TradingBot Documentation

Release 1.0.0

Alberto Cardellini

Nov 16, 2019

CONTENTS

CHAPTER

ONE

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.Broker import Interval
from .Strategy import Strategy
from Utility.Utils import Utils, TradeDirection
# 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)

    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
```

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

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

```
  def make_strategy(self, strategy_name):
      if strategy_name == StrategyNames.SIMPLE_MACD.value:
          return SimpleMACD(self.config, self.broker)
      elif strategy_name == StrategyNames.FAIG.value:
          return FAIG_iqr(self.config, self.broker)
      elif strategy.name == StrateyNames.MY_STRATEGY.value:
          return MY_STRATEGY(self.config, self.broker)
      else:
          logging.error('Impossible to create strategy {}. It does not exist'.
↪format(strategy_name))
```

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

1.2.2 Components

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

IGInterface

Enums

AVInterface

Enums

Broker

Enums

MarketProvider

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

get_market_from_epic(*epic*)
Given a market epic id returns the related market snapshot

next()
Return the next market from the configured source

reset()
Reset internal market pointer to the beginning

search_market(*search*)
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

Enums

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

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*, *broker*)
Generic strategy template to use as a parent class for custom strategies.

backtest(*market*, *start_date*, *end_date*)
Must override

fetch_datapoints(*market*)
Must override

find_trade_signal(*epic_id*, *prices*)
Must override

initialise()
Must override

read_configuration(*config*)
Must override

run(*market*)
Run the strategy against the specified market

set_open_positions(*positions*)
Set the account open positions

StrategyFactory

SimpleMACD

Weighted Average Peak Detection

1.2.5 Utils

Common utility classes and methods

Utils

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

static humanize_time(*secs*)
Convert the given time (in seconds) into a readable format hh:mm:ss

static is_between(*time*, *time_range*)
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 midpoint(*p1*, *p2*)
Return the midpoint

static percentage(*part*, *whole*)
Return the percentage value of the part on the whole

static percentage_of(*percent*, *whole*)
Return the value of the percentage on the whole

Enums

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

Exceptions

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

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

1.3 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

1.3.1 [1.2.0] - 2019-11-16

Added

- Added backtesting feature
- Created Components module
- Added TimeProvider module
- Added setup.py to handle installation

Changed

- Updated CI configuration to test the Docker image
- Updated the Docker image with TradingBot already installed

1.3.2 [1.1.0] - 2019-09-01

Changed

- Replaced bash script with python
- Moved sources in `src` installation folder
- Corrected IGInterface numpy dependency
- Added Pipenv integration
- Exported logic from Custom Strategy to simplify workflow
- Added dev-requirements.txt for retro compatibility
- Updated Sphinx documentation

1.3.3 [1.0.1] - 2019-05-09

Changed

- Updated renovate configuration

1.3.4 [1.0.0] - 2019-04-21

Added

- Initial release

PYTHON MODULE INDEX

INDEX