotSafeToTradeException**
Error to notify that it is not safe to trade

1.2.3 Interfaces

The Interfaces module contains all the interfaces used to exchange information between different TradingBot components. The purpose of this module is have clear internal API and avoid integration errors.

Market

class Interfaces.Market.**Market**
Represent a tradable market with latest price information

MarketHistory

class Interfaces.MarketHistory.**MarketHistory**(*market: tradingbot.Interfaces.Market.Market, date: List[str], high: List[float], low: List[float], close: List[float], volume: List[float]*)

MarketMACD

class Interfaces.MarketMACD.**MarketMACD**(*market: tradingbot.Interfaces.Market.Market, date: List[str], macd: List[float], signal: List[float], hist: List[float]*)

Position

class Interfaces.Position.**Position**(***kargs*)

1.2.4 Strategies

The `Strategies` module contains the strategies used by TradingBot to analyse the markets. The `Strategy` class is the parent from where any custom strategy **must** inherit from. The other modules described here are strategies available in TradingBot.

Strategy

class Strategies.Strategy.**Strategy**(*config: tradingbot.Components.Configuration.Configuration*, *broker: tradingbot.Components.Broker.Broker.Broker*)

Generic strategy template to use as a parent class for custom strategies.

run(*market: tradingbot.Interfaces.Market.Market*) → Tuple[tradingbot.Components.Utils.TradeDirection, Optional[float], Optional[float]]

Run the strategy against the specified market

set_open_positions(*positions: List[tradingbot.Interfaces.Position.Position]*) → None

Set the account open positions

StrategyFactory

class Strategies.StrategyFactory.**StrategyFactory**(*config: tradingbot.Components.Configuration.Configuration*, *broker: tradingbot.Components.Broker.Broker.Broker*)

Factory class to create instances of Strategies. The class provide an interface to instantiate new objects of a given Strategy name

make_from_configuration() → Union[tradingbot.Strategies.SimpleMACD.SimpleMACD, tradingbot.Strategies.WeightedAvgPeak.WeightedAvgPeak]

Create and return an instance of the Strategy class as configured in the configuration file

make_strategy(*strategy_name: str*) → Union[tradingbot.Strategies.SimpleMACD.SimpleMACD, tradingbot.Strategies.WeightedAvgPeak.WeightedAvgPeak]

Create and return an instance of the Strategy class specified by the strategy_name

- **strategy_name**: name of the strategy as defined in the json config file
- Returns an instance of the requested Strategy or None if an error occurres

SimpleMACD

class Strategies.SimpleMACD.**SimpleMACD**(*config: tradingbot.Components.Configuration.Configuration*, *broker: tradingbot.Components.Broker.Broker.Broker*)

Strategy that use the MACD technical indicator of a market to decide whether to buy, sell or hold. Buy when the MACD cross over the MACD signal. Sell when the MACD cross below the MACD signal.

backtest(*market: tradingbot.Interfaces.Market.Market*, *start_date: datetime.datetime*, *end_date: datetime.datetime*) → Dict[str, Union[float, List[Tuple[str, tradingbot.Components.Utils.TradeDirection, float]]]]

Backtest the strategy

calculate_stop_limit(*tradeDirection: tradingbot.Components.Utils.TradeDirection*, *current_offer: float*, *current_bid: float*, *limit_perc: float*, *stop_perc: float*) → Tuple[float, float]

Calculate the stop and limit levels from the given percentages

fetch_datapoints(*market: tradingbot.Interfaces.Market.Market*) → tradingbot.Interfaces.MarketMACD.MarketMACD
Fetch historic MACD data

find_trade_signal(*market: tradingbot.Interfaces.Market.Market*, *datapoints: tradingbot.Interfaces.MarketMACD.MarketMACD*) → Tuple[tradingbot.Components.Utils.TradeDirection, Optional[float], Optional[float]]
Calculate the MACD of the previous days and find a cross between MACD and MACD signal

- **market**: Market object
- **datapoints**: datapoints used to analyse the market
- Returns TradeDirection, limit_level, stop_level or TradeDirection.NONE, None, None

initialise() → None
Initialise SimpleMACD strategy

read_configuration(*config: tradingbot.Components.Configuration.Configuration*) → None
Read the json configuration

Weighted Average Peak Detection

class Strategies.WeightedAvgPeak.**WeightedAvgPeak**(*config: tradingbot.Components.Configuration.Configuration*, *broker: tradingbot.Components.Broker.Broker.Broker*)
All credits of this strategy goes to GitHub user @tg12.

backtest(*market: tradingbot.Interfaces.Market.Market*, *start_date: datetime.datetime*, *end_date: datetime.datetime*) → Dict[str, Union[float, List[Tuple[str, tradingbot.Components.Utils.TradeDirection, float]]]]
Backtest the strategy

fetch_datapoints(*market: tradingbot.Interfaces.Market.Market*) → tradingbot.Interfaces.MarketHistory.MarketHistory
Fetch weekly prices of past 18 weeks

find_trade_signal(*market: tradingbot.Interfaces.Market.Market*, *datapoints: tradingbot.Interfaces.MarketHistory.MarketHistory*) → Tuple[tradingbot.Components.Utils.TradeDirection, Optional[float], Optional[float]]
TODO add description of strategy key points

initialise() → None
Initialise the strategy

peakdet(*v: numpy.ndarray*, *delta: float*, *x: Optional[numpy.ndarray] = None*) → Tuple[Optional[numpy.ndarray], Optional[numpy.ndarray]]
Converted from MATLAB script at http://billauer.co.il/peakdet.html

Returns two arrays

function [maxtab, mintab]=peakdet(v, delta, x) %PEAKDET Detect peaks in a vector % [MAXTAB, MINTAB] = PEAKDET(V, DELTA) finds the local % maxima and minima ("peaks") in the vector V. % MAXTAB and MINTAB consists of two columns. Column 1 % contains indices in V, and column 2 the found values. % % With [MAXTAB, MINTAB] = PEAKDET(V, DELTA, X) the indices % in MAXTAB and MINTAB are replaced with the corresponding % X-values. % % A point is considered a maximum peak if it has the maximal % value, and was preceded (to the left) by a value lower by % DELTA.

% Eli Billauer, 3.4.05 (Explicitly not copyrighted). % This function is released to the public domain; Any use is allowed.

read_configuration(*config: tradingbot.Components.Configuration.Configuration*) → None
Read the json configuration

weighted_avg_and_std(*values: numpy.ndarray*, *weights: numpy.ndarray*) → Tuple[float, float]
Return the weighted average and standard deviation.

values, weights – Numpy ndarrays with the same shape.

Python Module Index

c

i

s

t

Index

G

H

I

M

N

P

Q

R

S

T

U

W

Y